## A ADDITIONAL RELATED WORK

There is relatively little related work in this space. Some authors have focused specifically on asymptotic efficiency (specifically big-O complexity) from several perspectives: identifying what makes big-O hard [16], tools for big-O practice [22], alternate approaches to teaching big-O [11, 19] and more. These are quite removed from our focus. A few authors have focused on trying to identify misconceptions in algorithm analysis and data structures [4, 6]. These are closer to our work but still focused on algorithmic topics rather than basic programs, and on abstract measures of complexity.

## B ADDITIONAL STUDY PROGRAMMING LANGUAGE CONSIDERATIONS

We could have created unfamiliarity by using a made-up pseudocode. However, that would create many new confounds. Students could be guessing about the meaning of the syntax in the absence of concrete knowledge of the language, and their answers might merely reflect their guesses. The answers might also reflect the extent to which the constructs reminded them of other languages they already knew, and that could be difficult to trace. Furthermore, if they responded saying that they didn't know, (a) those could be statements of ignorance about the language rather than of concepts; and (b) those answers might obscure misconceptions that they do suffer from (and that surfaced in our study). Therefore, we feel Racket was useful for this initial study. Nevertheless, it would be important to perform many more studies of this sort (section 13); our study can serve as a baseline for other such studies (section 14).

## C DETAILS OF STUDY INSTRUMENTS

Here we describe the study instruments. We provide a superset of the instruments with accompanying ground-truth performance explanations. In appendix C.1 and appendix C.2 we explain the differences between years.

The heart of the instrument is three sets of *behaviorally identical* programs, followed by questions designed to learn about student perceptions. The programs are shown in fig. 1, fig. 2, and fig. 3.

We will use the term BOT for big-O time complexity, and ART for actual running time (what is often called "wall-clock time", but we avoid the abbreviation WCT to avoid potential confusion with "worst-case time").

**Program Set 1** `cond` is a multi-armed conditional, akin to an `if ... else if ... else if ... else ...` construct in other languages.

All three versions expect to consume a list. If the list has no elements, they perform `P`. If it has one element, they perform `Q`. If it has more than one element, they perform `R`. Note that their BOT is identical.

In terms of actual running time and space, versions (A) and (C) are literally equivalent: the Racket compiler translates `cond` into a sequence of cascading `if`s. Therefore, both programs will expand into exactly the same program. Version (B) *appears* to do more work: the question `(cons? x)` (which asks whether `x` is a non-empty list) appears to be syntactically extra. However, because Racket is a safe language, the operation `(rest x)`, which operates only on non-empty lists, will first check whether `x` is empty: i.e., it will perform a `(cons? x)` check. Depending on various aspects of the compiler, it can recognize that this check is redundant, therefore performing it only once: thereby rendering (B) to be exactly the same code as (A) and (C).

In short, the correct answer to any time comparison is "(A) and (C) are identical in ART, and (B) may or may not be depending on specifics of the implementation strategy, its current configuration (e.g., optimization level), and current state (e.g., in case of a just-in-time compiler)". However, students would not necessarily know about the expansion of `cond`. Thus, ignorance would be very reasonable here.

It is also worth noting that the ART difference, even if present, could be minuscule. The function `f` as written may not be recursive (it depends, in particular, on the code in `R`). If it is not recursive, then the extra check is performed only once independent of the size of the list. Given contemporary tag representations, the value `x` already being in a register, and so on, it is unclear this difference can even be measured if the check does not occur frequently.

**Program Set 2** In this case, we compare a manual implementation of a list-length function, and the built-in implementation. We chose this pair to understand student conceptions of built-in functions. Clearly, again, their BOT is the same.

In terms of ART, is a built-in function necessarily faster? It depends! A built-in could, of course, exploit low-level representations that are not available at the source level. Therefore, it could potentially be much faster. However, it could also be slower for multiple reasons. It could have a much more general implementation: e.g., it may take arbitrary arity arguments, which—depending on the both the source and implementation [5]—could incur a large penalty at every call-site; the single-arity version avoids that penalty. It may also need to perform more safety checks that were assumed away in the manual implementation. Therefore, with the information provided, it is impossible to be sure.

**Program Set 3** In this case, we sum the numbers in a list through either a recursive function or a higher-order function. This pair effectively refines the previous pair: here we are probing student conceptions of built-in *higher-order* functions.

We intentionally choose the higher-order function `foldr`: its counterpart `foldl` is tail-recursive, and therefore would not correspond to version (A). But `foldr` has the same (non-tail-recursive) recursion structure as the `sum` function.

Again, both programs have the same BOT. In terms of ART, the same argument applies as above. The addition of a higher-order function does not substantially alter the situation. A particularly knowledgeable person might note that, in Racket, whereas `length` can take only one argument—obviating one of the ambiguity-inducing factors for Program Set 2—`foldr` can indeed take any number of lists, potentially reinforcing the calling penalty. (None of our students, however, made any such arguments.) Thus, again, with the information provided, we cannot say for sure.

## C.1 Version from 2020

*Programs Used.* Our initial study consisted of only Program Sets 1 and 2. Furthermore, Program Set 1 had only two versions, (A) and (B).

*Instrument Details.* For each Program Set, students were asked to pair-wise describe which they thought was "more efficient" than the other, with four choices: each one more efficient than the other, both equally efficient, and "I don't know" (IDK). Because the two Program Sets each had only two programs, there was only one comparison question per Set.

After each Set, students were asked:

> Please explain your choice. If you chose one of the first three options (not "I don't know"), please explain in as much detail as you can, step by step, why your answer is correct. BE SURE TO take into account the transformations a compiler performs on code and the optimizations an architecture performs during execution that would apply in this setting.

(emphasis in original). The last portion was meant to trigger the IoED, given that we expected that most students had little to no knowledge of the internals of compilers.

After responding to the two Program Sets (and hopefully having their IoED for these programs triggered), students were shown the *same* Program Sets, in the same order, again, and this time asked only to provide the choices. We would hope to see a movement away from firm answers to IDK.

## C.2 Version from 2021: First Iteration

After the 2020 iteration, we observed ways in which we could strengthen our instrument (and also corrected a benign typo). The changes do not materially change the interpretation of S20; they only add depth so we can better interpret our findings. These were the changes:

(1) We asked students up front whether they knew "anything at all about program efficiency". Only students who answered affirmatively proceeded to the rest of the instrument.
(2) We added version (C) to Program Set 1. Note that to a reader who understands the language (or even similar concepts from other languages), this program is effectively *syntactically* identical to version (A). This was added to study the extent to which students would continue to believe they found differences even in programs that are truly identical.
(3) Because Program Set 1 had three programs, we added two more "more efficient" questions to cover all three comparisons (1A–1B, 1A–1C, and 1B–1C).

(4) With each question, students were also asked to state “How confident do you feel about your opinion?” on a 7-point Likert-like scale from “Not confident at all” to “Very confident”.
(5) We added Program Set 3 to see whether their misconceptions about library functions carried over to higher-order functions. It was conceivable that, with little prior experience with higher-order functions, students would be much more likely to confess ignorance.

In 2020, we saw that most students provided only speculative verbal answers, not well-grounded in any technical details. We also anticipated that the IDKs would have little to say (per our instructions), but found some students answering the question anyway. (We analyze all responses in detail in section 11.)

In 2021, we therefore changed our strategy. After each Program Set we asked students to respond textually only if they had an opinion:

> For questions where you didn’t choose “I don’t know”, please explain your choice in as much detail as you can.

Note that this does not include the text about compilers and architecture, which had been roundly ignored.

Instead, we chose to make the compilers aspect extremely manifest. After students had responded to the three Program Sets the first time, we provided a list of *18 different compiler optimizations* (e.g., “Basic-block merging”, “Last-call replacement”, “Touch optimization”); students were asked to check off which of these they knew “well”.

The hope was that this explicit enumeration would achieve the IoED triggering that the previous version had failed to accomplish. Students were then asked to respond to the same three Program Sets again, without textual answers: just choosing which, if any, might be faster, and re-stating their confidence. The goal of IoED is to at least reduce their confidence in their beliefs.

## C.3 Version from 2021: Second Iteration

After the first iteration was finished, students were sent a refutation text (appendix J). This was shared again before the second iteration. In addition, it was again linked to the top of the second iteration instrument. Students were asked to confirm that they had read the linked document before continuing (though of course some may not have).

For the second iteration, conducted late into the semester, the instrument began with:

> Topic of Focus
> Like the earlier quiz, we are again dealing with “program efficiency”. Over the course of the semester we’ve extensively studied Big-O as a measure of efficiency. But in this quiz we’re asking you to NOT discuss Big-O but relate to your other notions of efficiency instead.

It also asked them what “efficiency” meant to them “now that the semester is over”.

The second iteration did not employ IoED. Instead the goal was to measure the impact of the refutation texts. For each Program Set, students were therefore asked to:

(1) Choose pairwise between the programs.
(2) Express their confidence for each pairwise choice.
(3) If they didn’t choose “I don’t know”, to explain their choice “in as much detail as you can”.

Our hope was that the refutation texts would cause students to largely choose IDK, so we were most interested in the case they chose differently.[1]

[1]In retrospect, it would have been wise to ask them why in either case, to confirm that they actually used the refutation text and picked the *right* reason.

## D DETAILS OF ANALYSIS METHODS

Multiple questions ask students for written feedback. We assessed their prose response using a rubric. For both rubrics, two authors generated a draft rubric; another author (who is significantly more experienced with qualitative methods, the course content, and the programming language performance concepts) reviewed it and provided feedback. The two authors revised the rubric, as they felt appropriate, using the feedback. They then applied it to a portion of the data and computed inter-rater reliability using Cohen's $\kappa$ [2]. This process iterated until they obtained a rubric that was both satisfactory to the other author and achieved a high enough reliability score. After this, the two authors split up and applied the rubric to the entire set of student responses, and lightly cross-checked each others' work.

Students were asked for their conceptions of efficiency on all three administrations. In two iterations, the team was able to obtain a $\kappa$ of 0.88. This rubric is shown in fig. 4.

For the program pairs, an earlier (pre-2020) version of the instrument had not had multiple-choice answers for students to express their opinion about the relative efficiency of the different versions, leading to answers that were difficult to analyze (and in some cases even decipher). For that reason, the instrument used in this paper was designed to make analyzing their overall conception easy while still providing rich narrative information.

Concretely, their opinions about the versions are easy to check just by counting. As we have noted above, all pairs of programs are identical by BOT; in terms of ART, they are either the same (because they are effectively the same program) or we cannot be certain without knowing a great deal more about the innards of the implementation. By design, there is never a case where one program is certain to be meaningfully faster than the other.

Even if we consider other measures of "efficiency" such as readability (section 10), there are still few substantial differences, and even here the answers are not obvious and open to interpretation. For instance, consider versions (A) and (B) from Program Set 1. It may seem that (A) is more readable because it's shorter. However, as *How to Design Programs* explains, (B) is arguably better because the structure of the program matches the structure of the datum; by collapsing cases, (A) makes the program harder to trace and hence maintain. Similarly, in the comparison of (B) and (C), (B) makes the type of `x` explicit in the `else` case, which can be an improvement for certain kinds of readability.

In addition to their rankings, students also provided written textual responses. The rubric for this is presented in fig. 5. This rubric obtained a $\kappa$ of 0.839 after five iterations.

## E MORE ABOUT EFFICIENCY CONCEPTIONS

Figure 4 shows what conceptions students had about efficiency.

We do see some differences between S21-1 and F21-2. While these could be a function of population differences, some of these changes are also explicable in the course context. The course asked students to consider (during lecture, not on any assignments) performance factors such as network utilization, power, and so on. The course also heavily assessed students on code readability and style. These could explain the increases in MEM, READ, MISC, and POW. The exposure to big-O analysis on functional programs, and to the refutation texts, could explain the reduction in STEP. Nevertheless, the dominance of big-O time analysis on the assignments may explain the rise of TIME.

## F MORE ABOUT PROGRAM RANDINGS

In presenting our findings we have chosen to not compute statistical significance, which we discuss in section 14. All %ages are rounded to nearest whole number for simplicity. Thus some sums may be very slightly above or below 100%.

### F.1 Results from S20

Recall that students were *not* asked whether they had any understanding of efficiency. In 2020, 22% of students said they "don't really" have prior opinions on program efficiency (section 6). One might therefore assume that these students would mark most of their answers as IDK. Yet they did not!

We present student responses in the following format. Each row gives the comparison of one program pair. The first column (e.g., "1A–1B") lists the pair. The second shows the percentage who expected the left entry (in this example, "1A") to be faster than that on the right ("1B"). The third shows how many had the opposite expectation. The fourth shows how many expected the two were equal. The last shows those who said they didn't know.

These are the results the first time they were shown the programs:

| Pair | L > R | R > L | Equal | IDK |
|---|---|---|---|---|
| 1A–1B | 74% | 8% | 12% | 7% |
| 2A–2B | 6% | 45% | 29% | 20% |

Their written responses, written after they make their choices, were meant to trigger an IoED reaction. These are their results from the second time they saw the same programs:

| Pair | L > R | R > L | Equal | IDK |
|---|---|---|---|---|
| 1A–1B | 70% | 11% | 11% | 8% |
| 2A–2B | 6% | 47% | 28% | 19% |

For direct comparison, we subtract the first percentage from the second. Recall that we would hope to see IDK significantly increase and the others to decrease. Instead we see:

| Pair | L > R | R > L | Equal | IDK |
|---|---|---|---|---|
| 1A–1B | -4% | +3% | -1% | +1% |
| 2A–2B | 0% | +2% | -1% | -1% |

which is essentially no real change at all. Perhaps more surprisingly, a small number of students switched from choosing one to the other being more efficient, without any particular reason for why. This suggests that perhaps their beliefs are not stable and firmly rooted—but they are still switching, in effect, between two wrong answers.

### F.2 Results from S21-1

Recall (appendix C.2) that students were asked if they knew about program efficiency; those who answered "no" were not invited to respond further. Our results are from the 58% (i.e., $N = 35$) that felt they had an opinion.

These are their preliminary rankings:

| Pair | L > R | R > L | Equal | IDK |
|---|---|---|---|---|
| 1A–1B | 83% | 6% | 11% | 0% |
| 1A–1C | 26% | 9% | 63% | 3% |
| 1B–1C | 3% | 74% | 23% | 0% |
| 2A–2B | 0% | 57% | 23% | 20% |
| 3A–3B | 6% | 49% | 31% | 14% |

Recall that 1A and 1C are the same program. Indeed, 63% recognize them as equal, but 35% do not; in particular, 26% are misled by the longer syntactic length (echoing [12] and [23]).

After this, they were asked to indicate which compiler optimizations they knew. 80% of the students indicated they knew none. Five were chosen by no students. Only two ("Dead code/store elimination" and "Inlining") had three students claim to recognize them; the remainder had only two or one. Therefore, most students admitted to little or no knowledge of the inner workings of programming language implementations.

When they were asked to rank programs again, their responses were:

| Pair | L > R | R > L | Equal | IDK |
|---:|---:|---:|---:|---:|
| 1A–1B | 80% | 6% | 14% | 0% |
| 1A–1C | 31% | 9% | 57% | 3% |
| 1B–1C | 6% | 69% | 26% | 0% |
| 2A–2B | 3% | 54% | 27% | 14% |
| 3A–3B | 9% | 46% | 31% | 14% |

Again, we subtract the second from the first, hoping to see IDK be positive and the others negative:

| Pair | L > R | R > L | Equal | IDK |
|---:|---:|---:|---:|---:|
| 1A–1B | +3% | 0% | -3% | 0% |
| 1A–1C | -5% | 0% | +6% | 0% |
| 1B–1C | -3% | +5% | -3% | 0% |
| 2A–2B | -3% | +3% | -4% | +6% |
| 3A–3B | -3% | +3% | 0% | 0% |

We see almost none of that, with opinion just moving between the other columns in most cases. Indeed, their belief in the equivalence of 1A and 1C seems to go *down* (positive value)!

Recall that we also asked them for their confidence in their view. Their confidence was given on a 7-point Likert-like scale. We can summarize their confidence by simply computing the average weighted sum. Here we show the rating confidence for each pair. We also show the increase (second minus first), hoping to see *negative* values (i.e., a *decrease*):

| Pair | Round 1 | Round 2 | Increase |
|---:|---:|---:|---:|
| 1A–1B | 4.86 | 5.00 | +0.14 |
| 1A–1C | 4.40 | 4.57 | +0.17 |
| 1B–1C | 4.74 | 4.85 | +0.11 |
| 2A–2B | 4.00 | 4.20 | +0.20 |
| 3A–3B | 4.09 | 4.09 | 0 |

Instead, we see that, after being confronted with a long list of compiler optimizations that they did *not* know, students seem to have *increased* in their confidence in assessing efficiency.

Despite these outcomes, there could be a silver lining in the data: it would be heartening if the confidence resided purely in students who responded IDK, and those who thought they did know exhibited very low confidence. Unfortunately, that is not the case:

| Pair | IDKs |
|---|---|
| 1A–1B | there were no IDKs |
| 1A–1C | the only IDK had a confidence of 2 |
| 1B–1C | there were no IDKs |
| 2A–2B | the IDKs had an average confidence of 3.40 |
| 3A–3B | the IDKs had an average confidence of 2.80 |

We have examined the data from several angles and found that in general, the moral seems to be clear. Students hold views that are either questionable or outright incorrect; they are confident in their views; and their confidence seems largely unshakeable.

## F.3 Results from F21-2

Three months after the previous iteration, students were again asked to respond to the same questions. In the intervening time students had been exposed (section 4) to many ideas new to them in computer science, had become proficient at (early stage) big-O analysis, and had learned about real-world considerations when applying big-O. Still, they were being asked to respond *not* in terms of big-O (appendix C.3). Their ratings were as follows:

| Pair | L > R | R > L | Equal | IDK |
|---|---|---|---|---|
| 1A–1B | 45% | 7% | 29% | 19% |
| 1A–1C | 10% | 3% | 71% | 16% |
| 1B–1C | 3% | 58% | 23% | 16% |
| 2A–2B | 0% | 10% | 58% | 32% |
| 3A–3B | 10% | 13% | 55% | 23% |

Here, even though students were told to not use big-O analysis, their familiarity with it would have led them to better understand program behavior. This likely explains the better (but not perfect) results for 1A–1C.

Fortunately, we see many more students unclear on their answer, perhaps caused by exposure to the refutation texts (but possibly also by a semester of complex material reducing their certainty). The greater uncertainty for 2A–2B and 3A–3B could be because both programs had features directly referenced in the refutation text (fig. 7).

Because we have more students expressing doubt, we examine the confidence figures by population. Here again we may see some impact of the refutation text:

| Pair | Confidence of IDKs | Confidence of Not-IDKs |
|---|---|---|
| 1A–1B | 4.67 | 4.96 |
| 1A–1C | 5.00 | 5.04 |
| 1B–1C | 5.00 | 4.96 |
| 2A–2B | 5.60 | 4.91 |
| 3A–3B | 5.42 | 5.21 |

We note that their confidence is fairly high even when misplaced, but refutation texts may have some value, at least when programs *visibly* use features that they reference.

Most tellingly, in a space given for final remarks, one student wrote:

> Overall it felt as if most of these were in the doc and it was difficult to draw concrete conclusions. In the document it was said that a lot of things were "not necessarily" true, which doesn't really give me the means to draw relevant conclusions (I think).

This was the *only* student who had chosen IDK across the board, but had also given all those scores a confidence of 1 (lowest). There is a telling disconnect between the student feeling all the programs were covered by the refutation text (which they were!), but feeling like they cannot draw conclusions from those very explanations. We applaud this student for their intellectual honesty, which at least did not lead to misplaced confidence (though in this case, their confidence would have been absolutely appropriate!).

## G MORE ABOUT RANKING EXPLANATIONS

We now explain three salient numbers in fig. 5:

- 28.8% of students in S20 reference compilers, which seems a surprisingly high knowledge of this topic. In fact, this is an artifact of the prompt (appendix C.1): the actual answers do not demonstrate any deep understanding of or engagement with them; many answers even suggest they have little to no understanding of what a compiler *is* (yet this did not prevent them from writing about one!). For instance, answers include:
  - "Efficiency is a word I'd use to compare the speed at which a compiler reads and performs the intended actions of code."
  - "[Efficiency is] the number of steps needed to evaluate the code by the compiler."
  - "[Efficiency] means the amount of operations that the compiler has to perform on a program to produce a result"
- 16.7% of F21-2 students reference Racket. After lengthy engagement with another programming language (Pyret) and with functional programming, looking back at Racket code seems to have made them speak about it directly. Some answers also indicated ignorance of the innards of Racket.
- 51.8% of F21-2 students admit to ignorance. Many of these responses explicitly reference the refutation text. This would seem to be a success! Unfortunately, as the other codes show, many students admitted to ignorance based on the text, but proceeded to venture an opinion anyway. We discuss this more in section 14.

## H THREATS TO VALIDITY IN DETAIL

This work naturally has many threats to validity. We discuss them below.

*Internal Validity.* We have notable differences in populations across years, and only a subset of S21-1 also took F21-2. These make some comparisons across these instances tricky. In particular, having similar populations might alter some of the ratios: seemingly problematic differences might disappear, while seeming non-differences may now stand out. We discuss this more in section 14.

In both S20 and S21-1, students saw the *same* pairs of programs twice. This could cause some entrenchment of views that reduces the impact of the IoED intervention. Giving isomorphic programs instead might show greater impact from the intervention. However, what might seem mathematically isomorphic might have hidden factors (as seen, for instance, in variants [3, 24] of the Wason selection task [27]) that trigger different outcomes from humans, making the results hard to interpret, which is why we did not use this approach.

In general, we have sampled students on very few programs. It is just possible that the programs we picked are especially problematic, and many similar programs would not produce the same entrenched results. Even beyond correcting for this, it would be valuable to use many more programs, and in particular similar programs, to test the stability of student preferences, and to identify what syntactic characteristics might throw them off.

Concretely, we have used pairs 2A–2B and 3A–3B to understand student conceptions of built-in functions: `length` and `foldr`, respectively. One possible confound is that neither was truly built-in to the students: assuming they read the text, they would have seen implementations of both of those functions. They may well have assumed that the actual Racket implementation does exactly what the (simplified) book version does. Some students still did assume that built-ins might have special behaviors—e.g., access to internal representations—but not all did. More students might have if we had used built-ins that were truly opaque to them—though if they are not even implementable in the source language, then it becomes difficult to perform the style of comparison used in this paper.

*External Validity.* There are many factors that affect the generalizability of our findings. Naturally, our choice of peculiar programming language and perhaps non-standard pedagogy (section 4) is certain to have impact, especially on the F21-2 data. In addition, our student population is almost certainly atypical of students leaving secondary school. The fact that many of them have completed (and probably done well) on AP exams and the like might make them overconfident. Still, we believe there is value to examining the attitudes of such students, because it could have impact on what such courses and exams should and should not cover, and what misconceptions they should try to address.

*Ecological Validity.* The "programs" in our instruments are not actually *programs* but rather templates with placeholders. This in itself may make the task somewhat artificial. When given real and full programs, students would naturally simply be able to run and measure them, especially if their primary efficiency concern is a physical quantity such as time or memory. However, it is noteworthy to us that no students commented on this, nor did any give the impression of having tried to fill in the bodies with sample code and performed experiments.

## I ADDITIONAL DISCUSSION POINTS

*On Statistics.* We have chosen to not compute statistical significance for our differences. We believe doing so is highly unlikely to be meaningful. There are notable differences in populations between the years. In addition, even within 2021, F21-2 was an optional exercise. It is quite possible that the students who responded were the ones who felt they were learning most from the course, who may also be the ones who also had their beliefs most challenged, leading to more IDKs. However, they may have had any number of other motivations (e.g., it may have been students not doing too well and looking for more learning opportunities).

Whatever the case, we feel statistics would be a distraction here. The macro phenomena we see are very telling. Rather than squeeze more out of our data, we feel it would be much more productive to first perform similar studies across several other populations. In other words, despite the presence of numbers and percentages, we think of this work as primarily *qualitative*. Furthermore, we intend this as a *formative* study, meant to raise questions and provide initial evidence for their importance rather than to offer definitive answers.

*Other Study Designs.* There are many natural variations of our study that immediately come to mind. For instance, one can imagine changing the programming language to one more familiar to students—though this familiarity may not extend to having an accurate model of program execution, especially for languages that have sophisticated compilers. (Furthermore, their familiarity may make students even more confident about wrong answers.) We might also wonder about the impact of a completely artificial language that was flagged as such and designed to minimize interference from languages they know.

Perhaps more intriguing is to present students with different relationships between programs. All our programs were chosen to have little to no differences; what happens when we choose programs that *do* have notable differences? What if the programs are designed so that they are more expensive to execute but syntactically smaller, as in Gal-Ezer and Zur [12]? What if the programs are amenable to significant optimizations that students do not know about—do they posit a "super-compiler" (akin to the "superbug" [17]) or do they retain a WYSI interpretation?

*Other Study Populations.* Our study population is quite uncommon, reflecting (mostly) students who had strong secondary-school computing. What about students without such a background? Do the results correlate with background? In particular, though we did not gather precise demographic information, based on the students in the class, we know that it is largely (over 70–80%) male and White/Asian, which reflects the demographic of many AP CS classes. Could the misplaced confidence we find be an artifact of this demographic?

*Impact on Post-Secondary Courses.* Our course context is unusual in that while it is "introductory", (a) many students coming in have had the AP A material, which is roughly similar to (at least the earlier parts of) a typical CS1 course; and (b) the course itself covers algorithmic content (section 4) akin to, say, the first part of many post-secondary algorithms courses. Therefore, the phenomena we see may possibly also be found in more typical student populations taking a post-secondary algorithms course.[2]

The fact that many misconceptions persist at the end of two months of instruction in algorithms—F21-2 was conducted after students had studied and implemented multiple graph algorithms—suggests that simply taking a typical algorithms course does not erase these misconceptions. In particular, a purely mathematical approach like big-O does not seem to

[2]However, there many be many differentiating factors. One readily apparent from section 6 is that many of these students consult Web sites like Reddit and StackOverflow, which may not be true of students who are less aware of such sites, are not as confident in their computing abilities, or are turned off by the nature of comments on such sites.

(and indeed, perhaps should not) impact how students view low-level efficiency issues. If these problems are found more broadly, that suggests we should be rethinking our approach to algorithm education to better relate the mathematical and physical models (akin to experimental algorithmics journals).

More broadly, we wonder about the impact of students' experiences with the world they live in and the material they learn in courses. Just like physics students experience forces long before formal physics education, many computing students increasingly live in worlds with rich computation. Consider, for instance, a typical algorithmic topic: binary search, and a proof that logarithmic complexity is a lower bound. Yet many students have spent a decade or more using search engines, which clearly respond in constant time even as the Web keeps growing. (A *small* constant: as of March 2022, Google's Core Web Vitals [`https://web.dev/vitals/`] claims that pages should start interacting in 100ms.)

How do students reconcile these phenomena? Do they fail to even make a connection (a form of failure to transfer [25])? Do they think these are unconnected? Do they believe search engine companies have special resources (that somehow negate bounds)? Do they simply hold contradictory views simultaneously? We believe these phenomena need much more investigation, and the findings could reform traditional algorithms education.

*Role of Secondary Education.* We see that many of our students obtained their conceptions of efficiency from past teachers. Given the problematic conceptions we see here, and given that our students come from all across the USA and some other countries, this potentially raises questions about the role of secondary school computer science education. Are there dangers to a shallow exposure to post-secondary technical content? When such content is covered, should it be done alongside materials (such as refutation texts) that also help students understand the limitations of their knowledge? We believe this is an important topic that warrants further discussion.

## J REFUTATION TEXT

The refutation text is shown split across fig. 6, fig. 7, and fig. 8.

Several of you exhibited inaccurate beliefs in your responses on performance. I have therefore compiled the following write-up to clarify these for you. Each one has been given a brief "code name" (like **[HIDDEN]**) to make it easier to

reference later.
Note that these are focused strictly on the *actual running time* (the "wall clock time") of programs. They are *not* statements about the big-O performance or any other measures.

**Belief [HIDDEN]:**
There are no hidden operations lurking below primitive operations.

**Status:**
Not true!

**Explanation:**
The programming language may check data before performing operations. For instance, before Racket applies first, it checks that the parameter truly is a non-empty list (and reports an error if not; this is how it's able to report an error). Thus, what seems like a single operation may actually be multiple operations.

**Belief [REPEAT]:**
When an expression appears to have been evaluated to produce an answer, it must necessarily have been evaluated.

**Status:**
Not necessarily true!

**Explanation:**
If the implementation notices that an expression has been repeated, and the value cannot have changed between the two instances, it may store the value and reuse it rather than re-compute it (i.e., trade off space and time). This is especially tricky given [HIDDEN].

**Belief [LENGTH]:**
A line of code takes a single unit (constant) amount of time to run.

**Status:**
Not necessarily true!

**Explanation:**
It depends on what code is on that line. Think about the most extreme case: you can write an infinite loop in one line; do you think an infinite loop finishes in one unit of time? Actually, it doesn't finish at all! In short, fewer lines of code don't necessarily run faster than more lines. A function call on that line means it could take a long time.

(continued)

Fig. 6. Refutation Text, part 1

(continued)

---

**Belief [KNOWLEDGE]:**
A programming language implementation ("compiler") knows what we know.

**Status:**
Not at all!

**Explanation:**
We may have all sorts of ways of knowing something about a program: a comment, facts about mathematics, awareness of what a user will do, etc. An implementation may not know any of these things. Our knowledge could be something as simple as an assumption that a particular variable is of a certain type; we can then assume that and skip some steps, but an implementation can't necessarily do that (e.g., [HIDDEN]). Unless we have explicitly communicated our knowledge to the implementation, or we are certain that the implementation has a way of knowing the same thing we do, we can't assume anything about what it knows (and, indeed, should assume it doesn't know what we know).

---

**Belief [BUILT-IN]:**
Built-in operations are faster than the same behavior written by hand.

**Status:**
Not necessarily. It could go either way.

**Explanation:**
On the one hand, a built-in operation may have access to some low-level programming features that are not provided in the surface language. However, on the other hand, the built-in operation may contain many more checks for things that the human did not think to write. It may even have to check for behavior caused by language features that the human didn't even know existed! All those checks and guards can make the built-in operator "safer" to use, but will also create overhead that is not present in the manual implementation.

---

**Belief [HOF]:**
Using a built-in higher-order function is faster than writing the same behavior by hand as an explicit loop/recursive function without a higher-order function.

**Status:**
Not necessarily. It could go either way.

**Explanation:**
Using the higher-order function may lead to more concise code, but as we've noted above ([LENGTH]), that doesn't make it faster. Similarly, a higher-order function may be built-in, but that also doesn't necessarily make it faster (e.g., [HIDDEN]). It is certainly true that in general, a built-in higher-order function may be able to improve on a hand-written version of the same behavior. However, there is also a cost associated with calling the function that we pass in. That cost may or may not be greater than the savings from using the built-in. Some smarter implementations can remove the overhead introduced by that call entirely, enabling the program to be more concise but equally fast or faster, but not all implementations are (or can be) smart about that.

---

(continued)

Fig. 7. Refutation Text, part 2

(continued)

---

**Belief [BOUNDS]:**
The implementation (“compiler”) has ways of making things faster that we don’t know about.

**Status:**
Maybe. But we can also reason about when this might not be possible.

**Explanation:**
The implementation is still governed by laws of physics, mathematics, and so on. In computer science we have results known as “lower bounds” that prove that we can’t do certain things more efficiently than some minimum (assuming we want a correct answer!). The compiler, built-in functions, etc. can’t magically beat these bounds.

---

**Belief [MEMORY]:**
Programs that run faster use less memory.

**Status:**
No, it’s often an inverse relationship!

**Explanation:**
One of the most powerful ideas in computer science is the space-time tradeoff: that we can often make things faster by using more memory, or increase memory-efficiency by making them slower. So speed and memory are often in an inverse relationship.

---

**Belief [POWER]:**
Programs that run faster consume less power.

**Status:**
It depends.

**Explanation:**
How much power a program consumes depends on the specific instructions that are executed. It is often possible to find lower-power instructions that execute more slowly. Therefore, there can be a positive or negative correlation between speed and power.

---

Fig. 8. Refutation Text, part 3

## REFERENCES

[1] M. Cakir. 2008. Constructivist Approaches to Learning in Science and Their Implications for Science Pedagogy: A Literature Review. *Intl. J. Env. & Sci. Ed.* 3, 4 (2008), 193–206.

[2] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20 (1960), 37–46.

[3] Leda Cosmides and John Tooby. 1992. Cognitive Adaptions for Social Exchange. In *The Adapted Mind: Evolutionary Psychology and the Generation of Culture*, Leda Cosmides, John Tooby, and Jerome H. Barkow (Eds.). Oxford University Press.

[4] Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. 2012. Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 21–26. https://doi.org/10.1145/2157136.2157148

[5] R. Kent Dybvig and Robert Hieb. 1990. A New Approach to Procedures with Variable Arity. *Lisp and Symbolic Computation* 3, 3 (1990), 229–244.

[6] Mohammed F. Farghally, Kyu Han Koh, Jeremy V. Ernst, and Clifford A. Shaffer. 2017. Towards a Concept Inventory for Algorithm Analysis Topics. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/3017680.3017756

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs*. MIT Press. http://www.htdp.org/

[8] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. In *Communications of the ACM*.

[9] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *SIGCSE International Computing Education Research Conference*. 35–42. https://doi.org/10.1145/2632320.2632346

[10] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *ACM Technical Symposium on Computer Science Education*.

[11] Judith Gal-Ezer, Tamar Vilner, and Ela Zur. 2004. Teaching Algorithm Efficiency at CS1 Level: A Different Approach. *Computer Science Education* 14, 3 (2004), 235–248. https://doi.org/10.1080/0899340042000302736 arXiv:https://doi.org/10.1080/0899340042000302736

[12] Judith Gal-Ezer and Ela Zur. 2004. The efficiency of algorithms—misconceptions. *Computers & Education* 42, 3 (2004), 215–226. https://doi.org/10.1016/j.compedu.2003.07.004

[13] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. 2017. Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. *International Journal of Artificial Intelligence in Education* 27 (2017), 65–100. https://doi.org/10.1007/s40593-015-0080-x

[14] J. Longfield. 2009. Discrepant Teaching Events: Using an Inquiry Stance to Address Students' Misconceptions. *Intl. J. Teach. and Learn. in Higher Ed.* 21, 2 (2009), 266–271.

[15] Robert McCartney, Dennis J. Bouvier, Tzu-Yi Chen, Gary Lewandowski, Kate Sanders, Beth Simon, and Tammy VanDeGrift. 2009. Commonsense Computing (Episode 5): Algorithm Efficiency and Balloon Testing. In *International Workshop on Computing Education Research*. 51–62. https://doi.org/10.1145/1584322.1584330

[16] Miranda Parker and Colleen Lewis. 2013. Why is Big-O Analysis Hard?. In *International Conference on Computing Education Research*. 201–202. https://doi.org/10.1145/2526968.2526996

[17] Roy D. Pea. 1986. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research* 2, 1 (Feb. 1986). https://doi.org/10.2190/689T-1R2A-X4W4-29J2

[18] G. J. Posner, K. A. Strike, P. W. Hewson, and W. A. Gertzog. 1982. Accommodation of a Scientific Conception: Toward a Theory of Conceptual Change. *Sci. Edu.* 66, 2 (1982), 211–227.

[19] Constantine Roussos. 2004. Teaching Growth of Functions Using Equivalence Classes: An Alternative to Big O Notation. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (Norfolk, Virginia, USA) *(SIGCSE '04)*. Association for Computing Machinery, New York, NY, USA, 170–174. https://doi.org/10.1145/971300.971361

[20] Leonid Rozenblit and Frank Keil. 2002. The misunderstood limits of folk science: an illusion of explanatory depth. *Cognitive Science* 26 (2002), 521–562.

[21] L. Savion. 2009. Clinging to discredited beliefs: The larger cognitive story. *J. Schol. of Teach. and Learn.* 9, 1 (2009), 81–92.

[22] Rebecca Smith and Scott Rixner. 2020. Compigorithm: An Interactive Tool for Guided Practice of Complexity Analysis. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 363–369. https://doi.org/10.1145/3341525.3387390

[23] Ruth Stavy and Dina Tirosh. 1996. Intuitive rules in science and mathematics: the case of 'more of A – more of B'. *International Journal of Science Education* 18 (1996), 653–667. https://doi.org/10.1080/0950069960180602

[24] Keith Stenning and Michiel van Lambalgen. 2008. *Human Reasoning and Cognitive Science*. MIT Press.

[25] Edward L. Thorndike and Robert S. Woolworth. 1901. The influence of improvement in one mental function upon the efficiency of other functions. *Psychological Review* 8 (1901).

[26] Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73 (1990), 231–248.

[27] Peter Cathcart Wason. 1966. Reasoning. In *New Horizons in Psychology I*, B. M. Foss (Ed.). Penguin.

[28] Kristin M. Weingartner and Amy M. Masnick. 2019. Refutation texts: Implying the refutation of a scientific misconception can facilitate knowledge revision. *Contemp. Edu. Psych.* 58 (2019), 138–148.