Using a Computational Context to Investigate Student Reasoning About Whether "Order Matters" in Counting Problems

Elise Lockwood
Oregon State University

Abstract. Students often struggle with issues of order – that is, with distinguishing between permutations and combinations – when solving counting problems. There is a need to explore potential interventions to help students conceptually understand whether "order matters" and to differentiate meaningfully between these operations. In this paper, I investigate students' understanding of the issue of order in the context of Python computer programming. I show that some of the program commands seemed to reinforce important conceptual understandings of permutations and combinations and issues of order. I suggest that this is one example of a way in which a computational setting may facilitate mathematical learning.

Key words: Combinatorics, Computational Thinking, Permutations, Combinations

Introduction and Motivation

Determining whether or not "order matters" in a counting problem is a perennial issue in combinatorics, and students often struggle with whether to use a formula involving permutations or combinations when they approach counting problems (if either formula is appropriate). In this paper, I report on a study in which students were given opportunities to engage in computational activity (in the form of elementary programming tasks) as they solved combinatorial problems. In this report, I elaborate episodes that demonstrate the ways in which computational activity may have served to advance students' mathematical thinking. Specifically, I focus on the particular case of students engaging in computer programming to reason about permutations, combinations, and differences between these two fundamental operations.

I seek to accomplish two goals in this paper. First, I want to highlight a potential pedagogical innovation that sheds light on our understanding of how students might reason about an important combinatorial idea in a meaningful way (namely, the difference between combinations and permutations). That is, I am interested in the combinatorial goal of identifying an activity, which involves computing, that might help students understand this important combinatorial distinction. Second, I want to provide an example of what computational thinking and activity might look like in a mathematical context. In this way I want to contribute to the conversation of how computing might be leveraged to help students to reason about mathematical concepts. I seek to answer the following research question: In what ways did programming commands help students to reason about whether or not "order matters" in a counting problem?

Literature Review and Mathematical Perspective

Literature on combinatorics. Permutations and combinations are two foundational combinatorial ideas, and they form the basis of much of the counting that students do. The key difference is that permutations count arrangements of objects – that is, differently ordered arrangements of elements of a set are counted as distinct outcomes. When counting combinations, on the other hand, differently ordered arrangements of elements are *not* counted as distinct outcomes. For example, suppose we have the set S = {1, 2, 3, 4}, and we wanted to count permutations (and combinations) of 3 of the elements of S. There are 24 such permutations

(Figure 1), and there are only 4 such combinations: 123, 124, 134, 234. An in-depth discussion of the formulas for permutations and combinations is beyond the scope of this paper.

123, 124, 132, 134, 142, 143
213, 214, 231, 234, 241, 243
312, 313, 321, 324, 341, 342
412, 413, 421, 423, 431, 432

Figure 1 – Permutations of three of the numbers 1, 2, 3, and 4

There is ample evidence that students struggle to learn and distinguish between these two ideas (e.g., Annin & Lai, 2010; CadwalladerOsker, Engelke, Annin & Henning, 2012). In particular, many researchers have cited that a common error and struggle for students is to determine when to use a combination formula or a permutation formula. Batanero, et al., (1997) cite "errors of order" as being one of the primary errors that students encounter, and Annin and Lai (2010) discuss difficulties that students have maneuvering issues of order in counting. Lockwood (2014) previously showed examples of students not being sure of how to differentiate between when "order matters" and when it does not. Lockwood reports that when solving a counting problem, an undergraduate student, Kristin, said "I'm doing the combination ones because I'm pretty sure order doesn't matter with combination" (Lockwood, 2014, p. 33). When asked why, Kristin said, "I'm not sure about that one (laughs). I just kind of go off my gut for it, on the ones that don't specifically say order matters or it doesn't matter" (p. 33). This response is perhaps indicative of students' approaches to the distinction between permutations and combinations – often they do not have well-understood ways to differentiate between the two. Some have reported on ways to try to address this. Lockwood (2013, 2014) contends that by focusing on the set of outcomes, students can reason about the nature of outcomes as a way to clarify what is being counted, thus helping to determine whether or not counting matters.

Literature on computational thinking and activity. Mathematics departments across the country increasingly emphasize the importance of computation. As evidence for this trend, consider a) departments that include "computational requirements" for their mathematics majors, b) the growth of the branch of computational mathematics, and c) the myriad applications of computational mathematics, ranging from work with big data to modeling real-world problems using sophisticated software. Science, Technology, Engineering, and Mathematics (STEM) education researchers have focused on computation in the last decade especially, and computer scientist Wing (2006, 2008) coined the term *computational thinking* as analytical thinking that "takes an approach to solving problems, designing systems, and understanding human behavior that draws on concepts fundamental to computing" (Wing, 2008, p. 3717). In addition, the Next Generation Sciences Standards (NGSS Lead States, 2013) includes "using mathematics and computational thinking" (p. 37) as one of eight key scientific practices. I currently adopt the following definition of computational thinking, adapted from Wing (2014): Computational thinking is the way of thinking that one uses to formulate a problem and/or express its solution(s) in such a way that a computer (human or machine) could effectively carry it out.

Weintrop, et al. (2016) developed a "taxonomy of practices focusing on the application of computational thinking to mathematics and science" (p. 128). I use this taxonomy of practices, especially the computational activities associated with Computational Problem Solving Practices, to characterize computational activity. These include preparing problems for computational solutions, programming, choosing effective computational tools, assessing different approaches/solutions to a problem, creating computational abstractions, and troubleshooting and debugging (p. 135). Practically, for the results described in this paper, the students engaged in basic programming tasks in Python, and this primarily included preparing problems for computational solutions, programming, and troubleshooting and debugging.

Theoretical Perspectives

Characterizing combinatorial thinking and activity. In considering students' combinatorial thinking, I use Lockwood's (2013) model, which frames students' combinatorial thinking in terms of three key components: *Formulas/Expressions*, *Counting Processes*, and *Sets of Outcomes*. *Formulas/Expressions* are mathematical expressions that yield some numerical value. A formula or expression is what a student may write as "the answer" to a counting problem. *Counting Processes* are the imagined or actual enumeration processes in which a student engages – that is, the steps or procedures that one completes when solving a counting problem. *Sets of Outcomes* are the sets of elements that are being counted. The cardinality of the set of outcomes typically determines the answer to the problem.

Reinforcing the relationship between counting processes and sets of outcomes. The relationship between counting processes and sets of outcomes is particularly important for students to develop. In terms of the model, one way to frame students' difficulties with counting is that students do not clearly connect their counting processes with the outcomes they are trying to enumerate (Lockwood, et al., 2015). Thus, a possible solution to improve students' combinatorial problem solving is "to reinforce the relationship between counting processes and sets of outcomes, and to help students integrate the set of outcomes as a fundamental aspect of their combinatorial thinking and activity" (Lockwood, 2014; p. 36). One way to establish and strengthen this relationship is through the systematic listing of outcomes.

Lockwood and Gibson (2016) showed that listing behavior (taken as partial and complete listing) was positively correlated with correctly answering combinatorial problems for novice counters. Lockwood and Gibson hypothesized potential reasons for this correlation, namely that in terms of the model, listing supports the relationship between counting processes and sets of outcomes. This prior work suggests that the activity of listing has the potential to strengthen the important relationship between counting processes and sets of outcomes, and thus serve as an avenue by which students can solve combinatorial problems more successfully.

Computational activities represent a natural extension of listing. Even though prior work has demonstrated that listing is a potentially valuable combinatorial practice (Lockwood & Gibson, 2016), solutions to combinatorial problems can be enormous (there are ten billion 10-digit PIN numbers, for example). It is often not feasible for students to generate complete lists of outcomes by hand. Partial listing also has limitations, as patterns do not always extend to all cases, and students often fail to detect subtle errors. Thus, there is a dilemma – we know that listing can be valuable, but listing by hand has clear drawbacks. This leads to a question of how we can move past limitations of by-hand listing in order to facilitate listing in more complex problems and contexts. Fortunately, there is a natural solution to this question: we can leverage technology and computational activities, allowing students to reap similar benefits of by-hand listing by designing algorithms and computer programs to enumerate lists. I hypothesize that such activity can potentially strengthen the relationship between counting processes and sets of outcomes, which can help students solve counting problems. As noted in the Literature Review, I adopt Weintrop, et al.'s (2016) taxonomy in defining computational activity. I particular focus on programing, trouble shooting, and debugging as the primary computational activities.

Methods

Participants and Data Collection. In this paper I report on data from a teaching experiment (Steffe & Thompson, 2000) that consisted of 15 hours of contact time with two students in 60-90 minutes sessions. The participants I discuss in this paper were two vector calculus students who were interviewed as a pair (pseudonyms Charlotte and Diana). Both were novice counters and had no programming experience in high school or in college, and they were chosen based on 30-minute selection interviews. They were paired together because they had relatively similar backgrounds and abilities, and they also had schedules that allowed them to meet together for 15 hours over the term. Charlotte was a sophomore and Diana was a freshman at the time of the interviews, and both students were majoring in chemistry with an interest in forensic science.

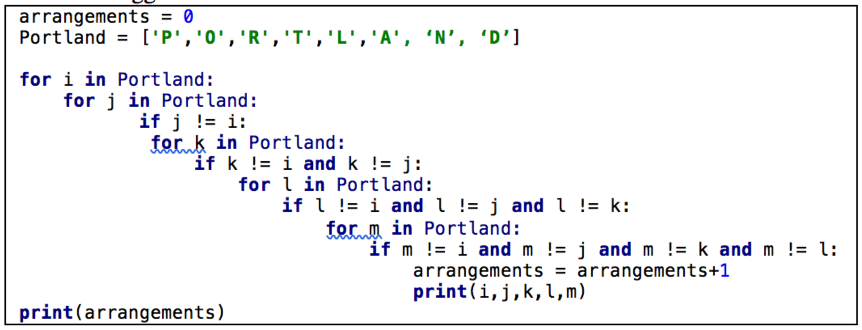
During the TEs, the students sat together and worked at a desktop computer in the programming environment PyCharm. I gave them paper handouts and also wrote the tasks and prompts in PyCharm, and the students used PyCharm to edit and run the Python code. To capture the interviews, I videotaped and audiotaped the interviews, and I also took a screen video recording of their work on the computer. This allowed me to view the students' on-paper work and their interactions, as well as what they programmed and how they used the computer.

Tasks. Over the course of the TE, I gave the students a variety of tasks in which they were asked to use the computer to determine the answers to counting problems. I created these tasks with the goal of targeting some fundamental combinatorial ideas, particularly focusing on the relationship between counting processes and sets of outcomes. The tasks overall followed a trajectory toward helping students reason about key combinatorial ideas including the multiplication principle, basic operations of permutations and combinations, and aspects of positional reasoning and encoding outcomes. For example, the tasks in Figures 2 and 3 represents typical tasks in the TE. Generally, I had them engage in programming directly by writing and running code, or I had them evaluate excerpts or outputs of code. I frequently asked for follow up questions or asked them to reflect on their thinking and activity. In this way, the interviews were interactive.

For the purposes of this paper, I focus especially on the tasks involving the development of permutations and combinations. In developing such tasks, I had considered some ways in which these ideas of permutations and combinations might be coded using Python. In particular, the task in Figure 2 shows how the symbol != helps to count permutations of 5 of the letters in the word PORTLAND. Note that != means "not equal to," and the if statements within the for loops indicate that the outcomes will not be printed if any of the characters are equal to previous characters. In this way, the inclusion of != in this code counts permutations in which repetition of characters is not allowed.

In a similar way, the task in Figure 3 shows how the symbol ">" might function in Python. By encoding the elements we want to count (books, in this case) as numbers, we can compare the elements using the greater than symbol. Thus, the "if $j > i$" condition will only consider arrangements for which a subsequent character is strictly greater than previous characters. Essentially, this would count something like 1, 2, 3, but it would not count 1, 3, 2 or 2, 1, 3, or any other arrangement of the numbers 1, 2, and 3. This is exactly what we want to count with combinations – subsets, but not arrangements, of some elements. The students were able to make sense of what the commands might mean and might do in terms of outcomes. As we will see in the results, the act of programming these ideas seems to have beceme meaningful and useful for them.

a) Can you write some code in order to create a list of all of the ways to arrange 3 of the letters in the word ROCKET?
b) Caleb had to answer the question: *How many arrangements are there of 5 of the letters in the word PORTLAND?* He wrote the following code to get the answer. Do you think he's correct? Why or why not? What numerical expression does his code suggest?

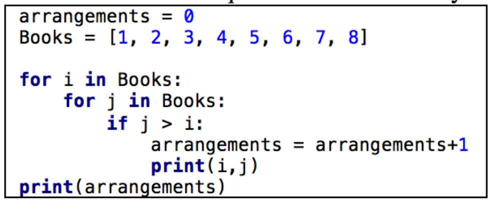


```
arrangements = 0
Portland = ['P','O','R','T','L','A', ‘N’, ‘D’]

for i in Portland:
    for j in Portland:
        if j != i:
         for k in Portland:
            if k != i and k != j:
                for l in Portland:
                    if l != i and l != j and l != k:
                        for m in Portland:
                            if m != i and m != j and m != k and m != l:
                                arrangements = arrangements+1
                                print(i,j,k,l,m)
print(arrangements)
```

Figure 2 – A task to elicit permutations

a) Suppose you have 8 books and you want to take a pair of them with you on vacation. How many ways are there to do this?
b) Consider the code below. Does this answer the same question as before? Why or why not?



```
arrangements = 0
Books = [1, 2, 3, 4, 5, 6, 7, 8]

for i in Books:
    for j in Books:
        if j > i:
            arrangements = arrangements+1
            print(i,j)
print(arrangements)
```

Figure 3 – A task to elicit combinations

Data Analysis. For the results shared in this paper, I reviewed transcripts, particularly episodes in which the students used, referred to, or reflected upon the "not equal to" or "greater than" symbols in their code. This allowed me to analyze the students' reasoning about these symbols, and I sought to understand and create a narrative (Auerbach & Silverstein, 2003) about their reasoning about and use of those symbols.

Results

In this section I describe the students' reasoning about the "not equal to" and "greater than" symbols as ways to express certain combinatorial constraints. In having to communicate with the computer via Python code, the students had to think about how the computer interpreted these different symbols and what the resulting output of the code would be. I will make the case that this experience helped the students make a meaningful distinction between these symbols and could clearly make a connection between these two ideas and what they did in terms of the outcomes. The students established meanings of these symbols as commands they gave the computer, and that this experience helped them to understand important aspects of counting.

The students first thought about the not equals to symbol (!=) in a problem in which they had to think about code that listed the number of ways to list arrangements of 5 people. In the excerpt below, we see the students initially interpreting and considering the != notation.

Int.: What do you this the code's doing.
Charlotte: Gosh a lot of code.

Diana: I think for sure that the statements have the exclamation point each time, that's making it so that these values will not repeat, which makes sense when you have five people because you can't just repeat a person.

Charlotte: Yeah, that makes sense. Yeah, kind of what she was saying, I think the code, yeah, just trying to figure out how many different arrangements each person can be in and then yeah, <u>each of these exclamation points, like Diana said, is to make sure John isn't sitting in two different seats at the same time</u>.

As we see in the underlined portion, the students were beginning to understand what the != symbol might be doing in terms of the context of the problem – namely, not allow for John to sit in two different seats at the same time.

Later in the teaching experiment the students were working on the Lollipop problem, which says, "How many ways are there to distribute 3 identical lollipops to 8 children?" The students had written code in which they used a greater than sign. Here they had established that they wanted to count sets of 3 numbers from the numbers 1 to 8, which would represent which children get lollipops. They noted that they did not want to count arrangements of these numbers because the lollipops are identical. In the excerpt below, we see Diana articulate the important fact that the use of "greater than" eliminates duplicates, in the sense of not allowing for both outcomes of 1, 2, 3 and 2, 3, 1 to be counted.

Charlotte: Because, yeah, then it eliminates the factor of duplicates.

Int.: Okay. And can you say again, how that 'greater than' sign eliminates the duplicates like you said?

Diana: So, like it says that *k* is not able to be less than *j*, it always has to be greater than. So, and in the example of the 1, 2, 3, it'll print 1, 2, 3, but then when it comes to printing 2, 3, 1, it won't be able to do it because *k* can't be 1 when these two are 2 and 3.

I suggest that, in these examples, the students were engaging in computational thinking. Diana's comments above suggest that she was considering what the computer would output, which suggests that she was thinking about what steps and procedures the computer was engaging in as it completed the program. In this way, the students seemed to be reasoning about the solution in such a way that they were considering what the computer must have done to carry it out.

Throughout the remainder of the interviews, the students continued to make this distinction and to use it in reasoning about problems. While I do not have space to detail each of these occurrences, I conclude these results with a wrap up discussion from the final session. We had explicitly asked the students some reflection questions about their coding and how they thought about certain aspects of their code. In the excerpt below, we see the students responding to a prompt that asked them to reflect on the difference between the > and != symbols.

Charlotte: Okay. So, the greater than symbol definitely plays an important role. In this problem with the alphabet, the greater than symbol played a role because you didn't wanna have A, E, I, O, and U not in alphabetical order. So, it helps arrange them in that order because A representing one, E representing 2, I representing 3, O for 4, and U for 5. You don't wanna have 3, 4, 5, 2, 1. So, that greater than symbol helps play a role for that. Do you wanna explain the not equal to?

Diana: Sure. So, the not equal to sign helps prevent the outcomes from being 1, 1, 1, 2, 2, 2.

> And in the case of the lollipop and the red balloon problem, you don't want one kid, which would be 1, 1, 1, getting all three lollipops. So, you use the not equal to statement.

In sum, while the students referred to particular problems in discussing the utility of each command, I contend that they were establishing ways of reasoning about these commands and issues of order in solving counting problems. They could clearly articulate the different commands and what they counted in terms of the sets of outcomes.

Discussion, Conclusion, and Implications

In this paper, I offered evidence of ways in which students reasoned about commands in Python in order to think about whether order should matter in solving counting problems. The students did eventually come to understand more general formulas for permutations and combinations, although they did not necessarily refer to them by those names. The point is that the students seemed to have established meaningful ways of thinking about generating outcomes through a program, and the symbols in the commands put certain constraints on what outcomes were being generated. In this way, the students were formulating a relationship between the counting process (the programs that involved nested for loops) and the outcomes that were being generated. By specifying that *i* != *j* or $j > i$, the students were imposing constraints that dictated the nature of the outcomes. I contend that the computing environment in particular leveraged this kind of activity and reasoning about these important combinatorial ideas.

There are obviously many different productive ways that students can reason about counting processes and outcomes. I am not claiming here that this is a superior way for students to reason, nor that it is the only way that they should reason about these ideas. But, the students seemed to demonstrate a solid and meaningful understanding of these ideas. Their understanding of what the greater than sign indicated in terms of duplicates stands in contrast to Lockwood's (2014) student who said she just went "off her gut." I certainly do not want to simply have mantras of "< means order doesn't matter" or "!= means order does matter", but I do not this was how the students were reasoning. Rather, it seems that by actually thinking carefully about what the program was doing in terms of those symbols, and thinking about both what those commands told the computer and how the computer implemented and carried them out, the students developed a better understanding of how the outcomes were being generated.

These findings provide an existence proof that meaningful mathematical ideas can be introduced and reinforced in computational settings. This suggests that there is more to study and learn related to the relationship between computational activity like programming in students' mathematical reasoning and activity.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1650943.

References

Annin, S. A., & Lai, K. S. (2010). Common errors in counting problems. *Mathematics Teacher, 103*(6), 402-409.

Auerbach, C. & Silverstein, L. B. (2003). *Qualitative data: An introduction to coding and analysis*. New York: New York University Press.

Batanero, C., Navarro-Pelayo, V., & Godino, J. (1997). Effect of the implicit combinatorial model on combinatorial reasoning in secondary school pupils. *Educational Studies in Mathematics, 32*, 181-199.

CadwalladerOlsker, T., Engelke, N., Annin, S., & Henning, A. (2012). Does a statement of whether order matters in counting problems affect students' strategies? In the *Electronic Proceedings of the 15th Annual Meeting of the Research on Undergraduate Mathematics Education*. Portland, OR: Portland State University.

Lockwood, E. (2013). A model of students' combinatorial thinking. *Journal of Mathematical Behavior, 32,* 251-265. DOI: 10.1016/j.jmathb.2013.02.008.

Lockwood, E. (2014). A set-oriented perspective on solving counting problems. *For the Learning of Mathematics, 34*(2), 31-37.

Lockwood, E., & Gibson, B. (2016). Combinatorial tasks and outcome listing: Examining productive listing among undergraduate students. *Educational Studies in Mathematics, 91*(2), 247-270. DOI: 10.1007/s10649-015-9664-5.

NGSS Lead States. (2013). *Next Generation Science Standards: For States, By State*s. Washington, DC: The National Academies Press.

Steffe, L. P., & Thompson, P. W. (2000). Teaching experiment methodology: Underlying principles and essential elements. In R. Lesh & A. E. Kelly (Eds.), *Research design in mathematics and science education* (pp. 267-307). Mahwah, NJ: Lawrence Erlbaum Associates.

Weintrop, D., Beheshti, E., Horn, M., Orton, K. Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science and Education Technology, 25*, 127-147. Doi: 10.1007/s10956-015-0581-5.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM. 49*(3)*,* 33-35.

Wing. J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A, 366,* 3717-3725.

Wing, J. M. (2014). Computational thinking benefits society. Retrieved from http://socialissues.cs.toronto.edu/index.html%3Fp=279.html.