Journal of Information Technology Education: Innovations in Practice Volume 14, 2015

Cite as: Crabtree, J., & Zhang, X. (2015). Recognizing and managing complexity: Teaching advanced programming concepts and techniques using the Zebra Puzzle. *Journal of Information Technology Education: Innovations in Practice, 14,* 171-189. Retrieved from http://www.jite.org/documents/Vol14/JITEv14IIPp171-189Crabtree1771.pdf

Recognizing and Managing Complexity: Teaching Advanced Programming Concepts and Techniques Using the Zebra Puzzle

John Crabtree and Xihui Zhang
Department of Computer Science & Information Systems
University of North Alabama, Florence, AL, USA

jcrabtree@una.edu xzhang6@una.edu

Abstract

Teaching advanced programming can be a challenge, especially when the students are pursuing different majors with diverse analytical and problem-solving capabilities. The purpose of this paper is to explore the efficacy of using a particular problem as a vehicle for imparting a broad set of programming concepts and problem-solving techniques. We present a classic brain teaser that is used to communicate and demonstrate advanced software development concepts and techniques. Our results show that students with varied academic experiences and goals, assuming at least one procedural/structured programming pre-requisite, can benefit from and also be challenged by such an exercise. Although this problem has been used by others in the classroom, we believe that our use of this problem in imparting such a broad range of topics to a diverse student population is unique.

Keywords: Information technology education, object-oriented programming, software engineering, design techniques, algorithm complexity.

Introduction

An important part of designing and developing quality software systems is learning how to recognize and manage complexity. James Gleick, the author of *Chaos: Making a New Science* and *The Information: A History, a Theory, a Flood*, has stated that:

> *"Computer programs are the most intricate, delicately balanced and finely interwoven of all the products of human industry to date. They are machines with far more moving parts than any engine: the parts don't wear out, but they interact and rub up against one another in ways the programmers themselves cannot predict."* (Gleick, 2002, p. 19)

Complexity is an inherent, essential feature of software (Brooks, 1987). Complexity comes from many different sources including the management of the development process, new programming languages, new development platforms, new information technology, as well as the problem domain itself (Booch et al., 2007).

Material published as part of this publication, either on-line or in print, is copyrighted by the Informing Science Institute. Permission to make digital or paper copy of part or all of these works for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage AND that copies 1) bear this notice in full and 2) give the full citation on the first page. It is permissible to abstract these works so long as credit is given. To copy in all other cases or to republish or to post on a server or to redistribute to lists requires specific permission and payment of a fee. Contact Publisher@InformingScience.org to request redistribution permission.

Editor: Keith A. Willoughby
Submitted: March 9, 2015; Revised: May 14, May 25, 2015; Accepted: June 4, 2015

The people who develop and test these systems are ultimately the key cause of software failures (Meneely, Williams, Snipes, & Osborne, 2008). Research has shown that the skill and experience level of the people involved in the development and testing process is a fundamental factor affecting failures in object-oriented systems (Arisholm & Briand, 2006).

There are many skills that need to be acquired in order to make the transition from the novice to the advanced level and beyond. Advanced programmers are aware of the existence of many different programming languages, some of the more popular categories into which these languages are divided, and the important distinctions between the categories. Familiarity with data structures, algorithms, and computational complexity is important as well. While correctly solving a particular problem is critical, solving the problem efficiently can make the difference between a successful system and one that is too slow or too inefficient to be practical. In industry, as novice programmers gain experience in implementation tasks, they are often expected to perform the higher-level analysis and design work. Experience with important and recognized solutions to frequent problems (i.e., design patterns) is expected of the competent designer. Early exposure to the concept of design patterns can greatly benefit students who expect to work with information systems in their professional careers. Testing is key to information systems (IS) project success and unit testing is an important concept and skill in ensuring system quality. Many other language-specific concepts and techniques can be explored either directly or indirectly using the puzzle below.

In 1962, *Life International* magazine published the following puzzle (see Table 1, referred to as the Zebra Puzzle or the Puzzle henceforth) that asked “Who drinks water?” and “Who owns the zebra?” given the following set of rules that apply to five houses, painted different colors with inhabitants of different nationalities who “*own different pets, drink different beverages, and smoke different brands of American cigaretes [sic]*” (“A problem for the logical,” 1962).

Table 1: The Zebra Puzzle
1. There are five houses. 2. The Englishman lives in the red house. 3. The Spaniard owns the dog. 4. Coffee is drunk in the green house. 5. The Ukrainian drinks tea. 6. The green house is immediately to the right of the ivory house. 7. The Old Gold smoker owns snails. 8. Kools are smoked in the yellow house. 9. Milk is drunk in the middle house. 10. The Norwegian lives in the first house. 11. The man who smokes Chesterfields lives in the house next to the man with the fox. 12. Kools are smoked in the house next to the house where the horse is kept. 13. The Lucky Strike smoker drinks orange juice. 14. The Japanese smokes Parliaments. 15. The Norwegian lives next to the blue house.

This puzzle, which is said to have been invented by Albert Einstein as a boy and is also sometimes attributed to Lewis Carroll (http://en.wikipedia.org/wiki/Zebra_Puzzle), has several variants that, for example, replace cigarettes with occupations or modes of transportation (see Mohr, 2004).

Although this problem has been used in the classroom for many purposes (most frequently when used with declarative programming languages), we believe that our use of this problem to impart such a wide variety of topics is unique. Furthermore, we believe that this problem can be very

effective in communicating important subjects to students who may have dissimilar educational backgrounds and academic majors.

Analysis

The first step in analyzing the problem is to recognize the combinatorial patterns involved in generating possible solutions. Each variable in the problem can take on five different values. For example, the nationalities of the occupants can be represented as the set {E, J, N, S, U}, using only the first letter of each nationality. Table 2 shows the five different values for each of the five variables.

Table 2: Variables and Values in the Zebra Puzzle

VARIABLE	VALUES (in alphabetic order)				
Nationality	Englishman	Japanese	Norwegian	Spaniard	Ukrainian
House Color	Blue	Green	Ivory	Red	Yellow
Drink	Coffee	Milk	Orange Juice	Tea	Water
Cigarette Brand	Chesterfields	Kools	Lucky Strike	Old Gold	Parliaments
Pet	Dog	Fox	Horse	Snail	Zebra

If we recognize that each "house" may represent a positional repository for attributes, students should be able to reason that, given just two houses and two nationalities {E, J}, there are two possible permutations:

{E, J}, {J, E}

And given three houses and three nationalities {E, J, N}, there are six permutations:

{E, J, N }, { J, E, N }, {E, N, J }, { J, N, E}, { N, E, J }, { N, J, E}

Using induction, the fact that, 4! or 24 permutations of four nationalities exist should lead the students to the general form of n! permutations for each of the variables. Therefore, there are 120 possible combinations of the five nationalities (5! = 120). Likewise, there are 120 different ordered lists of five values (i.e., 120 5-tuples) for pets, 120 5-tuples for drinks, and so on.

Exhaustive Search

Assuming that we know how to generate all 120 permutations for each attribute (we will address this topic shortly), the problem can be reduced to an exercise in combining all of the 5-tuples to produce all possible positional combinations of the attributes. Nested loops can be used to generate each of these combinations (Knuth, 2011).

Algorithm 1: Mixed-radix Generation

For each nationality
    For each house color
        For each drink
            For each cigarette brand
                For each pet
                    Test the combination of attributes against the rules.

This algorithm should lead the students to the conclusion that there are $(5!)^5$ = 24,883,200,000 different combinations to be tested.

This provides an excellent opportunity to introduce a number of valuable concepts from computer science. Assuming that each loop iteration requires 70 nanoseconds of compute time, how long would this calculation take? What if each attribute could take on one of six different values instead of five?

This brute-force approach, using the Java code in Appendix A, requires about 29 minutes to execute using a machine with a 2.80 GHz Dual-Core CPU operating with 4 GB of RAM. Increasing the number of possible values in the puzzle from five to six for each attribute would increase the runtime as to put a single execution of the program beyond the time available in the typical semester, as $(6!)^5$ iterations would take approximately 157 days.

This can obviously lead to a discussion of algorithm complexity analysis and NP-Complete problems. Even students in computer-related programs that do not cover algorithmic analysis in depth, should be aware of the large class of common problems, like the *Traveling Salesman Problem* (http://en.wikipedia.org/wiki/Travelling_salesman_problem), for which no efficient algorithm exists.

Interfaces

Experienced object-oriented programmers know that they should code to an interface and not to an implementation (Gamma, Helm, Johnson, & Vlissides, 1994). This problem provides students the opportunity to learn how to properly design an abstract class or interface that hides the implementation details of generating all permutations of each attribute. It also provides an opportunity to learn how to integrate a well-documented, third-party library that is provided without source code.

The following algorithm, which has been traced back to 14th century India, should be simple to comprehend and implement for most students who have completed an introductory programming course (Knuth, 2011):

Algorithm 2: Lexicographic Permutation Generation
Given a sequence of *n* sorted elements, generate all permutations in lexicographic order. Repeat forever Add the current sequence to the list of permutations Starting at the right, find the first pair of adjacent elements where the left element is strictly less than the right element (call the position of the left element “x”) If there is no such element, the algorithm terminates Starting at the right, find the first element that is strictly greater than element x and swap this element with element x Swap all elements from x+1 to the end of the array (assuming x+1 is not the end): Starting with position x+1 (which we will rename y) and the last element (we will call this position z), swap the elements at y and z Increment y, decrement z and swap; repeat as long as y < z

The example provided by Knuth (2011) is {1, 2, 2, 3} which produces the following permutations by following the steps in Algorithm 2 above: 1223, 1232, 1322, 2123, 2132, 2213, 2231, 2312, 2321, 3122, 3212, 3221.

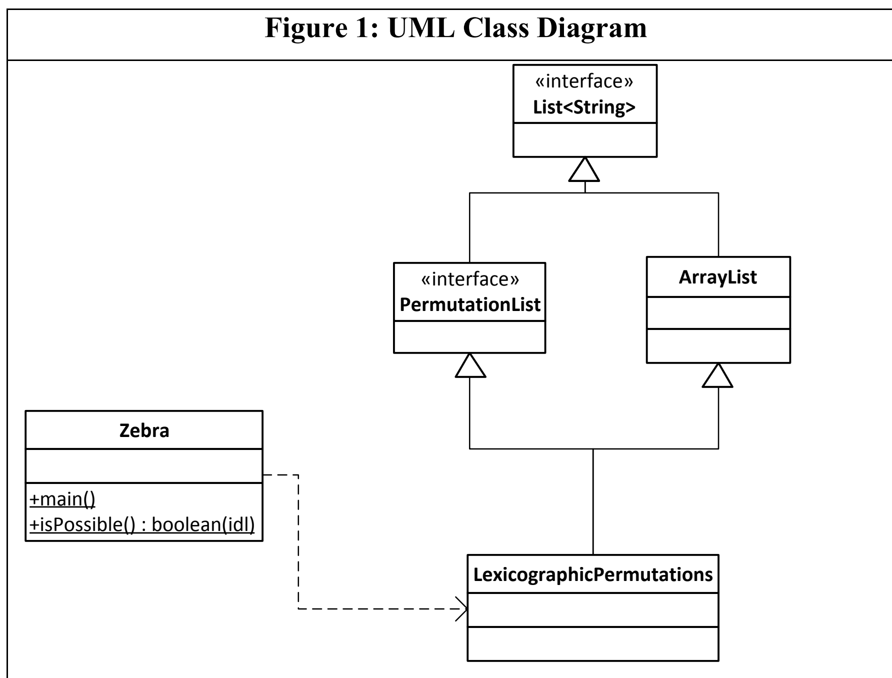
Advanced students could be challenged to create implementations that are optimized in some fashion (e.g., for efficiency). The following is an example of a recursive implementation in Java:

Listing 1: Recursive Permutation Generation in Java

```
private static void genPerms(String left, String right, List<String> list) {
        if(right.length() < 2) {
                if (!list.contains(left + right)) {
                        list.add(left + right);
                }
        }
        else {
                for(int i = 0; i < right.length(); i++) {
                        genPerms(left + right.charAt(i), right.substring(0, i) +
                                right.substring(i + 1), list);
                }
        }
}
```

The UML class diagram in Figure 1demonstrates the design of the Java code included in the appendices.

Figure 1: UML Class Diagram



This design can be used to prompt discussion of multiple inheritance, the differences in C++ and Java (which has traditionally not allowed multiple inheritance of implementation), the use of parameterized classes, the proper use of collections, and related design issues. The instructor could also use this as an opportunity to explore *lambda* expressions in Java 8 which now provide support for the multiple inheritance of implementation.

In a team environment with students of widely variant ability levels (e.g., a web design course with both programmers and non-programmers), students with better technical skills could implement the permutation code that could be shared with classmates. This would illustrate and enforce the "code to an interface" principle. The Java code from Listing 1 could be used to develop another class that implements the PermutationList interface and extends ArrayList that could be used in place of LexicographicPermutations in the main method of Zebra. The source code for

PermutationList and LexicographicPermutations can be found in Appendices B and C, respectively.

Test-driven development could also be taught by having students develop test cases *first* that ensure, for example, that all possible permutations are generated. Students would then make sure that the test cases fail (since no implementation yet exists). Finally, they would implement the code to allow the test cases to run correctly (Beck, 2002).

Prolog

This puzzle is a perfect candidate for a constraint-based programming language like Prolog. Even if learning Prolog is beyond the scope of the course, as it was in our case, the nature of this problem can be used to spark a discussion of the different classes of programming languages.

Most students today are exposed to structured, object-oriented languages (e.g., Java, C++, C#, and Visual Basic .NET) which are considered to be imperative languages in which the programmer specifies the tasks that the machine is instructed to carry out. Declarative languages, on the other hand, rely on an underlying set of processing assumptions (i.e., an algorithm) in order to formulate a result. The classic example of declarative programming is SQL. Prolog is another interesting declarative language that could be demonstrated by the instructor solving the problem in the classroom using the following code (our approach), requiring the students to write their own Prolog code to solve the Zebra problem, or some variation between these two options.

The following code generates a solution to the puzzle using GNU Prolog:

Listing 2: Prolog Solution to the Zebra Puzzle

```
nationality(norwegian).
nationality(ukrainian).
nationality(englishman).
nationality(spaniard).
nationality(japanese).

color(yellow).
color(blue).
color(red).
color(ivory).
color(green).

drink(water).
drink(tea).
drink(milk).
drink(oj).
drink(coffee).

pet(fox).
pet(horse).
pet(snails).
pet(dog).
pet(zebra).

smokes(kools).
smokes(chesterfields).
smokes(old_gold).
smokes(lucky_strikes).
smokes(parliaments).
```

```
solve(Puzzle, Solution) :-
        Solution = Puzzle,
        Puzzle = [[N1, C1, D1, P1, S1],
                  [N2, C2, D2, P2, S2],
                  [N3, C3, D3, P3, S3],
                  [N4, C4, D4, P4, S4],
                  [N5, C5, D5, P5, S5]],

        nationality(N1), nationality(N2), nationality(N3), nationality(N4),
        nationality(N5),
        \+(N1 = N2), \+(N1 = N3), \+(N1 = N4), \+(N1 = N5),
        \+(N2 = N3), \+(N2 = N4), \+(N2 = N5),
        \+(N3 = N4), \+(N3 = N5),
        \+(N4 = N5),

        color(C1), color(C2), color(C3), color(C4), color(C5),
        \+(C1 = C2), \+(C1 = C3), \+(C1 = C4), \+(C1 = C5),
        \+(C2 = C3), \+(C2 = C4), \+(C2 = C5),
        \+(C3 = C4), \+(C3 = C5),
        \+(C4 = C5),

        drink(D1), drink(D2), drink(D3), drink(D4), drink(D5),
        \+(D1 = D2), \+(D1 = D3), \+(D1 = D4), \+(D1 = D5),
        \+(D2 = D3), \+(D2 = D4), \+(D2 = D5),
        \+(D3 = D4), \+(D3 = D5),
        \+(D4 = D5),

        pet(P1), pet(P2), pet(P3), pet(P4), pet(P5),
        \+(P1 = P2), \+(P1 = P3), \+(P1 = P4), \+(P1 = P5),
        \+(P2 = P3), \+(P2 = P4), \+(P2 = P5),
        \+(P3 = P4), \+(P3 = P5),
        \+(P4 = P5),

        smokes(S1), smokes(S2), smokes(S3), smokes(S4), smokes(S5),
        \+(S1 = S2), \+(S1 = S3), \+(S1 = S4), \+(S1 = S5),
        \+(S2 = S3), \+(S2 = S4), \+(S2 = S5),
        \+(S3 = S4), \+(S3 = S5),
        \+(S4 = S5),

        member([englishman,red,_,_,_], Puzzle),
        member([spaniard,_,_,dog,_], Puzzle),
        member([_,green,coffee,_,_], Puzzle),
        member([ukrainian,_,tea,_,_], Puzzle),
        % The green house is immediately to the right of the ivory house.
        member([_,_,_,snails,old_gold], Puzzle),
        member([_,yellow,_,_,kools], Puzzle),
        nth(3, Puzzle, [_,_,milk,_,_]),
        nth(1, Puzzle, [norwegian,_,_,_,_]),
        % The man who smokes Chesterfields lives in the house next to the man with the fox.
        % Kools are smoked in the house next to the house where the horse is kept.
        member([_,_,oj,_,lucky_strikes], Puzzle),
        member([japanese,_,_,_,parliaments], Puzzle),
        nth(2, Puzzle, [_,blue,_,_,_]).
```

If Prolog is not explicitly mentioned, students should at least be exposed to the notion that some problems lend themselves to more efficient solutions when the proper programming language is used.

The Assignments

This problem was assigned to students in three separate parts in an advanced programming course, CIS 315: Advanced Programming Using Java, during the fall semester of 2013. The course is required of all CIS majors in the Enterprise Information Systems option. Therefore, the majority of the students who registered for the class (22 out of 28) were CIS majors, followed by four computer science majors, one geographic information systems major and one entertainment industry major. Most of the students (21) were seniors, along with five juniors, one sophomore, and one graduate student. Three of the students were female. The average age of the class was 24 years of age. Eight students dropped the course at some point during the semester. Three of the five juniors and the only sophomore withdrew from the course before the end of the semester. It must be noted that this was an online course. We have found that, in general, younger students have a more difficult time completing online courses.

Assignment 1: Puzzle Analysis

Given the text of the 1962 *Life International* magazine puzzle, use only paper and pencil to attempt to solve this puzzle. Do not look it up on the Internet or use any kind of software. For full credit report your answers (if any) and the amount of time you spent on the problem.

Assignment 2: Puzzle Algorithm Design

The first step in analyzing the problem is to recognize the combinatorial patterns involved in generating possible solutions. Each variable in the problem can take on five different values. For example, the nationalities of the occupants can be represented as the set {E, S, U, N, J}, using only the first letter of each nationality.

If we recognize that each “house” may represent a positional repository for attributes, you should be able to recognize that, given just two houses and two nationalities {E, S}, there are two possible permutations:

{E, S}, {S, E}

And given three houses and three nationalities {E, S, U}, there are six permutations:

{E, S, U}, {S, E, U}, {E, U, S}, {S, U, E}, {U, E, S}, {U, S, E}

Students were then asked to provide answers to the following questions:

1. How many combinations of the five nationalities exist?
2. How many combinations of the five different pet options exist?
3. How many different combinations of all of the different problem variables exist? In other words, how many different possibilities must be evaluated in order to determine if there is at least one solution that satisfies all of the rules?
4. Create an algorithm (using pseudocode or a flowchart) that could be used to solve the puzzle.

Assignment 3: Puzzle Implementation

Students were given the three-page document displayed in Appendix D and asked to provide answers to the following questions:

1. Assuming that each iteration of the innermost loop requires 70 nanoseconds of compute time, how long would it take to test all possible solutions?
2. If the problem was modified so that each attribute could take on one of six different values instead of five, how long would it take to test all possible solutions?
3. Implement a Java program that will solve the puzzle using the algorithm above and the library provided.

Students were provided with a JavaARchive (JAR) file containing the compiled code that generated the permutations discussed in the Analysis section. They were also provided with the javadoc-generated HTML pages that provide the API-level documentation for the code within the JAR file.

Results

Assignment 1: Puzzle Analysis

Out of a total of 22 students in class, 20 students submitted an attempt to solve the puzzle using only pencil and paper. Of those, 17 solved the problem correctly, 2 submitted partially correct solutions, and 1 solved the problem incorrectly. The average amount of time spent on solving the puzzle was 1.95 hours; the median was 1.63 hours; the minimum was 15 minutes; the maximum was 5 hours. One student admitted to seeing the problem before and one admitted to being distracted by watching football while trying to solve the puzzle.

Assignment 2: Puzzle Algorithm Design

Of the 17 students who submitted the assignment only two answered the questions correctly and were able to produce an algorithm that would find a solution. One other student submitted an algorithm that displayed some understanding of the problem, but was too vague to be implemented. The other students either did not submit a workable algorithm or misunderstood the instructions. Five of the students did not derive the correct number of combinations.

Assignment 3: Puzzle Implementation

The first question enables the students to predict how long their programs will run given the worst-case scenario (i.e., all possible permutations must be investigated). It was hoped that the answer (29 minutes) would encourage students to investigate ways to minimize the problem domain.

The second question was designed to illustrate how the search space could grow too large (given the time constraints of a typical semester), by simply adding one more possible value to each variable.

Of the 18 students who submitted the assignment only one answered both questions correctly and submitted an efficient program that found the solution quickly. Two others produced efficient code but did not calculate the correct answers to the questions. Eleven more solved the problem but did not minimize the search space. Two produced incorrect output and two more were unable to avoid runtime errors.

Eight predicted how long the worst-case program should run (i.e., about 29 minutes, given 70 ns per iteration) and of those only four correctly discovered that adding one more value in each category would require over 157 days to search the entire problem space (assuming a worst-case, brute-force approach).

Student Perceptions

To evaluate the students' perceptions of the "Puzzle" assignments, an online survey was created using services provided by SurveyMonkey.com. This ensures anonymity so that students would be more comfortable in sharing their thoughts and perceptions. The survey included two essay questions. The first question is: "Did you think that the 'Puzzle' assignments provided valuable learning experiences? Why or why not?" The second question is: "What were the most difficult challenges that you encountered while working on these assignments?" The survey link was distributed by course announcement through the school's Angel Learning System.

Each question received 15 responses**.** In response to question one, 80% (12 out of 15) of the respondents said that the "Puzzle" assignments did provide valuable learning experiences, mostly because it helped them to think through a problem logically and then solve it using a variety of programming skills. Some of the representative comments are as follows:

- "Yes. It does help to get to learn how to turn a true problem into a program."
- "I think the Puzzle assignments did provide experience for learning. Unlike most of the problems given to us in the books, the Puzzle assignments provided us with a real problem that we had to use Java code in order to solve. Instead of having cookie-cutter type questions provided in the book, the assignment allowed us to use multiple skills we had learned over the course of the semester in order to fully demonstrate our knowledge."
- "I think the assignments provided good experience because the first assignment helped to visualize the problem by solving it on paper. The second assignment gave me insight into a rough draft for the program. The third assignment gave me experience with taking a logical problem combined with mathematics and developing a program to solve it. I enjoyed working through the reasoning and seeing an example of how programming could be used in everyday situations."
- "Yes, it allows you to think through a problem logically and then solve it using programming skills."
- "Yes, because it demanded that you think in a very analytical way, which will really improve your problem-solving skills."

Three students didn't think the "Puzzle" assignments had provided valuable learning experiences: One thought they were too easy, and the other two thought they were very confusing and tough.

In response to question two, over 93% (14 out of 15) students felt that they were challenged while working on the "Puzzle" assignments. The top three most reported challenges include solving the puzzle with pencil and paper, translating ideas into code, and checking whether the code works correctly. Some of the representative comments are as follows:

- "It is difficult to find the right method to solve the problem. It is also very difficult to figure out whether the program works fine or not."
- "Encapsulating the given set of rules into coded logic; Testing whether the code works correctly or not per the given rules."
- "The most difficult challenge was working the original puzzle out by hand and figuring out the best way to do so."
- "The most difficult part of the puzzle program was trying to conceptually think through what the program was supposed to do."
- "Figuring out how to translate my ideas into code."

One student pointed out that lacking of clarity of the whole process had posed a big challenge, as he responded to the second question: "*One challenge I had with the Puzzle assignments was the lack of clarity as to the whole process. I wish instead of parsing the assignment out over the course of the second half of this semester that we had it all given to us at once so that we could*

understand the point of it all. It was confusing as to why we were solving puzzles with pen and paper for the first assignment. If all the cards had been laid out from the start, I believe the whole process would have [sic] more sense, and enhanced the learning experience." This is the biggest lesson learned. We will definitely try to improve the clarity of the whole process in the future.

Analysis of Results

A common theme discovered in the feedback was some difficulty understanding either the assignment, the problem, or the problem-solving process itself. An early lecture focused on metacognition and a decomposition of the problem-solving process (Morin, Thomas, & Saadé, 2014) may help address this issue. In addition, similar smaller problems earlier in the semester may help students understand this larger project.

Throughout the semester, these students are made aware of the various design and implementation options that exist. There is rarely one right way to solve a programming problem and software design is a very creative and enjoyable process that few students can truly appreciate in the classroom setting. We intend to broaden our approach with an emphasis on more design options and more opportunity for expressing creativity in the future.

Fewer than half of the students correctly calculated the estimated runtime of the algorithm provided. Again, similar smaller quantitative problems should allow students to gain confidence in making these kinds of quick calculations that can provide valuable insight into various design choices.

Conclusion

This article described the assignments generated from the classic Zebra Puzzle, which was used to teach students advanced programming concepts and techniques in order to better recognize and manage complexity in the software development process. After a brief review of the history of the puzzle, the problem was analyzed from several perspectives. Quite a few different design and implementation options were discussed and presented. We provided the specifications of the assignments as well as the report on student performance. We also collected survey data for evaluating student perceptions regarding the assignments. Results from the instructor's grading report and student perceptions demonstrated that the assignments were effective in teaching students advanced programming concepts and techniques.

Specifically, the Zebra Puzzle can be used as an effective tool to teach advanced programming students a number of important concepts and techniques including: algorithm analysis, performance analysis, object-oriented design, abstraction and encapsulation, combinatorial problems, NP-completeness, constraint-based programming, and test-driven development. Students performed well on the post-test, demonstrating a solid understanding of the concepts and techniques described above. Students with a variety of academic majors (e.g., computer science, information systems, geography, and entertainment) were able to work through a solution to the problem and in the process gain insight into these important concepts and techniques.

In the next iteration, we will attempt to improve the pedagogical effectiveness of the assignments by: (1) clarifying the design instructions; (2) focusing more class time on design issues; (3) introducing quantitative analysis of problems and programs earlier in the semester; and (4) including more assignments that involve quantitative analysis. In addition, we intend to explore the possibility of incorporating some features of a practicum experience into this project in order to prepare most of these students for the CIS capstone course for which this course is a pre-requisite (Mason, 2013).

References

A problem for the logical: Who owns the zebra? (1962, December 17). *Life International, 95*.

Arisholm, E., & Briand, L. C. (2006). Predicting fault-prone components in a java legacy system. *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering* (ISESE 2006, pp. 8-17). New York, NY, USA: ACM.

Beck, K. (2002). *Test-driven development: By example*. Boston, MA, USA: Addison-Wesley Professional.

Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., & Houston, K. A. (2007). *Object-oriented analysis and design with applications* (3rd ed.). Boston, MA, USA: Addison-Wesley Professional.

Brooks, F. P., Jr. (1987). No silver bullet: Essence and accidents of software engineering. *Computer, 20*(4), 10-19.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Professional.

Gleick, J. (2002). *What just happened: A chronicle from the information frontier*. New York, NY, USA: Pantheon Books.

Knuth, D. E. (2011). *The art of computer programming - Volume 4A: Combinatorial algorithms* (1st ed.). Boston, MA, USA: Addison-Wesley Professional.

Mason, R. T. (2013). A database practicum for teaching database administration and software development at Regis University. *Journal of Information Technology Education: Innovations in Practice, 12*, 159-168. Retrieved from http://www.jite.org/documents/Vol12/JITEv12IIPp159-168MasonFT117.pdf

Meneely, A., Williams, L., Snipes, W., & Osborne, J. (2008). Predicting failures with developer networks and social network analysis. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (SIGSOFT 2008/FSE-16, pp. 13-23). New York, NY, USA: ACM.

Mohr, J. (2004). *The zebra puzzle*. Retrieved from http://www.augustana.ab.ca/~mohrj/courses/common/csc370/lecture_notes/prolog_examples/zebra_puzzle.html

Morin, D., Thomas, J. D. E., & Saadé, R. G. (2014). Problem-solving and web-based learning. *Proceedings of the e-Skills for Knowledge Production and Innovation Conference* (e-Skills 2014, pp. 243-253). Santa Rosa, CA, USA: Informing Science Institute.

Appendices

Appendix A

```
package logic;

/**
 * This program solves a logical puzzle first published in 1962 in Life International magazine.
 *
 * <ol>
 * <li>There are five houses.</li>
 * <li>The Englishman lives in the red house.</li>
 * <li>The Spaniard owns the dog.</li>
 * <li>Coffee is drunk in the green house.</li>
 * <li>The Ukrainian drinks tea.</li>
 * <li>The green house is immediately to the right of the ivory house.</li>
 * <li>The Old Gold smoker owns snails.</li>
 * <li>Kools are smoked in the yellow house.</li>
 * <li>Milk is drunk in the middle house.</li>
 * <li>The Norwegian lives in the first house.</li>
 * <li>The man who smokes Chesterfields lives in the house next to the man with
 * the fox.</li>
 * <li>Kools are smoked in the house next to the house where the horse is kept.</li>
 * <li>The Lucky Strike smoker drinks orange juice.</li>
 * <li>The Japanese smokes Parliaments.</li>
 * <li>The Norwegian lives next to the blue house.</li>
 * </ol>
 *
 * <p>
 * Now, who drinks water? Who owns the zebra?
 * </p>
 *
 * <p>
 * In the interest of clarity, it must be added that each of the five houses is
 * painted a different color, and their inhabitants are of different national
 * extractions, own different pets, drink different beverages and smoke
 * different brands of American cigarets [sic]. One other thing: in Statement 6,
 * right means your right.
 * </p>
 *
 * <p>
 * source: Life International, December 17, 1962
 * </p>
 *
 */

public class Zebra {

        private static boolean isPossible(String nationality, String house,
                        String drink, String brand, String pet) {
                // The Englishman lives in the red house.
                if (house.charAt(nationality.indexOf("E")) != 'r') {
                        return false;
                }
                // The Spaniard owns the dog.
                if (pet.charAt(nationality.indexOf("S")) != 'd') {
                        return false;
                }
                // Coffee is drunk in the green house.
                if (drink.charAt(house.indexOf("g")) != 'c') {
                        return false;
                }
                // The Ukrainian drinks tea.
                if (drink.charAt(nationality.indexOf("U")) != 't') {
                        return false;
                }

                // The green house is immediately to the right of the ivory house.
```

```
int ivory = house.indexOf("i");
if (ivory > 3) {
        return false;
}
if (house.charAt(ivory + 1) != 'g') {
        return false;
}

// The Old Gold smoker owns snails.
if (pet.charAt(brand.indexOf("o")) != 's') {
        return false;
}

// Kools are smoked in the yellow house.
if (brand.charAt(house.indexOf("y")) != 'k') {
        return false;
}

// Milk is drunk in the middle house.
if (drink.charAt(2) != 'm') {
        return false;
}

// The Norwegian lives in the first house.
int norwegian = nationality.indexOf("N");
if (norwegian != 0) {
        return false;
}

// The man who smokes Chesterfields lives in the house next to the
// man with the fox.
int chest = brand.indexOf("c");
if (chest == 0) {
        if (pet.charAt(chest + 1) != 'f') {
                return false;
        }
} else if (chest == 4) {
        if (pet.charAt(chest - 1) != 'f') {
                return false;
        }
} else {
        if (pet.charAt(chest + 1) != 'f' && pet.charAt(chest - 1) != 'f') {
                return false;
        }
}

// Kools are smoked in the house next to the house where the horse
// is kept.
int horse = pet.indexOf("h");
if (horse == 0) {
        if (brand.charAt(horse + 1) != 'k') {
                return false;
        }
} else if (horse == 4) {
        if (brand.charAt(horse - 1) != 'k') {
                return false;
        }
} else {
        if (brand.charAt(horse + 1) != 'k'
                        && brand.charAt(horse - 1) != 'k') {
                return false;
        }
}

// The Lucky Strike smoker drinks orange juice.
if (drink.charAt(brand.indexOf("l")) != 'o') {
        return false;
}

// The Japanese smokes Parliaments.
```

```
                if (nationality.charAt(brand.indexOf("p")) != 'J') {
                        return false;
                }

                // The Norwegian lives next to the blue house.
                if (norwegian == 0) {
                        if (house.charAt(norwegian + 1) != 'b') {
                                return false;
                        }
                } else if (norwegian == 4) {
                        if (house.charAt(norwegian - 1) != 'b') {
                                return false;
                        }
                } else {
                        if (house.charAt(norwegian + 1) != 'b'
                                        && house.charAt(norwegian - 1) != 'b') {
                                return false;
                        }
                }

                return true;
        }

        public static void main(String args[]) {

                int count = 0;
                int outerLoopCount = 0;

                // Generate all n! permutations of input strings
                PermutationList nations = new LexicographicPermutations("EJNSU");
                PermutationList houseColors = new LexicographicPermutations("bgiry");
                PermutationList drinks = new LexicographicPermutations("cmotw");
                PermutationList smokes = new LexicographicPermutations("cklop");
                PermutationList pets = new LexicographicPermutations("dfhsz");

                // Brute force search of all (5!)^5 = 24,883,200,000 combinations.
                for (String nation : nations) {
                        for (String house : houseColors) {
                                for (String drink : drinks) {
                                        for (String brand : smokes) {
                                                for (String pet : pets) {
                                                        if (isPossible(nation, house, drink, brand, pet)) {
                                                                System.out.println(nation);
                                                                System.out.println(house);
                                                                System.out.println(drink);
                                                                System.out.println(brand);
                                                                System.out.println(pet);
                                                                count++;
                                                        }
                                                }
                                        }
                                }
                        }
                        outerLoopCount++;
                        System.out.println("Finished " + outerLoopCount + " of 120");
                }
                System.out.println("Found " + count + " solution(s).");
        }
}
```

Appendix B

```
package logic;

import java.util.List;

public interface PermutationList extends List<String> {

}
```

Appendix C

```
package logic;

import java.util.ArrayList;

public class LexicographicPermutations extends ArrayList<String> implements PermutationList {

	/**
	 * Java code implemented by the authors (Crabtree and Zhang) based on
	 * Knuth's algorithm L from Section 7.2.1.2 Generating all permutations.
	 *
	 * NOTE: Knuth uses an auxiliary element (a0) which is not used in this implementation.
	 *
	 * Knuth, D. E. (2011). The Art of Computer Programming,
	 * Volume 4A: Combinatorial Algorithms, Part 1 (1st ed.).
	 * Addison-Wesley Professional.
	 */
	public LexicographicPermutations(String inputTokens) {
		super();

		do {
			// L1. [Visit.] Visit the permutation a1a2...an
			this.add(inputTokens);

			// L2. [Find j.]
			int j = inputTokens.length() - 1;
			while (inputTokens.charAt(j -1) >= inputTokens.charAt(j)) {
				if (j == 1) {
					return;
				}
				j -= 1;
			}

			// L3. [Increase aj.]
			int l = inputTokens.length();
			while (inputTokens.charAt(j - 1) >= inputTokens.charAt(l -1)) {
				l -= 1;
			}
			inputTokens = swap(inputTokens, j - 1, l - 1);

			// L4. Reverse aj+1...an.]
			int k = j + 1;
			l = inputTokens.length();
			while (k < l) {
				inputTokens = swap(inputTokens, k - 1, l - 1);
				k += 1;
				l -= 1;
			}
		} while (true);
	}

	private String swap(String in, int a, int b) {
		StringBuffer temp = new StringBuffer();
		for (int i = 0; i < in.length(); i++) {
			if (i == a) {
				temp.append(in.charAt(b));
			} else if (i == b) {
				temp.append(in.charAt(a));
			} else {
				temp.append(in.charAt(i));
			}
		}

		return temp.toString();
	}
}
```

Appendix D

The Problem

In 1962, Life International magazine published the following puzzle that asked "who drinks water?" and "who owns the zebra?" given the following set of rules:

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.

Analysis

The first step in analyzing the problem is to recognize the combinatorial patterns involved in generating possible solutions. Each variable in the problem can take on five different values. For example, the nationalities of the occupants can be represented as the set {E, S, U, N, J}, using only the first letter of each nationality.

If we recognize that each "house" may represent a positional repository for attributes, you should be able to reason that, given just two houses and two nationalities {E, S}, there are two possible permutations:

{E, S}, {S, E}

And given three houses and nationalities there are six permutations:

{E, S, U}, {S, E, U}, {E, U, S}, {S, U, E}, {U, E, S}, {U, S, E}

Using induction, the fact that, 4! or 24 permutations of four nationalities exist should lead you to the general form of n! permutations for each of the variables. There are 120 possible combinations of the five nationalities (5! = 120). Likewise, there are 120 different ordered lists of five values (i.e., 120 5-tuples) for pets, 120 5-tuples for drinks, and so on.

Exhaustive Search

Assuming that we know how to generate all 120 permutations for each attribute (we will address this topic shortly), the problem can be reduced to an exercise in combining all of the 5-tuples to produce all possible positional combinations of the attributes. Nested loops can be used to generate each of these combinations (Knuth, 2011).

For each nationality
    For each house color
        For each drink
            For each snack food

For each pet
Test the combination of attributes against the rules.

Implementation

Generating all 120 permutations for each attribute is probably the most challenging part of this problem. We have done that for you and provided a jar file containing our class file for you to use. You will not have the source code, but rather the compiled code and the associated javadoc-generated HTML pages.

The idea is to create an input string that represents unique characters for each attribute and call the code in the library (i.e., jar file) to create all of the permutations for you. For example, the pets can be represented by the string "dsfhz" (a dog in house 1, snails in house 2, a fox in house 3, a horse in house 4, and a zebra in house 5).

The following method invocation will return a collection of 120 strings representing all of the possible permutations of placing the pets in the different positions (i.e., houses):

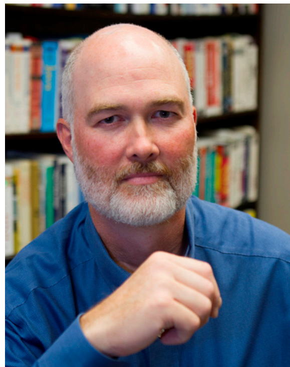
pets = Combinatorics.generatePermutations("dsfhz");

The first test (performed in the inner-most loop of Algorithm 1) could be the following combination:

ESUNJ (nationalities)
rgiyb (colors)
ctmow (drinks)
okclp (cancerSticks)
dsfhz (pets)

You need to write the code to figure out that this combination does not adhere to the rules (Statement 6 is violated since the *g* is not to the right of the *i* in this configuration). Only 24,883,199,999 more to go!

Biographies



John Crabtree is a Professor in the Department of Computer Science and Information Systems at the University of North Alabama. He holds a PhD in Computer Science from the Colorado School of Mines. Before joining the faculty at UNA, Dr. Crabtree was a software consultant in the Denver area. His research interests include algorithms, cheminformatics, GIS, and software engineering. He currently teaches software engineering, e-commerce, enterprise architecture, database management systems, and object-oriented programming using C++, Java, and PHP.

Xihui Zhang is an Associate Professor of Computer Information Systems in the College of Business of the University of North Alabama. He earned a Ph.D. in Business Administration with a concentration in Management Information Systems from the University of Memphis. His teaching and research interests include the human, social, and organizational aspects of Information Systems. He has published in such leading journals as the *Journal of Strategic Information Systems*, *Information & Management*, and *Journal of Database Management*. He earned two master's degrees, one in Business Administration and one in Engineering Technology, both from the University of Memphis. He also earned a master's degree and a bachelor's degree in Earth Sciences from Nanjing University in China. He serves on the editorial review board for several academic journals, including the *Journal of Computer Information Systems*, *Journal of Information Systems Education*, and *Journal of Information Technology Management*.