Comparing and Contrasting Different Algorithms Leads to Increased Student Learning

Elizabeth Patitsas, Michelle Craig and Steve Easterbrook
University of Toronto
40 St George St.
Toronto ON M5S 2E4
{patitsas, mcraig, sme}@cs.toronto.edu

ABSTRACT

Comparing and contrasting different solution approaches is known in math education and cognitive science to increase student learning – what about CS? In this experiment, we replicated work from Rittle-Johnson and Star, using a pretest–intervention–posttest–follow-up design (n=241). Our intervention was an in-class workbook in CS2. A randomized half of students received questions in a compare-and-contrast style, seeing different code for different algorithms in parallel. The other half saw the same code questions sequentially, and evaluated them one at a time. Students in the former group performed better with regard to procedural knowledge (code reading & writing), and flexibility (generating, recognizing & evaluating multiple ways to solve a problem). The two groups performed equally on conceptual knowledge. Our results agree with those of Rittle-Johnson and Star, indicating that the existing work in this area generalizes to CS education.

Categories and Subject Descriptors

K.3.2 [**Computers and Information Science Education**]: Pedagogy, education research

General Terms

Experimentation

Keywords

Computer science education, CS2, worked examples

1. INTRODUCTION

Computer science is a field in which problems call for a great deal of creativity, and problem solvers need to generate, recognize and evaluate different ways of solving problems. However, in our experience, typical CS1 courses tend to show a limited view of the the discipline; due to resource-constraints and tradition, students are seldom challenge to solve problems in different ways, compare approaches, or do things differently.

For our students, CS2 is where we as educators first truly begin to show that there are different approaches to solving problems. We show different ways to sort, and different data structures to store information. The standard approach is sequential in presentation: *Here's an algorithm. Here's another algorithm. Here's a data structure. Here's another.*

While we as CS instructors aim to give students an understanding of how these algorithms and data structures relate, comparison is often done as an afterthought. More often, we compare algorithms and data structures to abstract benchmarks on performance: *"hashtable insertion is O(1)"* rather than *"hashtable insertion is faster than heap insertion"*.

And to be fair: from a cognitive load theory perspective, it makes sense to first teach, for example, one partitioning strategy for quicksort, and then to teach a different strategy, rather than to teach them in parallel. Yet, the evidence from the math education and cognitive science literature indicates that it is actually more effective to teach different approaches to solving a problem in parallel, comparing and contrasting them in the process.[1] For example, Loewenstein et al compared students who saw two novel case studies either sequentially, or in parallel, and found students were better at analyzing the case studies when they were presented in parallel [3]. They described the effect of learning the case studies in one batch as "analogical encoding" and relate it to analogical learning [3]. Unlike analogical learning, where students relate new information to existing schemas, in analogical encoding, the students create schemas by using the differences in the case studies to understand relational structures embedded within the cases.

In this study, we investigate the effect of comparing different solution approaches to programming questions in CS2. We build on the existing literature on comparing; our study is a replication of two empirical studies by Rittle-Johnson and Star. We hypothesize that comparing and contrasting is more effective than showing different approaches sequentially, consistent with those two studies [4, 5].

1.1 The Original Studies

1.1.1 The Algebra Study

In 2007, Rittle-Johnson and Star published a study of grade 7 students (n=70), using a pretest–intervention–posttest design [4]. For the intervention, students were paired; pairs

[1]Furthermore, the empirical evidence for cognitive load theory is mixed, if not negative [1, 2].

were randomly assigned to one of two conditions. In one condition, students studied two worked examples side-by-side; one example showed "Mandy's solution" to solving an algebra problem, and the other showed "Erica's solution", another valid approach. The students were asked why Mandy and Erica got the same solution, and why one might use Erica's approach, which had fewer steps.

The other condition saw the same two examples, but sequentially. On one page was Mandy's solution, and a question asking why one might use that approach. On the next page was Erica's solution, and a question about it.

Rittle-Johnson's and Star's pretest/posttest was written to probe students' knowledge using the following knowledge taxonomy [4], which is established in the math education literature [6]:

- *Procedural knowledge*, defined as "the ability to execute action sequences to solve problems, including the ability to adapt known procedures to novel problems (transfer)" [7]
  - Familiar problems, those that are isomorphic to those in the intervention
  - Transfer problems, new problems solvable with the methods seen in the intervention
- *Procedural flexibility*, also known simply as flexibility: knowledge of different ways of solving problems and when to use them [8]
  - Generating multiple methods to solve a problem
  - Recognizing multiple methods to solve a problem
  - Evaluating new methods
- *Conceptual knowledge*, defined as generalizable knowledge that is "an integrated and functional grasp of mathematical ideas" [8]

In their study, they found that the compare-and-contrast condition outperformed the sequential condition with regard to procedural knowledge and flexibility, and both groups performed equally well on conceptual knowledge gains [4]. This had been the first study to empirically demonstrate that comparing and contrasting was effective at improving learning in mathematics. Previously, it had only been known through experience reports spanning over two decades [4, 9, 10, 11]; and there was a body of research in cognitive science using very different contexts [3, 12, 13].

1.1.2 The Estimation Study

In 2008, Star and Rittle-Johnson replicated the algebra study with one on computational estimation [5] (n=157), to investigate whether the same effects would be seen in a situation where there is no right answer. While in algebra there are multiple ways to solve a problem to get the same right answer, in estimation there are numerous valid ways to make estimations.

For this study, they added a second posttest to their design to probe retention of knowledge, and otherwise used the same design. They found again that the compare-and-contrast group learnt more with regard to procedural knowledge and flexibility – and not conceptual knowledge.

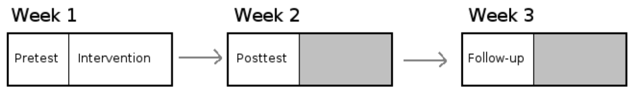


Figure 1: Our study design; the pretest, posttest and follow-up posttest were all the same.

1.1.3 This Study

From these studies and established cognitive science research, it is reasonable to expect that the same would be true if we were to ask questions about theoretical computer science. But what about programming? We know that reading code is full of structure that students have difficulty comprehending [14] – and high cognitive load as a result. A problem such as "estimate 10*27" has much less cognitive load than to read ten lines of code and write its output, or to write five lines of code.

Furthermore, a grade 7 student has experience with numbers and equations from the past seven years of mathematics education. A first-year computer science student has much less to draw on when presented code for the first time. As such, we were motivated to see whether replicating Rittle-Johnson and Star's studies with a programming example would produce different results.

To be consistent with the experiment being performed on inexperienced non-beginners, we chose to do our experiment in a first-year CS2 course. Basic algorithms and data structures provide a context rich in examples of different approaches to solving problems – for this study, we chose collision resolution of hashing.

2. METHODS

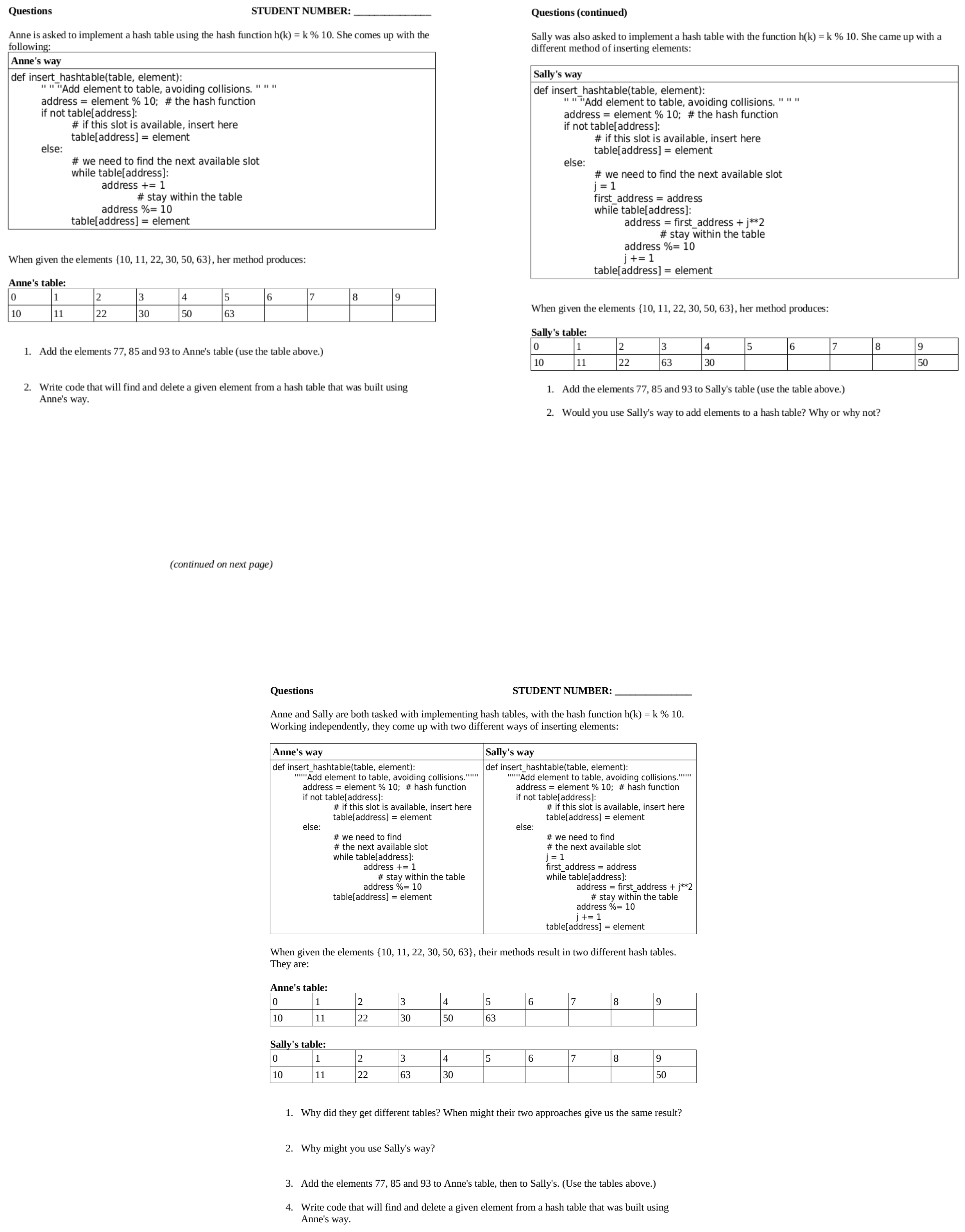
2.1 Study design

Our study took place in a CS2 class in Python at the University of Toronto in the spring of 2012; over 300 students were enrolled in the class, in three separate lecture sections[2]. Two lecture sections were taught by a senior teaching-track faculty member; the third section was taught by a graduate student.

Our study used a pretest–intervention–posttest–follow-up design (we refer to the pretest, intervention, posttest and follow-up as the four "portions" of the study). The pretest, posttest, and follow-up posttest were all the same. The intervention given was a workbook on hashtables, containing an introduction, worked examples, and questions. 241 students partook in at least one portion of the study, 83 of which participated in all four portions.

A randomly-distributed half of the class received workbooks where questions were presented sequentially (control group); the other half received workbooks where the questions were presented in a compare-and-contrast fashion (experimental group). Both groups received the same introduction and worked examples. The intervention was double-blind in that the researcher did not know which student had done which workbook, and students did not know whether

[2]Our study was performed with approval from the University of Toronto Office of Research Ethics, Protocol #27388.



Questions **STUDENT NUMBER: _______________**

Anne is asked to implement a hash table using the hash function h(k) = k % 10. She comes up with the following:

Anne's way

```
def insert_hashtable(table, element):
        " " "Add element to table, avoiding collisions. " " "
        address = element % 10;  # the hash function
        if not table[address]:
                # if this slot is available, insert here
                table[address] = element
        else:
                # we need to find the next available slot
                while table[address]:
                        address += 1
                             # stay within the table
                        address %= 10
                table[address] = element
```

When given the elements {10, 11, 22, 30, 50, 63}, her method produces:

Anne's table:

0	1	2	3	4	5	6	7	8	9
10	11	22	30	50	63				

1. Add the elements 77, 85 and 93 to Anne's table (use the table above.)
2. Write code that will find and delete a given element from a hash table that was built using Anne's way.

(continued on next page)

Questions (continued)

Sally was also asked to implement a hash table with the function h(k) = k % 10. She came up with a different method of inserting elements:

Sally's way

```
def insert_hashtable(table, element):
        " " "Add element to table, avoiding collisions. " " "
        address = element % 10;  # the hash function
        if not table[address]:
                # if this slot is available, insert here
                table[address] = element
        else:
                # we need to find the next available slot
                j = 1
                first_address = address
                while table[address]:
                        address = first_address + j**2
                                # stay within the table
                        address %= 10
                        j += 1
                table[address] = element
```

When given the elements {10, 11, 22, 30, 50, 63}, her method produces:

Sally's table:

0	1	2	3	4	5	6	7	8	9
10	11	22	63	30					50

1. Add the elements 77, 85 and 93 to Sally's table (use the table above.)
2. Would you use Sally's way to add elements to a hash table? Why or why not?

Questions **STUDENT NUMBER: _______________**

Anne and Sally are both tasked with implementing hash tables, with the hash function h(k) = k % 10. Working independently, they come up with two different ways of inserting elements:

Anne's way

```
def insert_hashtable(table, element):
        """"Add element to table, avoiding collisions.""""
        address = element % 10;  # hash function
        if not table[address]:
                # if this slot is available, insert here
                table[address] = element
        else:
                # we need to find
                # the next available slot
                while table[address]:
                        address += 1
                             # stay within the table
                        address %= 10
                table[address] = element
```

Sally's way

```
def insert_hashtable(table, element):
        """"Add element to table, avoiding collisions.""""
        address = element % 10;  # hash function
        if not table[address]:
                # if this slot is available, insert here
                table[address] = element
        else:
                # we need to find
                # the next available slot
                j = 1
                first_address = address
                while table[address]:
                        address = first_address + j**2
                             # stay within the table
                        address %= 10
                        j += 1
                table[address] = element
```

When given the elements {10, 11, 22, 30, 50, 63}, their methods result in two different hash tables. They are:

Anne's table:

0	1	2	3	4	5	6	7	8	9
10	11	22	30	50	63				

Sally's table:

0	1	2	3	4	5	6	7	8	9
10	11	22	63	30					50

1. Why did they get different tables? When might their two approaches give us the same result?
2. Why might you use Sally's way?
3. Add the elements 77, 85 and 93 to Anne's table, then to Sally's. (Use the tables above.)
4. Write code that will find and delete a given element from a hash table that was built using Anne's way.

Figure 2: Above: The questions of the control group's workbook; below: the experimental group

theirs was the control. Unlike Rittle-Johnson and Star's work, we did not pair students for the intervention.

Students were repeatedly told during the pretest, posttest, and follow-up posttest, to leave questions blank rather than guess. Each page of the tests also repeated this request, in bold. Students received no grades or other compensation for participating in the study; analysis was performed after the term had ended, and we did not have access to the mapping of student IDs (used to connect pretests to workbooks to posttests) to any other student information (name, final grade, etc).

The four portions of the study were delivered over a three week period, as shown in Figure 2. In the first week, we gave both pretest and the workbooks; in the second week, we gave only the posttest; in the third week we gave the follow-up. While we would have liked more time between the posttest and follow-up, we were constrained by the instructors' ability to fit in time for the study, as well as to administer each stage of the study to the three lecture sections on the same days. The third week of the study, it should be noted, was also the last week of the term.

2.2 Pre/Posttests

In their experiments, Rittle-Johnson and Star separated questions into whether they assessed procedural knowledge, flexibility, and conceptional knowledge. As a replication of their studies, our pretests were based on their assessments and categorizations (Table 1).We felt it was important to use the same knowledge taxonomy so that we could properly relate our findings to theirs. In their studies, procedural knowledge meant solving mathematics problems – here, it means reading and writing code. In both cases, the procedural knowledge represents the questions the students worked on for the interventions.

We piloted our questions informally on seven volunteer graduate students who had experience as TAs for the CS2 course we were probing, and two other TAs who had no Python experience (to ascertain how much guesswork could be used on the questions).

2.3 Workbook questions

After a two-page introduction to hash tables with worked examples, students' workbooks then contained four questions to complete on hash tables. (This is the only instruction students received on hash tables.) The control group received questions sequentially (shown in Figure 2):

1. Code to insert using linear probing
2. A question to insert some elements to a table
3. A question on writing code to delete elements from a table
4. Code to insert using quadratic probing
5. A question to insert some elements to a table
6. A question why one would or would not use this collision resolution method

The experimental group was asked the same or similar questions, but in a different order:

1. Code to insert using linear probing and to insert using quadratic probing was displayed side-by-side
2. The same questions to insert elements using linear probing and quadratic probing
3. The same question to write code to delete with linear probing
4. A question asking the students to compare and contrast the two methods (instead of why one would use or not use quadratic probing)

2.4 Data analysis

Pretests, workbooks, and posttests were first transcribed – each distinct response to each question was given a code. Codes were recorded along with the students' ID number to link their different assessments. Examples of codes here mean if two students hashed the numbers 41, 89, 23 and 55 as [41, 89, 23, 55, , , , ,] then they would hve the same code; if a student answered that question with [, , 23, , 41, 55, , , 89,], they would get a different code.

These numerous preliminary codes were then grouped and consolidated into a smaller, more workable set of new codes. This stage of encoding had a level of subjectivity to it, so the first author gave a fifth of the transcriptions to the other two authors to encode. The three authors then discussed their encodings until a consensus was reached. The first author then used the consensus on how to group codes to encode the rest of the transcriptions. Each code was then given a quantitative score. A total of 11 marks were available for procedural questions; 16 for flexibility questions; and 4 for conceptual questions.

The coding and marking of the pretests and posttests was done with a blind analysis [15] – we did not know which students were in which experimental condition when coding a given pre/posttest.

To determine whether the two conditions performed differently, we performed a hierarchical linear analysis, grouping students by section. This was done separately for each question type (procedural, flexibility, conceptual). The `nlme` package in `R` (version 2.15.1) was used to generate the hierarchical linear models [16]. In our interpretation of the results, we use the standard $p = 0.05$ threshold for statistical significance.

2.5 Hierarchical linear modeling

Hierarchical linear modeling (HLM) is also known as "multilevel modelling" and "mixed-effect modelling" [17]. It is used often in education studies for its ability to handle nested data structures, such as students being nested in schools or classes; ignoring this nesting and performing a regression analysis will result in an inflated Type I error [17]. It also allows us to test cross-level interactions, and the analysis allows for drop-out between tests. HLM also allows us to examine individual growth over time, unlike traditional ANOVA and ANCOVA methods [18].

3. PRETEST

As the procedural questions on the pretest could correctly be answered by any student with sufficient Python knowledge, students' scores on the pretest were fairly high – students could answer about 40% of the procedural questions on the pretest. Flexibility questions, however, were less likely to be known in advance (Table 2).

We began by first testing whether the experimental group's pretest scores were different from the control group's, on a section by section basis. There was no significant difference between the two groups within each section.

Problem type	Sample items	Scoring on sample item
Procedural knowledge (7 questions – 11 points)		
Familiar: code reading (4 Qs)	Add the elements 41, 89, 23 and 55 to the table [using the code above for linear probing].	1 pt for each correct answer
Transfer: code reading (2 Qs)	Add the elements 2, 9891, 30, 27, 33 and 34 for the table [using the code above for chaining].	1 pt for each correct answer
Transfer: code writing (1 Q)	Write code to remove a given element from the hash table [that uses chaining].	5 pts for code correctness
Flexibility (6 questions – 16 points)		
Generating multiple methods (2 Qs)	Hash the elements 12, 22, 33 and 42 in two different ways. [Table] What did you use for your hash function and collision resolution for way 1? Way 2?	1 pt for each reasonable way
Recognize multiple methods (2 Qs)	What are two ways of resolving collisions?	1 pt for each reasonable way
Evaluate non-conventional methods (2 Qs)	Heather comes up with a collision resolution strategy where if inserting an element would result in a collision, she uses a second hash function to try to hash the element. Is this a good idea?	1 point for a conclusion, 1 point for a correct argument
Conceptual knowledge (2 Qs – 4 points; 1 Q worth 1 pt, 1 Q worth 3 pts)		
Conceptual questions (2 Qs)	All our examples [with h(k) = k%10] had a 10-element list. What sort of hash function might you use if you had a 100-element list?	1 pt for "%100" or equivalent

Table 1: Sample items for assessing procedural knowledge, flexibility, and conceptual knowledge

Next, we compared the three lecture sections. No differences were found between the three lecture sections regarding procedural knowledge, and flexibility. However, for conceptual knowledge, the section taught by the graduate student performed significantly worse than the two lecture sections taught by the teaching-track faculty member (M = 22% vs 37%/39%; p = 0.047).

It is worth noting that while in aggregate, students in the experimental group had a higher mean on the pretest than the control group, the difference was not statistically significant.

4. POSTTEST RESULTS

For this portion of the analysis, we only used scores of participating students who completed the intervention. The analysis includes students who only completed one of the two posttests. Between the three tests, a total of 366 tests are used herein, from 165 students.

	Pretest		Posttest		Follow-up	
	M (%)	σ_{err}	M (%)	σ_{err}	M (%)	σ_{err}
Control (sequential)						
Procedural	37.6	2.4	50.6	2.7	50.6	3.3
Flexibility	13.2	2.5	47.1	3.7	49.4	4.4
Conceptual	32.6	4.0	64.7	3.8	65.4	4.8
Experimental (compare & contrast)						
Procedural	41.0	2.7	57.2	2.5	56.2	2.9
Flexibility	21.9	3.5	53.9	3.0	57.5	3.7
Conceptual	35.7	4.0	60.1	3.8	62.9	4.9

Table 2: Aggregate scores including all three sections, for students who wrote the pretest/intervention. Students included in this table may have written 0, 1, or 2 posttests.

4.1 Procedural knowledge

In our hierarchical model, we found that students in the experimental section demonstrated significantly more procedural knowledge on the two posttests (p = 0.02). Of the 11 points for procedural knowledge, the experimental section scored 0.61 points (6%) better than the control (4.1 vs. 3.5; $\sigma = 0.3$).

We found that while scores varied between lecture sections ($\sigma = 0.3$, scores are out of 11), that variation was not seen in the effect of treatment on procedural knowledge ($\sigma \cong 0.00$).

Students' measured procedural knowledge increased by 0.83 points out of 11 (7%) between posttests ($\sigma = 0.4$), indicating students were improving at reading and writing code. Two factors may be responsible. First, at this point in term, students may be better at reading and writing code. Second, that students were more familiar with the test and had written it previously.

4.2 Flexibility

As expected, the experimental group outperformed the control group on the two posttests (p = 0.047). The experimental group scored 0.12 points out of 16 better than the control group (0.33 vs. 0.45; $\sigma = 0.1$).

Like for procedural knowledge, this did significantly improve between posttests, but the effect was slight (on average, students gained 0.08 points with each test; $\sigma = 0.06$).

Posttest flexibility scores did not vary significantly between lecture sections ($\sigma \cong 0.0$), nor did the learning gains vary between lecture sections ($\sigma \cong 0.0$).

4.3 Conceptual knowledge

For conceptual knowledge, no significant effect was found between treatments (p = 0.860) in our hierarchical linear model. The control group had slightly, but non-significantly better scores on conceptual questions (0.89 points out of 4 vs. 0.87; $\sigma = 0.1$ for both).

The three lecture sections did not differ regarding conceptual knowledge gains on the posttests ($\sigma \cong 0.0$), although they began with different pretest scores.

And like in the other question categories, a small and significant learning effect was seen between posttests (each new test added 0.64 points out of 4 to students' scores; $\sigma = 0.09$).

5. STUDENTS WITHOUT PRETESTS AND INTERVENTIONS

Excluded in the previous section were students absent the day we gave the pretest and the intervention. However, a number of those students participated when we gave the posttests in the following weeks (n=24). For these students, their first posttest is effectively a pretest, but given later in the term than the rest of the class. And if these students wrote both of the posttests, we can use it to probe how much they learnt only from repeated testing, since the effect of the intervention will not be present in their scores. These students fall into three mutually exclusive categories:

Group 1 Those who did only the first posttest (n=9)
Group 2 Those who did only the second posttest (n=7)
Group 1+2 Those who did only the first *and* second posttests (n=8)

There are no significant differences between the scores of these three groups. For the students who did both posttests (Group 1+2), no significant gains are seen between the two tests (Table 3). This section acts as a sanity check of our results in the previous section[3].

	First posttest				Second posttest			
	G 1 ∪ 1+2		G 1+2		G 2		G 1+2	
	M	σ	M	σ	M	σ	M	σ
Proced.	44%	7%	60%	9%	33%	13%	41%	8%
Flexib.	18%	7%	21%	10%	20%	12%	22%	10%
Concep.	26%	9%	13%	13%	31%	14%	31%	16%

Table 3: Performance of students who did not write pretest/intervention, but wrote one (Group 1 and Group 2) or two posttests (Group 1+2).

These three groups of students had scores that were significantly like the scores of students who wrote the pretest. Although these students wrote their first test one or two weeks later than their peers, they did not perform better despite having had more time to learn how to read and write Python. Looking at Group 1+2, we see their scores did not increase between posttests. These students' second posttests are significantly similar to the wider class' pretests, and not significantly similar to the wider class' posttests. This indicates that the interventions had a real effect on the students who completed them, regardless of treatment.

6. DISCUSSION

Overall, we found that the experimental group outperformed the control group. The experimental group performed better on the posttests with regard to procedural knowledge, and flexibility – and the groups were tied on conceptual knowledge. Our findings indicate that having students compare-and-contrast different valid approaches to solving programming problems is more effective than showing the same approaches sequentially.

We found that the effect held in our second posttest, and that repeated testing resulted in higher scores with regard to procedural knowledge, flexibility, and conceptual knowledge. However, this effect was only seen in students who participated in the intervention – students who wrote only both the posttests did not see a significant gain between them. In short, while the control group learnt less than the experimental group, the control group still learnt more than those who had no intervention at all!

Looking at the students who only wrote posttests, we observed that scores were not dependent on how far along in the term the student was when they wrote their first test. This indicates that the learning effect between posttests seen in students who did the intervention was not due to increased practice with the overall course content, but due to increased practice with our particular activity.

It is not surprising that students' scores would increase with repeated testing [19] – this has been found in of itself to be effective in teaching.

6.1 Theoretical Understanding

So, why would comparing and contrasting different solutions facilitate learning? First, it aids students in identifying what is important in a problem, and what the different features are [13]; in a sense, this provides a road-map to the problem. If presented with only one piece of code, it is difficult for a novice to interpret which parts are important [14] – with more pieces of code a novice can construct patterns and schema.

This runs against what one would expect from a cognitive load theory perspective – by presenting more at once, this increases cognitive load. However, there is mixed evidence for cognitive load theory [1, 2] – and compare-and-contrast activities may help students abstract the code they are given, allowing them to handle a larger cognitive load.

In our study, the students who did the sequential workbooks did better than those who did no workbook at all – though our sample size for the latter group is very small. Interestingly, in Loewenstein et al's study, students given two sequential case studies to analyze learnt *less* than students not given any case studies at all [3]. It is possible that this difference is due to the study setting: our study was in situ, and theirs was a laboratory setting.

6.1.1 Conceptual knowledge

In our analysis, conceptual knowledge had different results than procedural knowledge, and flexibility. In the pretests, conceptual knowledge was the only category where we saw differences between lecture sections: the more experienced instructor's sections had better conceptual knowledge. Conceptual knowledge was also the only category where the experimential and control groups had similar gains from having an intervention done – both outperforming students who had neither intervention.

This was somewhat unexpected; existing literature has found that comparing and contrasting would lead to greater improvements in conceptual knowledge [20, 7]. However, Rittle-Johnson and Star's study had the same result as ours:

[3]This section has two notable threats to validity: the small sample size, and the likelihood that students who skipped 1-2 lectures may not be representative of the whole class.

conceptual knowledge was not increased by their compare condition [4]. Examining this more, they found in later work [5] that conceptual knowledge increases more in response to compare-and-contrast when a student already knows one way to solve the problem, but the effect is not seen when the student does not begin with that much knowledge.

To probe this, we compared students who scored at least one point on the pretest conceptual questions to students who scored no points.

For the students who began with conceptual knowledge, there was no overall significant difference between the control and experimental groups in terms of conceptual learning gains. However, the difference between treatments varied by lecture section: in the section taught by the graduate student, the gains in the experimental group were significantly more than those in the control group. For the other two lecture sections, there was no significant difference between conditions.

For the students who began without conceptual knowledge, those in the experimental group had better follow-up posttest scores than those who were in the control group. Again, the effect was strongest in the lecture section taught by the graduate student, as seen in Table 4.

Instructor/Group	Pretest		Posttest		Follow-up	
	M %	σ	M %	σ	M %	σ
Has pretest conceptual knowledge						
PhD's experimental	70	12	90	10	81	12
PhD's control	57	7	75	7	75	14
TTF's experimental	59	3	62	5	60	6
TTF's control	63	4	71	5	74	6
Does not have pretest conceptual knowledge						
PhD's experimental	0	0	65	7	73	11
PhD's control	0	0	50	10	50	16
TTF's experimental	0	0	45	9	56	10
TTF's control	0	0	58	5	49	6

Table 4: Scores on conceptual questions; PhD = graduate student; TTF = Teaching-track faculty.

These results are the opposite of what Rittle-Johnson and Star found. Here, the students who *began* without conceptual knowledge benefited more from the compare and contrast condition, and those who already had some conceptual knowledge did not respond differently to the two treatments! It appears that comparing-and-contrasting is even more useful to novices in CS than expected. It is also possible that this is more useful for novice instructors, given the effect was stronger in the graduate student's section.

6.2 Threats to Validity

If the study were to be repeated under similar constraints, we would perform the intervention on the same day as the first posttest, rather than on the same day of the pretest. Our experience was that upon completing a pretest, students were less inclined towards a workbook activity. They were probably less likely to read it in detail than if we had done the activity the week afterward, directly before the posttest.

With our large sample size, we are confident we have the necessary statistical power in our analysis; for a study with our level of groupings using HLM, n = 90 is considered sufficient [21][4].

By performing a blind analysis, and administering the tests and workbooks in a double-blind fashion, we are confident the effect of researcher bias is minimal. This study is a replication of two others[4, 5], and consistent with their findings. We hence can generalize to the existing theory that comparing and contrasting is more effective in mathematical fields than teaching different approaches sequentially.

What is more difficult to generalize, however, is whether our study represents *all* programming problems. Our study finds that comparing and contrasting programming problems is more effective than a sequential approach in our context – and, importantly, we can generalize our findings to the existing theory. What we have here is a strengthening of existing theory (analytical generalization), rather than a statistical generalization to all programming problems and all CS students. As such, we expect we would find the same result had it been code about binary trees, or linked lists.

But what if this were replicated with comparing and contrasting `for`-loops to `while`-loops in CS1? It is not as clear that we have the necessary construct validity to claim this to be the case. But our findings do strengthen the theory that would predict this to be the case.

Finally, it is worth noting the possibility that the students in the experimental condition could have had better learning gains in our study due to be being better learners, rather than due to the intervention. However, given the random assignment of students to conditions, and the large sample size of the study, we consider this to be unlikely.

6.3 Implications for instruction

We recommend to CS instructors to demonstrate multiple solution approaches while teaching programming – showing solutions in parallel to students in CS2 (and presumably above) appears to allow students to better differentiate important problem features and consider multiple methods [4]. In particular, having students compare and contrast consolidates these effects, and improves learning in multiple categories.

For CS2, where multiple approaches are typically shown sequentially, material can be often easily rearranged to discuss different algorithms and data structures in parallel. Comparing and contrasting as the material is taught is more likely to have students consider the different aspects of algorithms upfront.

Math teachers who have adopted teaching multiple different solution approaches have found another pedagogical benefit: by discussing different solutions with students "offered them access to their students' thinking and to misconceptions that they might be harboring" [9].

6.4 Future work

While we have shown in CS2 that comparing and contrasting different programming approaches leads to increased learning, it remains open whether this actually changes student perceptions. Verifying this theoretical expectation is the area of potential future work.

[4]There is no agreement as to how to calculate effect sizes for HLM [22]; however, there are rules of thumb for what a necessary sample size is.

Also worth examining is whether the effect we observed would hold in a CS1 class. Rittle-Johnson and Star, in their work, conclude that "students may need familiarity and fluency with a limited range of strategies before comparison of additional strategies aids knowledge of related concepts" [5], which would be absent at the beginning of CS1. Possible ideas would include comparing and contrasting `for`-loops to `while`-loops, or using arrays to having multiple variables.

7. CONCLUSIONS

In our study, we found that students who compared and contrasted different solutions to programming problems outperformed those who saw the same problems sequentially, with regard to procedural knowledge (reading and writing code), and flexibility (generating multiple methods to solve a problem, recognizing multiple methods, and evaluating multiple methods). No difference was found between the two groups in conceptual knowledge.

This indicates that teaching programming by demonstrating multiple solutions to a problem and comparing and contrasting them is more effective than the traditional approach of showing one solution at a time – in this case, teaching linear probing and quadratic probing in parallel was more effective than teaching them sequentially.

By teaching programming in this way, not only can we expect more learning gains, but it is more likely that students will see programming as a discipline where there is no magical "right answer" to every problem.

8. ACKNOWLEDGMENTS

Thank you to the instructors who allowed us to perform this study in their classes, and to the students who participated. The first author received funding from the National Science and Engineering Research Council of Canada. Also, a thank you to the anonymous reviewers for their detailed feedback on the manuscript.

9. REFERENCES

[1] Doug Holton. Cognitive load theory: Failure?, 2009.

[2] Ton de Jong. Cognitive load theory, educational research, and instructional design: some food for thought. *Instructional Science*, 38:105–134, 2010. 10.1007/s11251-009-9110-0.

[3] J. Loewenstein, L. Thompson, and D. Gentner. Analogical learning in negotiation teams: Comparing cases promotes learning and transfer. *Academy of Management Learning & Education*, 2(2):119–127, 2003.

[4] Bethany Rittle-Johnson and Jon R. Star. Does comparing solution methods facilitate conceptual and procedural knowledge? An experimental study on learning to solve equations. *Journal of Educational Psychology*, 99(3):561–574, 2007.

[5] Jon R. Star and Bethany Rittle-Johnson. It pays to compare: An experimental study on computational estimation. *Journal of Experimental Child Psychology*, 102(4):408 – 426, 2009.

[6] J.E. Hiebert. Conceptual and procedural knowledge: The case of mathematics. In *Colloquium series of the College of Education, University of Delaware.* Lawrence Erlbaum Associates, Inc, 1986.

[7] B. Rittle-Johnson, R.S. Siegler, and M.W. Alibali. Developing conceptual understanding and procedural skill in mathematics: An iterative process. *Journal of Educational Psychology*, 93(2):346, 2001.

[8] J. Kilpatrick, J. Swafford, and B. Findell. *Adding it up: Helping children learn mathematics.* National Academies Press, 2001.

[9] Edward A. Silver, Hala Ghousseini, Dana Gosen, Charalambos Charalambous, and Beatriz T. Font Strawhun. Moving from rhetoric to praxis: Issues faced by teachers in having students consider multiple solutions for problems in the mathematics classroom. *The Journal of Mathematical Behavior*, 24(3-4):287 – 301, 2005.

[10] J.L. Fraivillig, L.A. Murphy, and K.C. Fuson. Advancing children's mathematical thinking in everyday mathematics classrooms. *Journal for research in mathematics education*, pages 148–170, 1999.

[11] K. Hufferd-Ackles, K.C. Fuson, and M.G. Sherin. Describing levels and components of a math-talk learning community. *Journal for Research in Mathematics Education*, pages 81–116, 2004.

[12] K.J. Kurtz, C.H. Miao, and D. Gentner. Learning by analogical bootstrapping. *The Journal of the Learning Sciences*, 10(4):417–446, 2001.

[13] Daniel L. Schwartz and John D. Bransford. A time for telling. *Cognition and Instruction*, 16(4):475–5223, 1998.

[14] Lauren E. Margulieux, Mark Guzdial, and Richard Catrambone. Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proceedings of the ninth annual international conference on International computing education research*, ICER '12, pages 71–78, New York, NY, USA, 2012. ACM.

[15] Scott Oser. Blind analysis, or, 'The answer's not in the back of the book', 2008.

[16] University of North Texas Information Research and Analysis Lab. Hierarchical linear models (HLM) using R package nlme, 2012.

[17] Mike W.L. Cheung. Workshop in hierarchical linear modeling (HLM), 2009.

[18] J.R. Rausch, S.E. Maxwell, and K. Kelley. Analytic methods for questions pertaining to a randomized pretest, posttest, follow-up design. *Journal of Clinical Child and Adolescent Psychology*, 32(3):467–486, 2003.

[19] W.L. Cull. Untangling the benefits of multiple study opportunities and repeated testing for cued recall. *Applied Cognitive Psychology*, 14(3):215–235, 2000.

[20] A.J. Baroody and A. Dowker. *The development of arithmetic concepts and skills: Constructive adaptive expertise.* Lawrence Erlbaum, 2003.

[21] I. Kreft, J. De Leeuw, and J. de Leeuw. *Introducing multilevel modeling.* Sage London, 1998.

[22] Jean-Philippe Laurenceau. Introduction to hierarchical linear modeling, 2009.