USING GRAPHICAL EXAMPLES TO MOTIVATE THE STUDY OF RECURSION*

Ben Stephenson
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
ben.stephenson@ucalgary.ca

ABSTRACT

Recursion is an important part of a complete computer science education. It is a topic that is often introduced during the first course, and then revisited when tree data structures are introduced and functional languages are discussed. Because recursion is introduced early in the curriculum, the range of problems that can be discussed when it is first encountered is limited. In particular, because students generally haven't been introduced to data structures like trees or efficient sorting algorithms such as quick sort and merge sort, these areas which use recursion to elegantly solve important problems can't be used to motivate its study.

This paper describes three graphical problems which can be used to motivate the study of recursion. Each of the examples demonstrates the utility of recursion while being appropriate for students nearing the end of their first term in a computer science course. In particular, these examples can be understood and implemented by students who have a reasonable understanding of loops, function calls, and, for some problems, arrays. They do not require any knowledge of object oriented design, pointers or complex data structures. Furthermore, while iterative solutions exist for each of the problems that we describe, the iterative solution for each problem is more complex than its recursive solution. We believe that this is an additional strength of these examples because it helps convince students that recursion is a tool that will allow them solve some problems more easily.

---

1 INTRODUCTION

In many introductory computer science courses, students are introduced to imperative control structures such as for loops and while loops before the notion of recursion is introduced. Students generally find these imperative structures reasonably intuitive because they are providing instructions to the computer in a manner similar to how they have received instructions throughout their lives.

Solving problems recursively requires students to expand the way that they look at problems and consider them from another perspective. Students often find using the new technique difficult, as has been documented in numerous previous studies [1, 3, 5, 9], among others. In our experience, this problem is exacerbated when students are asked to solve a problem recursively that has an obvious imperative solution. As such, we believe that carefully selected examples which meet each of the criteria outlined below can aid in students' understanding of this important topic.

- **Utility of Recursion:** The problem being solved must be at least as easy to solve using recursion as it is to solve using the imperative control structures students have already been introduced to.
- **Relevance:** The problem, or an extension of it that can be outlined to students, should serve some practical purpose.
- **Engaging:** The problem being solved should be interesting to the students who are being taught.

We have identified three graphical problems which meet each of the constraints that we have outlined. The problems include the generation of fractal images, implementing the flood fill or "paint bucket" tool present in many image manipulation applications, and using recursion to find a path through a maze.

2 COMPELLING USES OF RECURSION

This section presents three graphical problems with recursive solutions. Each problem meets our goals of demonstrating the utility of recursion, solving a relevant problem, and engaging students. Numerous graphics libraries and packages are available for introducing graphics at the first year level. We have successfully used QuickDraw [8] with each of these examples, and as such, have used it when presenting code in the remainder of this section. However, it should be possible to use these examples with any other graphics package that provides similar features.

2.1 Generating Fractal Images

We have previously taught recursion using traditional examples such as computing factorials and Fibonacci numbers, and determining if a string is a palindrome. The last time we taught recursion, we diverged from our traditional approach, replacing many of the traditional examples with examples that were graphical in nature. In particular, we examined fractal images as one motivating example for study in this area.

Fractal images are self-similar, meaning that when the image is broken into parts, each part is a smaller version of the whole image. Depending on the nature of the fractal,

the similarity may be exact, or it may be an approximation. Numerous visually stunning fractal images have been created, but are generally rather complex to generate. In contrast, simpler images that are also interesting can be generated recursively using less than 50 lines of code.

Common simple fractal images include the Sierpinski Triangle and fractal fern. When students examined these images, we found that they quickly observed that each part of the image was a smaller copy of the entire image. This led naturally to the idea that the same function that was used to create the first approximation of the fractal could also be used to generate the missing details by giving the function the ability to draw the image at different scales.

```
def tsquare(x,y,w,h):
  if (w < 4) or (h < 4): return

  print "fillrect", x + w/4.0, y+h/4.0, w/2.0-1, h/2.0-1

  tsquare(x,y,w/2.0,h/2.0)
  tsquare(x+w/2.0,y,w/2.0,h/2.0)
  tsquare(x,y+h/2.0,w/2.0,h/2.0)
  tsquare(x+w/2.0,y+h/2.0,w/2,h/2.0)

tsquare(272,172,256,256)
```

Figure 1: Python Code for Generating the Fractal T-Square Using QuickDraw

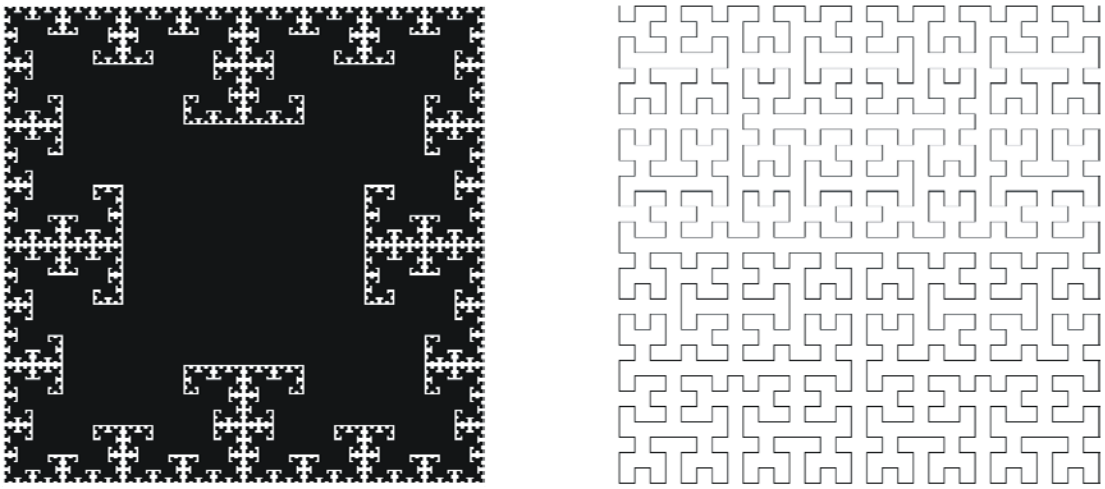

Figure 2: The Fractal T-Square (left) and Hilbert Curve (right)

Students' understanding of recursion was further reinforced through part of an assignment where they were asked to write a recursive function to draw the Fractal T-Square. This shape is constructed by drawing a square in the center of a square region, where the length of each side of the square that is drawn is half of the length of each side of the region. Using recursion, this shape can be drawn using less than 15 lines of code.

A sample solution to this problem and the resulting image are shown in Figure 1 and Figure 2 respectively.

While the total number of lines of code that need to be written is small, this task demonstrates several important aspects of writing a recursive function. First, it requires students to pass parameters to each recursive call to control the size and location of each square that is drawn. In addition, new values must be computed based on the parameter values provided to the current call in order to make subsequent recursive calls. Once complete, the program generates a modestly complex image that students could not easily generate in an iterative manner. Our strong impression was that students found this far more engaging than being asked to solve numerically based problems.

During the fall 2008 term, my colleagues used a slightly more complex fractal for their recursion assignment. The Hilbert Curve is defined by the pair of mutually recursive functions shown below.

```
R(0) = do nothing
R(k > 0) = - L(k-1) F + R(k-1) F R(k-1) + F L(k-1) -
L(0) = do nothing
L(k > 0) = + R(k-1) F - L(k-1) F L(k-1) - F R(k-1) +
```

Within these definitions, '+' denotes a 90 degree left turn, '-' denotes a 90 degree right turn, and F denotes drawing a forward line segment one unit in length. The position and direction of the cursor can be computed using the definitions above. However, this presents a significant challenge to most first year students, and doesn't reinforce the concept of recursion. This difficulty is easily overcome by using a turtle graphics environment which supports operations such as turning and moving a cursor as standard operations. In such an environment, the Hilbert Curve can be drawn using approximately 40 lines of code. A Hilbert Curve drawn with *k* initially set to 5 is shown in Figure 2. The complexity of the curve increases with the value of *k*.

Using the Hilbert Curve as an assignment provides many of the same benefits of drawing the fractal t-square. It still requires students to pass a parameter to each function call, and to use that parameter to compute a value passed to the recursive call. In addition, this task demonstrates that recursion can occur indirectly. While the function L calls itself directly, it also calls itself indirectly by calling R, which subsequently calls L.

Numerous simple fractals exist which are appropriate for use as assignments in a first year course. Any of these can be used to engage students by having them generate graphically complex and interesting images. Generating fractals also effectively demonstrates the utility of recursion because most fractal images are difficult to generate iteratively without using a data structure such as a stack or queue. Finally, once students have created simple fractal images themselves, they are able to appreciate how fractals can be used for useful tasks such artificial terrain generation [7].

2.2 Flood Fill

The flood fill algorithm is used to identify all of the elements in a two dimensional array that are connected to a specific element. The algorithm is easily extended to additional dimensions, but such extensions are not needed for the specific problems that we will consider in this paper. While the definition of connected elements can vary by the

problem that is being solved, it generally consists of elements that are the same as, or highly similar to, the specific element that was selected.

There are numerous uses for this algorithm. One graphical application is the flood fill or "paint bucket" tool that is commonly available in image editing software. This tool changes the color of a connected region in the image to a new color without impacting other unconnected pixels of that color. It is normally used by clicking on a single pixel in the image. Then the color of that pixel is identified, and all connected pixels of the same color are replaced with the new color.

We have successfully used this algorithm as part of an assignment on recursion. Students were provided with code that loaded an image from disk, storing it into a two dimensional array of integers. They were also provided with code that would display the data stored in a two dimensional array of integers on the screen as an image. The students were asked to implement the flood fill operation and demonstrate its functionality by changing the color of some of the regions in the test images.

The flood fill algorithm can be implemented recursively or iteratively. However the iterative solution requires the use of a stack or a queue. Since we cover recursion before these data structures are discussed, the existence of an iterative solution is not identified by most students. Even when the existence of an iterative solution is recognized, this example continues to demonstrate the utility of recursion because the iterative solution is no simpler than the recursive solution.

```
public void floodfill(int x, int y, int c) {
  if ((x < 0) || (x >= getWidth()) ||
      (y < 0) || (y >= getHeight())) {
    throw new RuntimeException("...");
  }

  ff_helper(x, y, c, getPixel(x,y));
}

private void ff_helper(int x, int y, int c, int old_c)
{
  if ((x >= 0) && (x < getWidth()) &&
      (y >= 0) && (y < getHeight()) &&
      (getPixel(x,y) == old_c)) {
    setPixel(x,y,c);

    ff_helper(x+1,y,c,old_c);
    ff_helper(x-1,y,c,old_c);
    ff_helper(x,y+1,c,old_c);
    ff_helper(x,y-1,c,old_c);
  }
}
```

Figure 3: Java Code for Implementing Flood Fill

A recursive solution to flood fill can be implemented in approximately 25 lines of Java code, as shown in Figure 3. In this listing, we have assumed that flood fill is being implemented as a member of an image class that modifies the image represented by the receiver object. We also assume that common methods such as getPixel, setPixel, getWidth and getHeight have either been provided or previously implemented by the student. While this provides cleaner code, the problem can also be used without using

object oriented techniques by explicitly passing the image to each function call that requires it.

Flood fill builds on our previous fractal examples by demonstrating an additional concept that students need to be familiar with in order to use recursion effectively. In particular, it demonstrates that a recursive method may require data beyond what is provided by the parameters specified for the method. In this specific case, one would expect to perform a flood fill by invoking a method that takes three parameters: the x and y coordinates where the fill will begin, and the new color that should be used. However, these values are not sufficient to implement a recursive solution successfully because the recursive function needs to know what color is being replaced in order to detect the boundary for the region that is being filled.

To overcome this problem, students needed to create two methods. The first is the public interface that programmers can use to initiate a flood fill on the image. It is a reasonably simple method that performs some basic error checking, acquires the additional value required for the recursive solution (the color that is being replaced), and initiates the recursive solution. The second method is private, takes one additional parameter, and performs the actual recursive work required to fill the specified region in the image.

In our experience, students found implementing flood fill more challenging that drawing a fractal. This challenge stemmed from the need to manipulate a data structure instead of drawing directly to the screen as the recursion progressed, and the need to create the second function that takes an additional parameter. While this problem was more challenging, students found it highly engaging because it allowed them to see how a tool that they have used could be implemented.

2.3 Maze

The final example that we will consider in this paper is using recursion to search for the solution to a maze. While we have not yet used this example with students, we anticipate doing so during the current term because we believe that this example is also well suited to teaching students about the importance of recursion. We expect students to find this problem relevant and engaging because finding a path through a maze is a component of some computer games. Like the previous examples, this example also clearly demonstrates the utility of recursion because solving this problem iteratively requires data structures that haven't yet been introduced.

We have chosen an intuitive representation for the maze – a two dimensional array. Within this array, each element can initially contain one of four possible values:

- A barrier
- An open space
- The start of the maze
- The exit from the maze

As the solution progresses, blocks can take on additional values indicating that a space is part of the path from the starting location to the location that is currently being explored, or that a space has been visited previously and should not be considered again. By traversing the two dimensional array, one can easily draw an overhead view of the

maze by drawing squares of different colors to represent each of the possible values of a block.

In addition to demonstrating the utility and relevance of recursion in an engaging manner, we believe that this is also a particularly strong example because it is reasonably easy to animate the algorithm as progress is made toward the solution. This gives a visually interesting result and may help students gain greater understanding of the steps that there program is taking to reach a solution. The animation also allows students to see the difference in behaviour of their program when the search order is changed by performing the recursive calls in a different order.

In Python, our solution to this problem is approximately 120 lines, including the code necessary to load a description of the maze from a file and animate the solution. As a result, we feel that this is appropriately challenging problem to use as an assignment in a CS2 course. In CS1 it may be necessary to provide the students with the code necessary to load the maze from the file so that the amount of code that must be written is reduced, and so that they can concentrate on the recursive task.

We have also explored the possibility of animating the solution to a maze in three dimensions. Using a tool like QuickDraw, or another high level 3D library, it is possible to generate a simple 3D world with relatively little code. By moving the camera as the algorithm progresses, the exploration of the maze can be visualized. However, while this gives a nice visual result, it significantly adds to the complexity of the program. Our very simple 3D animation is approximately 250 lines of code. Furthermore, we found that building and navigating a 3D model added a significant level of complexity without offering any obvious advantages with regard to teaching students about recursion. As a result, we do not anticipate requiring students to create a 3D animation when we use this example.

2.4 Examples Summary

This section has outlined three compelling examples of recursion that are appropriate for an introductory computer science course. Each example solves a non-trivial problem which is visually interesting, successfully engaging students and demonstrating the relevance of recursion. In addition, the problems considered cannot be solved iteratively without using data structures that the students have not yet been exposed to. As such, these examples also effectively demonstrate the utility of recursion.

3 RELATED WORK

Recursion is widely recognized as an important part of the undergraduate computer science curriculum that is also challenging to teach effectively. In this section, we consider those studies that are most closely related to our own work on using graphical examples to improve students' understanding of recursion. The following paragraphs explore related work involving fractals and mazes. To the best of our knowledge, no previous work has been published on using flood fill as a motivating example for recursion.

Fractals have been used to motivate the study of recursion in both Prolog [2] and Java [4]. Based on our observation of our own students, we strongly believe that we were also successful in using fractals to motivate the study of recursion with first year students using imperative languages (Pascal and Python). In addition to considering imperative languages, we have extended this work by minimizing the arithmetic involved in two ways. First, we consider fractals that are mathematically simpler in a Cartesian coordinate system such as the fractal t-square. In addition, we have shown that turtle graphics can be used to simplify drawing some other fractals to a level that is appropriate for first year students. Each of these simplifications is important because they allow students to concentrate on the recursion rather than the arithmetic.

Using a maze as a compelling example of recursion has been documented previously in [6]. In this study, students were provided with a maze abstract data type to hide many of the internal details of the maze representation, and the process of displaying the maze. Providing the abstract data type was necessary because the maze representation was somewhat complex, with barriers existing between the cells. Our proposed representation of the map is simpler, with each square either being open or filled with a barrier. As a result, it should not be necessary to provide support code to the students, especially at the CS2 level. In addition, we believe that extending this example with animation as we have proposed will significantly increase its value.

4 CONCLUSION

This paper has described three interesting graphical problems that can be solved recursively. In each case, using recursion provides a tidy solution with reasonable performance. Solving these problems iteratively requires knowledge of data structures beyond arrays, making them effective demonstrations of the utility of recursion, particularly to students who have not yet taken a data structures course. Finally, since the problems are visual, we found that students find them more engaging than traditional examples, and as such, these examples help motivate students to spend more time on this important, but often challenging subject.

REFERENCES

[1] J. Angel Velazquez-Iturbide. Recursion in gradual steps (is recursion really that difficult?). In SIGCSE ’00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education, pages 310–314, New York, NY, USA, 2000. ACM.

[2] Bruce S. Elenbogen and Martha R. O’Kennon. Teaching recursion using fractals in prolog. In SIGCSE ’88: Proceedings of the nineteenth SIGCSE technical symposium on Computer science education, pages 263–266, New York, NY, USA, 1988. ACM.

[3] David Ginat. Do senior cs students capitalize on recursion? SIGCSE Bull., 36(3):82–86, 2004.

[4] Aaron Gordon. Teaching recursion using recursively-generated geometric designs. J. Comput. Small Coll., 22(1):124–130, 2006.

[5] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. SIGCSE Bull., 37(3):14–18, 2005.

[6] Ivan B. Liss and Thomas C. McMillan. An amazing exercise in recursion for cs1 and cs2. In SIGCSE ’88: Proceedings of the nineteenth SIGCSE technical symposium on Computer science education, pages 270–274, New York, NY, USA, 1988. ACM.

[7] Przemyslaw Prusinkiewicz and Mark Hammel. A fractal model of mountains with rivers. In Proceedings of Graphics Interface ’93, pages 174–180, 1993.

[8] Ben Stephenson and Craig Taube-Schock. QuickDraw: Bringing Graphics Into First Year. In SIGCSE ’09: Proceedings of the 40th ACM technical symposium on Computer science education, pages 211–215, New York, NY, USA, 2009. ACM.

[9] Cheng-Chih Wu, Nell B. Dale, and Lowell J. Bethel. Conceptual models and cognitive learning styles in teaching recursion. In SIGCSE ’98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education, pages 292–296, New York, NY, USA, 1998. ACM.