600.112: Intro Programming for Scientists and Engineers

Assignment 5: Recursion and Backtracking[*]

Peter H. Fröhlich
phf@cs.jhu.edu

Joanne Selinski
joanne@cs.jhu.edu

Due Dates: 3pm on Thurs 10/15 & Weds 10/21

Introduction

This project for *600.112: Introductory Programming for Scientists and Engineers* is all about *recursion* and *backtracking search*. It's one of the few assignments that doesn't have a specific application from science or engineering as its basis, just to mix things up. It does however focus on key concepts in computer science.

There are three things to do: First you'll write a program to solve the *Knight's Tour* problem on a chess board. Second you'll write a program to solve instances of the well-known *Sudoku* puzzle. Third, for optional extra credit, you'll write programs to draw some interesting recursive structures using *turtle graphics* once again. As usual, submit a complete zip file on Blackboard including some text input files for testing your knight.py and sudoku.py solution.

We'll continue working with lists in this project, expanding their use to two-dimensional matrices. Starting with this assignment we will be using the pep8 tool to insure our programs follow standard style guidelines. We will also continue to use the doctest tool to do unit testing of our functions. See part 5 of the posted Installing Python Tools document for instructions on installing the pep8/doctest plugin for IDLE, as well as class slides on using them. Part of your grade for each program will be based on your testing and styling.

You must submit a complete zip file with all *.py and *.txt files needed to run your solutions as detailed below, and on Blackboard before the deadlines. Don't forget to include descriptive doc strings for all your programs and the functions in them. Also, you should provide doctests for as many functions as possible, continuing the testing techniques developed in the last assignment. It's best to write these as you go along. Improperly written doc strings (missing the closing characters for example) can result in a 0 for your program! Remember that you can use the "Check Program" option in the IDLE Run menu to see if your program has any syntax errors. If it does, you *must* fix them in order to avoid a 0 grade.

Background: Recursion

The factorial function $n!$ shows up in a wide variety of places in mathematics and therefore also in science and engineering. Consider, for example, the Taylor expansion of a trigonometric function:

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

Or consider, in probability theory, the number of possible permutations of a finite set: A set of n elements has $n!$ permutations. The factorial function is often defined informally as the product $1 \times 2 \times 3 \times \cdots \times n-1 \times n$, but "the dots" leave something to be desired: the definition is not entirely explicit. Of course we could use a notation such as $\prod_{i=1}^{n} i$ instead, but now we're "hiding" something under the $\prod$ symbol instead of the $\ldots$ symbol. There is, however, a completely explicit definition of the factorial function:

$$n! = \begin{cases} 1 & \text{if } n \leq 1, \\ n \times (n-1)! & \text{otherwise.} \end{cases}$$

[*]Disclaimer: This is *not* a course in physics or biology or epidemiology or even mathematics. Our exposition of the science behind the projects cuts corners whenever we can do so without lying outright. We are *not* trying to teach you anything but computer science!

If we want to compute 3! according to this definition, we have to compute $3 \times 2!$ which in turn leads us to $3\times2\times1!$ which is $3\times2\times1$ which is (of course) 6. This is a *recursive* definition because we define the factorial function in terms of itself. At first this may seem a bit circular, but it is not: we can only subtract 1 from n a certain number of times before reaching 1, and the result for $n \leq 1$ does not rely on the factorial function anymore. Roughly speaking a recursive definition is *well-founded* if (a) the recursive cases lead to "smaller and smaller" problems and (b) we eventually reach a "base case" that is not recursive. In the definition above, the first case for $n \leq 1$ is our base case, the second case is our recursive case. Regardless what n we start with, sooner or later we'll reach the base case and the recursion stops.

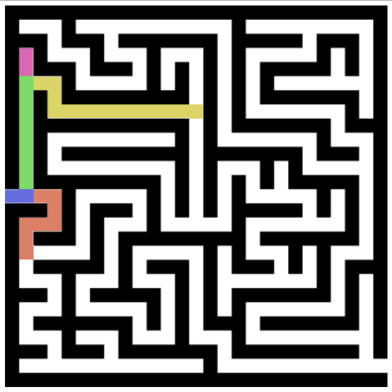
We can translate both definitions of the factorial function into PYTHON code in the obvious way:

```
def iterative(n):
    result = 1
    for i in range(2, n + 1):
        result = result * i
    return result

def recursive(n):
    if n <= 1:
        return 1
    else:
        return n * recursive(n - 1)
```

The call of `recursive(n - 1)` in the last line is a *recursive call* since we call the very function we are defining. Both of these functions are valid PYTHON code, and both of these functions will compute the correct result. However, there is one *very important difference* between them: The `iterative` function needs memory for exactly *three* variables: `n`, `result`, and `i`. The `recursive` function, on the other hand, needs memory for n variables: one `n` for the first call, one `n` for the second call, ..., and one `n` for the nth call. So if we compute 10! the `recursive` function will use *more memory* (10 variables) than the `iterative` function (3 variables).[1] Since both functions are otherwise identical in their behavior, nobody in their right mind would suggest using the recursive factorial function in an actual program: why waste more memory than we have to? Even in terms of how complicated the functions are to read, there is really no significant difference. There are, however, certain problems for which a recursive solution is *much simpler* than the equivalent iterative solution would be. It's for those problems that recursion really begins to shine as a programming technique. Sadly we had to explain it with something simplistic like the factorial function first.

Figure 1 Backtracking search in a simple maze.



Background: Backtracking

Consider the task of solving a maze like the one in Figure 1: Starting from the entrance on the left, what sequence of steps will get you to the exit on the right? There are many possible solution strategies but let's focus on one in particular:

- At any point in the maze, arbitrarily decide on one possible direction we have not tried before.
- If we reach a dead end (a point where we cannot pick a new direction anymore), a previous decision was flawed; undo as many steps as necessary to make a different decision.
- If we reach the exit, we are done.

This process is illustrated with different colors in Figure 1 above. We begin in blue at the entrance where we only have one option, namely to go right. After that first step we have a choice: We can either go up (green) or continue right (orange). We decide to go up, which leads to a number of additional steps up because there are no other options, at least until we get to the next intersection. Here we can decide again: We can either continue up

1. Furthermore PYTHON limits the number of recursive calls we can make to some fixed number. On most systems we can only make 1000 recursive calls, so the `recursive` function can only be used to compute 1000! or thereabouts. The `iterative` function has no such limitation.

(pink) or go right (yellow). We decide to go up again, but after a few steps we are at a dead end: the last decision to go up was bad, so we return to it (green) and go right instead (yellow). I don't want to belabor this process, suffice it to say that we won't ever find the exit after that decision either. So eventually we are back (once again) at the last green decision and we are out of options here as well. So we have to return to the last decision before that one, which was our initial "going up" decision at the last blue square. We now make that decision differently and go right instead (orange). And so on and so forth.

The process of making different decisions at different points and "undoing them" if they don't lead to the goal is called *backtracking* and recursion is particularly well-suited to it because the "last decision made" is not something we need to explicitly remember anywhere, the recursive calls remember that information automatically.

1 A Knight's Tour [15 points]

This problem deals with a lonely knight on an otherwise empty chess board. Knights move in a peculiar way: either one square north-south and two squares east-west or two squares north-south and one square east-west. A knight in (roughly) the center of the board can reach eight different squares from the one it is sitting on; the closer it gets to a border, the fewer moves are possible since some would take it outside the board. The question we are going to address is the following: How can we move a knight in such a way that it will *visit each square* of the chess board *exactly once*? A sequence of moves that achieves this is called an *open knight's tour*.[2] Obviously the *size* of the chess board is relevant for this problem.[3] On a 1-by-1 board we're done immediately since we have only one square to visit and we do that by placing the knight there. On a 2-by-2 board, however, we can never actually move the knight after placing it on the first square, thus no knight's tours exist. On a 3-by-3 board we can move the knight around the outer 8 squares, but we can never move it into the center square; if we start at the center square, however, we cannot move anywhere else; so again no knight's tour exists. On a 4-by-4 board no knight's tour exists either, but on a 5-by-5 board there are suddenly 304 possible open knight's tours if we start in a corner of the board. There is actually a total of 1728 open knight's tours on a 4-by-4 board, but some starting squares do not lead to a tour. So apart from the size of the board, the starting position matters as well.

Figure 2 Open knight's tour on 5-by-5 board.

```
  1   12    3   18   21
  4   17   20   13    8
 11    2    7   22   19
 16    5   24    9   14
 25   10   15    6   23
```

Figure 3 Open knight's tour on 6-by-6 board.

```
  1   20    3   18    5   22
 36   11   28   21   30   17
 27    2   19    4   23    6
 12   35   10   29   16   31
  9   26   33   14    7   24
 34   13    8   25   32   15
```

For this problem you will write a program that computes *one* possible knight's tour for a chess board of a given size.[4] Please call your program `knight.py` and nothing else. Figure 2 shows what the output of your program should look like for a 5-by-5 chess board, Figure 3 shows the output for a 6-by-6 chess board. As you can see we start in the upper left corner of the board and then make our way around following the movement pattern for a knight. We number the squares indicating the order in which they were visited.

We need two essential pieces of information to find a knight's tour: the size of the board and the valid moves of a knight. Since we are trying to get a version of this program working quickly and understand scoping rules better, we'll use global variables for a change. Both of these values should be defined at the top of your program *exactly* as shown

2. A *closed* knight's tour is one in which the knight *ends up* in the same square it *started from*, so that square is visited twice (once at the beginning and once at the end). We will ignore closed tours for this problem.

3. Computer scientists have a tendency to generalize everything as far as possible (something we obviously learned from mathematicians). Hence we are not going to limit ourselves to just "regular" 8-by-8 chess boards. We will, however, stick to square chess boards.

4. Computing *all* possible tours from a given starting position is actually not much harder, but it would take *way* too long to print them all for $n \geq 6$.

here in the first draft:

```
SIZE = 6
MOVES = (
    (−2, −1), (−2, 1),
    (2, −1), (2, 1),
    (−1, −2), (−1, 2),
    (1, −2), (1, 2),
)

def main():
    print ”No␣solution!”

main()
```

The moves are being stored in a tuple of pairs. Each pair is also a tuple, and represents the difference in row and column indices between the current position of the knight on the board, and the position after the move. For example, (-2, -1) will move the knight two rows up (north) and one column to the left (west). Likewise, (1, 2) would move it one row down and 2 columns to the right. Therefore, the `MOVES` collection of pairs represents all possible legal moves for a knight.

Of course we need to represent the actual chess board as well, and we'll use a matrix (nested lists) to do that. Concretely we'll use a matrix of integers, all of which are 0 initially which we'll take to mean "not yet visited" in the program; if a cell of the matrix contains a positive integer, it's the "step" in the tour we're currently investigating. So when we first place the knight in the upper left corner of the board, we'll set matrix element (0, 0) to 1 to represent the first move. The next square we jump to will be labeled 2, the next 3, and so on; if we ever label a square with n^2 (where n is the size of the chess board) that was our last move. The technique we will use to find a knight's tour is *backtracking* as we've explained above. Suppose we are at some square and we are trying to "finish" the tour from that square; there may be a number of possible moves we can make, so we'll pick one which leads us to a new square from which we in turn try to finish the tour again. If we can't finish the tour because we are not done yet, but we have no more possible moves to make from a given square, then we "undo" as many decisions as necessary to pick a different option earlier and try to finish the tour again. We'll do this for as long as we're not out of options at *all* squares; if we run out of options and have never finished a tour, we give up. So our main program will look like this:

```
def main():
    board = zero_matrix(SIZE, SIZE)
    finish_tour(board, 0, 0, 1)
    print ”No␣solution!”
```

We first create a zero matrix to represent the board. Once the board is initialized, we call the function that will (recursively!) finish the tour from the given starting position; we always start in the upper left corner of the board, and that square will be labeled with 1 as indicated in the call. So why do we still print "No solution!" after we return from that function? We will write the function in such a way that if it actually succeeds in finding a tour, it'll print the tour and then stop the program; in other words, if we are successful, the function will *not actually return* to `main` at all! The only time we do return to `main` is if we could not find a tour, and in that case printing the message is the correct thing to do.

To create the board, we can use a mixture of loops and list comprehensions, similar to some of the matrix operations we explored in lecture. We want to write the function more generally than we will be using it, by giving it a parameter for the number of rows and the number of columns. Here is a starting point with documentation and tests; examine some of the special cases carefully:

```
def zero_matrix(rows, cols):
    '''
␣␣␣␣Create␣a␣matrix␣with␣the␣specified↩
        ␣number␣of␣rows␣and␣columns,
␣␣␣␣with␣all␣values␣initialized␣to␣0.
␣␣␣␣>>>␣zero_matrix(0,␣0)
␣␣␣␣[]
␣␣␣␣>>>␣zero_matrix(1,␣1)
␣␣␣␣[[0]]
␣␣␣␣>>>␣zero_matrix(2,␣4)
␣␣␣␣[[0,␣0,␣0,␣0],␣[0,␣0,␣0,␣0]]
␣␣␣␣>>>␣zero_matrix(3,␣3)
␣␣␣␣[[0,␣0,␣0],␣[0,␣0,␣0],␣[0,␣0,␣0]]
␣␣␣␣'''
    pass
```

Now we can continue to the function statements, building the matrix one row at a time. We use a simple list comprehension to create one row, and then append a *copy* of each row to the matrix we are creating.

```
def zero_matrix(rows, cols):
    # documentation and tests here...
    # make an empty list to start
    mtrx = []
    # create one row
    arow = [0] * cols
    # repeat for each row
    for _ in range(rows):
        # use [:] to create copies of ↩
            the row
        mtrx.append(arow[:])
    return mtrx
```

Now to the tricky part - recursion! Our first draft of the `finish_tour` function is as follows:

```
def finish_tour(board, x, y, move):
    board[x][y] = move
    if move == SIZE*SIZE:
        pass
    else:
        pass
```

The first thing we do is record the move we are supposed to make on the board. Then we check if this was the last move; if it was, we need to print the tour and exit the program; if it wasn't, we need to make a move from the current square and finish the tour given that move. Filling in the first part is easy:

```
def finish_tour(board, x, y, move):
    board[x][y] = move
    if move == SIZE*SIZE:
        print_board(board)
        exit()
    else:
        pass
```

You'll need to add one doctest to this function that matches the output given in Figure 3.

Next we need to write a function to print the board in the nice format illustrated in Figures 2 and 3:

```
def print_board(board):
    for i in range(SIZE):
        for j in range(SIZE):
            print "%4d" % board[i][j],
        print
```

We simply run over the matrix as we've done several times before, printing each element. However, in order to get a nicely formatted "square" we use PYTHON's string formatting operator `%` to force a width of 4 for each integer we print; if the integer is not "long enough" PYTHON will "fill in" enough spaces at the front to line everything up nicely.

The second part of `finish_tour` is a little bit more complex. First we have to figure out what the valid moves are that we can make from the square we are currently on, then we have to pick one of those moves and try to finish the tour using it; but if we cannot finish the tour, we have to try the next possible move and try to finish again. So we'll first write a function that given the current state of the board and the current position of the knight, will return a list of possible squares to move to next:

```
def possible_squares(board, x, y):
    result = []
    for move in MOVES:
        dx, dy = move
        nx = x + dx
        ny = y + dy
        if valid(nx, ny) and board[nx↩
            ][ny] == 0:
            result.append((nx, ny))
    return result
```

In other words, we'll try every legal move a knight can make from the current square; if the move stays within the board, and if we have not visited the resulting square yet, then we'll include that position in the list of possible squares to jump to. Due to the size of the board in this version of the problem, you do not have to write doctests for the possible squares function. However, we strongly suggest using a smaller board size to work through this part of the problem if you don't understand all the steps.

The `valid` function is very straightforward, but you must add doctests to it:

```
def valid(x, y):
    return 0 <= x < SIZE and 0 <= y < ↩
        SIZE
```

These functions make it rather straightforward to complete the `finish_tour` function:

```
def finish_tour(board, x, y, move):
    board[x][y] = move
    if move == SIZE*SIZE:
        print_board(board)
        exit()  # quit the program ↩
            entirely
    else:
        for position in ↩
            possible_squares(board, x, ↩
            y):
            nx, ny = position
            finish_tour(board, nx, ny,↩
                move+1)
        board[x][y] = 0
```

So we try to finish the tour for every possible move the knight can make from the current square; if we succeed we won't return, but if don't succeed, we'll have to eventually "undo" the move we were told to make when we were first called, which is why we assign 0 to the current square after we exhaust all possible ways to finish a tour from it: we'll have to go back and "undo" an earlier decision. And done.

Figure 4 An easy Sudoku puzzle.

4	1	?					3	7
6	2						9	8
		5	9		2	1		
		2	7		3	4		
				5				
		1	4		6	9		
		4	3		8	6		
2	5						4	3
9	6						1	2

Figure 5 Input file for the easy Sudoku puzzle.

```
41.....37
62.....98
..59.21..
..27.34..
....5....
..14.69..
..43.86..
25.....43
96.....12
```

Except for one thing: If you try to solve the 6-by-6 board using this approach, you'll notice that it takes a rather long time. On a netbook, the 5-by-5 board is done in 1.3 seconds while the 6-by-6 board takes over 4 minutes! The reason for this is that there are many, many more possibilities to explore on a 6-by-6 board, and our program will try all of them until it finds a tour. If you want, you can look up "Warnsdorff's Rule" online which is a way to restrict our search to only certain possible moves of the knight. With that rule added, you can solve the 6-by-6 board in *under* 0.1 seconds on a netbook; a 31-by-31 board takes roughly 0.3 seconds; for a 32-by-32 board it runs out of memory though. Amazing!

If you work on Warnsdorff's Rule, please hand in a separate program called `warnsdorff.py`, the `knight.py` program should *not* use the additional rule!

2 Solving Sudoku [25 points]

Sudoku is a well-known number-placement puzzle: Fill a 9-by-9 grid with digits so that each column, each row, and each 3-by-3 sub-grid ("block") contains *all* the decimal digits from 1 to 9 *exactly once*. If you're not familiar with this game, you can find a explanation of it online at www.sudoku.com.

Consider the puzzle in Figure 4 and imagine it as a 9-by-9 matrix with rows and columns numbered from 0 to 8. If we want to solve Sudoku puzzles by backtracking, we'd have to start at the first "free" square, marked with "?" at position (0, 2) here. In the most simplistic approach, we'd just put each digit from 1 to 9 into that position and then try to solve the rest of the puzzle under that assumption.

However, notice that the constraints of the puzzle restrict our choices rather severely: If we placed a 1 for the "?" we'd immediately fail because there's already a 1 in row 0; if we placed a 2 instead we'd fail again because there's already a 2 in column 2; same for 3, 4, and 5; if we placed a 6 instead we'd fail once more because there's already a 6 in the first block of the puzzle. The existing constraints make it so that the only possible options for the "?" at position (0, 2) are 8 and 9, nothing else is allowed. We'll use this to our advantage: Instead of trying every possible digit from 1 to 9, we start by computing the set of actually valid digits for a certain position. We'll then only pick from that set, one at a time, and then try to solve the rest of the puzzle recursively. If we ever get to a position where we have no options whatsoever, so the set of possible values at that position is empty, we have run into a dead end and we need to backtrack. If, however, we end up filling every last spot, we have found a solution to the puzzle: we print that solution and stop, just like in the knight's tour problem.

For this problem you will write a program that computes *one* possible solution to a given Sudoku puzzle. Please call your program `sudoku.py` and nothing else. Figure 5 shows what the input to your program, which you'll read from a file `puzzle.txt`, will look like. Figure 6 shows the corresponding output from your program, the solution to the Sudoku puzzle given in Figure 5.

Instead of guiding you step-by-step through the development of this program, we'll just give you a few hints about how to approach it:[5]

5. You'll have to start thinking these things through yourself at some point, and we're now at a place in the course where you should be comfortable enough with the basics so you can concentrate on the harder parts of programming.

Figure 6 Output for the easy Sudoku puzzle.

```
4 1 9 8 6 5 2 3 7
6 2 7 1 3 4 5 9 8
3 8 5 9 7 2 1 6 4
5 9 2 7 1 3 4 8 6
8 4 6 2 5 9 3 7 1
7 3 1 4 8 6 9 2 5
1 7 4 3 2 8 6 5 9
2 5 8 6 9 1 7 4 3
9 6 3 5 4 7 8 1 2
```

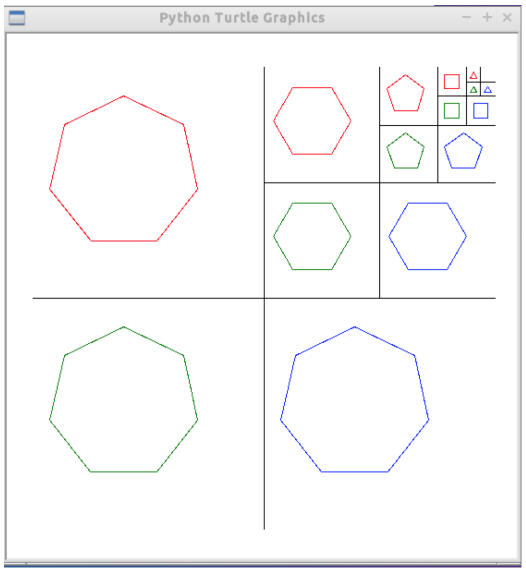
- Represent the puzzle as a matrix of integers. Cells that are not constrained initially should be 0, cells that are constrained should have the value from the input file in them.

- You'll need three functions that, given a matrix, can return a certain row, column, or block as a list. If you number the blocks from 0 to 8, left to right and top to bottom you can figure out which block a particular cell is in by dividing the cell's row and column indices by 3. Here is the exact formula, using integer division: $block = x/3 * 3 + y/3$ where $block$ is the block number [0,8], x is the row number of the cell and y is the column number of the cell.

- Before you pick a digit for an empty cell at position (x, y), compute the *set of possible values*: Start with the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$; look at the values that are already fixed in row x and remove them; then look at the values that are already fixed in column y and remove them; then look at the values that are already fixed in the block that contains (x, y) and remove them. Use the resulting set as the possible values for your recursive backtracking function.

- The simplest way to find the next cell to work on is to always start from the first cell and run through the matrix row-by-row, column-by-column, until you find a cell that's still 0 and therefore needs to be filled in. If you're looking for a 0 but don't find one anymore, it means that all your empty cells have been filled in and you have (hopefully!) solved the puzzle. There are, however, smarter ways of accomplishing both of these tasks.

Figure 7 Recursive images with depth 5 and size 500.



If everything goes well, your program for the Soduko solver should be just a little big longer than the program for the knight's tour. If you end up with way more code than that, you're probably doing something wrong or overly complicated.

Remember that your solutions for this assignment must be completely pep8 compliant. However, due to the nature of the problems, you do not have to write doctests for them, except where specified for the first part (Knight's Tour).

3 Recursive Images [20 points extra credit]

Combining recursion with graphics leads to a wide variety of interesting images. For the extra credit, you will use recursion to create a drawing of various polygons, in boxes of decreasing sizes. Please call your program `figures.py` and nothing else. The expected output of the program is shown in Figure 7 for a starting depth of 5 and size 500.

As you can see, each polygon will be drawn three times, in red, green and blue colors. As the figures get smaller the number of sides also decreases by one. We refer the total number of repetitions (recursions) as the *depth* of the image. The first (largest) set of polygons will have $depth + 2$ sides; this is sometimes called the *order* of the polygon. So for example, if you start with a

Good luck!

depth of 2, your program should draw two shapes - squares and triangles.

The recursively smaller RGB process we are creating here is actually used to represent images in computer graphics and is called 'mipmapping'. There is a nice article in Wikipedia explaining the process and its usage, with a recursive image somewhat similar to ours: https://en.wikipedia.org/wiki/Mipmap.

Your program must get the required depth of the image as well as the starting size of the overall image as input from the user when it begins. Then simply draw the figures. You can and should reuse code from the first turtle project to draw the polygons and box borders (lines). You'll have to play around a bit with starting positions to figure out a good starting point for drawing each polygon inside its box. They don't have to be perfectly centered, but do your best. You should also experiment a bit to figure out a good length for the sides of the polygons so that they mostly fill their boxes.

Note that extra credit points are only applied to your homework point total. However, when course grades are calculated, the homework averages will be capped at 100%. In other words, extra credit homework only helps you recover from poor homework grades, not anything else.