Problem Solving:
Algorithmic & Recursive

Paul S. Wang, Sofpower.com

May 13, 2023

To become a computational thinker we need just two things: (1) To have a good understanding of computing and digital technologies and (2) To be inspired and apply that knowledge to solve problems and improve the way we do things in all areas and in our daily lives.

At the center of computing is the idea of automation. Computers are universal machines that can be programmed to perform almost any task. They allow us to automate solutions to problems. Such solutions can be performed by computers repeatedly, reliably, precisely, and with great speed. Not all problems lend themselves to automated solutions. But computational thinkers prefer solutions that can be automated.

The stages of computer automation are *conceptualization*, *algorithm design*, *program design*, and *program implementation*. Specifying and implementing solution algorithms requires precise and water-tight thinking. Also required is anticipation of possible input as well as execution scenarios. Few are born with such talents. But, we all can become better problem solvers by studying cases, experimentation, and building our solution repertoire.

Often there are multiple algorithms to solve a particular problem or to achieve a given task. Some may be faster than others. Some may use less resources. Others may be easier to program or less prone to mistakes. Analyzing and comparing different solution approaches is also an important part of problem solving.

This post is part of our *Computational Thinking* (CT) blog where you can find many other interesting and useful articles.

Solving Puzzles

A good way to begin thinking about problem solving is perhaps by looking at a few puzzles.

Egg Frying

The Problem: A pan can fry up to two eggs at a time (Figure 1). We need to fry 3 eggs. Each egg must be fried for one minute on each side. Design an algorithm to fry the three eggs in as little time as you can manage.

Figure 1: Egg Frying

The slowest method would fry one egg at a time, cooking each side for one minute. It would take a total of 6 minutes.

A better method would fry two eggs, turning them over after a minute, and cook for another minute. After that, fry the third egg. This method takes a total of 4 minutes.

Is this the best we can do? Well, no. We can do better.

1. Start with frying two eggs for one minute.

2. Take one of the half-done eggs out of the pan, turn the other egg over, and add the third egg in the pan, fry for one minute.

3. Take out the egg that is done, flip the other egg over, and add back the half-done egg, fry for a minute.

4. Take both eggs out and terminate.

Each step takes one minute. The whole procedure takes 3 minutes to get the job done.

Again, is this the best we can do? Well, yes. How do we know? Can we prove that no method can be faster?

Here is a proof. The three eggs require 6 egg-side-minutes of frying. The pan can supply a maximum of 2 egg-side-minutes per minute. Thus, at least three minutes are needed to produce 6 egg-side-minutes, no matter how the frying is done.

Liquid Measuring

The Problem: We have a 7-oz cup and a 3-oz cup (Figure 2). Unfortunately, neither has any volume markings on it. We have a water faucet but no other containers. Find a way to measure exactly 2 ounces of water.

Figure 2: Two Cups

We see the allowable operations are filling water from the faucet, pouring water from one cup to the other, and emptying the cups. Here is how we can proceed.

1. Fill the 3-oz cup completely.

2. Empty the 3-oz cup into the 7-oz one.

3. Repeat steps 1 and 2 one more time.

4. Fill the 3-oz cup again completely.

5. Pour water from the 3-oz cup into the 7-oz cup until it is full.

6. Two ounces of water now remain in the 3-oz cup.

Let's now switch the 7-oz cup to an 8-oz cup. Can you solve the same problem? What if we switch it to a 6-oz cup?

A Magic Tray

The Problem: A magic tray is a perfect square and has four corner pockets (Figure 3) whose opening look exactly the same. Inside each pocket is a cup hidden from view. The tray also has a green light at the center.

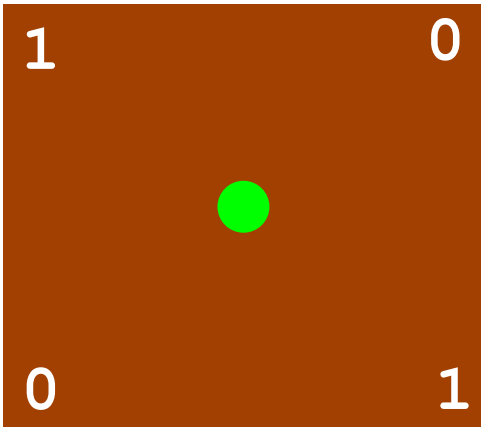

Figure 3: Magic Tray Puzzle

Cups inside the pockets can be either up (1) or down (0). The light will turn red automatically if all four cups are in the same orientation.

Your job is to turn the light red by performing a number of steps. Each step consists of reaching into one or two pockets to examine and optionally re-orient the cups. No other operations are allowed. Remember, you can't see the cups. Figure 3 shows one possible configuration to give you an idea.

To complicate things, the tray immediately spins wildly after each step. When it stops spinning, there is no way to tell which pockets you had examined in the last step.

Your job is to create an algorithm to turn the light red, no matter what the initial orientations of the cups are. Remember, an algorithm must specify exactly what to do at each step and guarantee termination after a finite number of steps. Therefore, keep reaching into pockets and turning cups up (or down) is not a solution because you may be extremely unlucky and reach into the same pockets every time.

One observation we can make is that we may choose to reach into pockets along a side or on a diagonal. But there is a chance, however small, that we may reach into the same side/diagonal all the time. Our algorithm must work even if it never ever examines all four pockets. This puzzle is fun and hits home the algorithm ideas perfectly. We will leave the reader to work out this algorithm. It should take no more than 7 steps.

Recursion

A circular definition is usually no good and to be avoided, because it uses the terms being defined in the definition, directly or indirectly. For example *Bright: looks bright when viewed*. Or *Adult: person not a child* and *Child: person not an adult*. A circular definition is like a dog chasing its tail. It goes on and on to no end. Figure 4 shows two mirrors reflecting infinitely into each other to illustrate the recursion concept visually.

However, a term or concept can be defined *recursively* when the definition contains the same term or concept without becoming circular. A *recursive definition* has *base cases* that stop the circling at the end. For people first exposed to recursion, the concept can be confusing. But, it is simple once you understand it. Please read on.

In mathematics, a *recursive function* is a function whose expression involves the same function. For example, the factorial function

$$n! = n \times (n-1) \times (n-2) \times ... \times 1, \text{ integer } n > 0$$

can be recursively defined as

For $n = 1$, $1! = 1$ (base case)
For $n > 1$, $n! = n \times (n-1)!$ (recursive definition)

Figure 4: Infinity Mirror

In computing, a recursive function or algorithm calls itself either directly or indirectly. Here is the pseudo code for a recursive `factorial` function.

Algorithm `factorial(n)`:
Input: Positive integer `n`
Output: Returns `n!`

1. If `n` is `1`, then return 1
2. Return `n` $\times$ `factorial(n-1)`

The "return" operation terminates a function and may also produce a value. Step 2 returns the value `n` times the value of `factorial(n-1)`, which is a call to the same function itself (Figure 5).

To solve a problem recursively, we reduce it to one or more smaller problems of the same nature. The smaller cases can then be solved by applying the same algorithm *recursively* until they become simple enough to solve directly.

Remember Recursion; *Think of recursion when solving problems. It can be a powerful tool.*

To appreciate the power of recursion and to see how it is applied to solve nontrivial problems, we will study several examples.

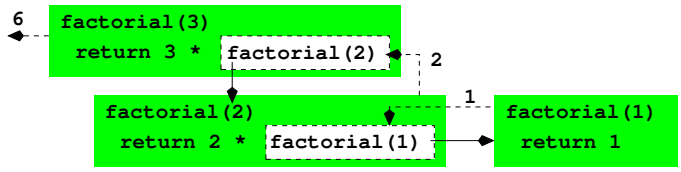

Figure 5: Recursive Calls and Returns

Greatest Common Divisor

Consider computing the *greatest common divisor* (GCD) of two non-negative integers a and b, not both zero. Recall that $gcd(a, b)$ is the largest integer that evenly divides both a and b. Mathematics gives $gcd(a, b)$ the recursion

1. If b is 0, then the answer is a
2. Otherwise, the answer is $gcd(b,\ a\ mod\ b)$

Recall that $a\ mod\ b$ is the remainder of a divided by b. Thus, a recursive algorithm for $gcd(a, b)$ can be written directly:

Algorithm `gcd(a,b)`:
Input: Non-negative integers `a` and `b`, not both zero
Output: The GCD of `a` and `b`

1. If `b` is zero, return `a`
2. Return `gcd(b, a mod b)`

Note, the algorithm `gcd` calls itself, and the value for `b` gets smaller for each successive call to `gcd` (Table 1). Eventually, the argument `b` becomes zero and the recursion unwinds: The deepest recursive call returns, then the next level call returns, and so on until the first call to `gcd` returns.

Recursive Solution Formula

The Recursion Magic: *Answer two simple questions and you may have magically solved a complicated problem.*

For many, recursion is a new way of thinking and brings a powerful tool for problem solving. To see if a recursive solution might be applicable to a given problem, you need to answer two questions:

Table 1: Recursion of `gcd(1265,440) = 55`

Call Level	`a`	`b`
1	1265	440
2	440	385
3	385	55
4	55	0

- Do I know a way to solve the problem in case the problem is small and trivial?
- If the problem is not small or trivial, can it be broken down into smaller problems of the same nature whose solutions combine into the solution of the original problem?

If the answer is yes to both questions, then you already have a recursive solution!

A recursive algorithm is usually specified as a function that calls itself directly or indirectly. Recursive functions are concise and easy to write once you recognize their basic structure. All recursive solutions use the following sequence of steps.

(i) Termination conditions: Always begin a recursive function with tests to catch the simple or trivial cases (the base cases) at the end of the recursion. A base case (array size zero for `quicksort` and remainder zero for `gcd`) is treated directly and the function returns.

(ii) Subproblems: Break the given problem into smaller problems of the same kind. Each is solved by a recursive call to the function itself passing arguments of reduced size or complexity.

(iii) Recombination of answers (optional): Finally, take the answers from the subproblems and combine them into the solution of the original bigger problem. The function call now returns. The combination may involve adding, multiplying, or other operations on the results from the recursive calls.

For problems, like the GCD and quicksort, where no recombination is necessary, this step becomes a trivial return statement. However, in the factorial solution, we need to multiply by n the result of the recursive call $factorial(n-1)$.

The *recursion engine* described here is deceptively simple. The algorithms look small and quite innocent, but the logic can be mind-boggling. To illustrate its power, we will consider the *Tower of Hanoi* puzzle.

Tower of Hanoi

Legend has it that monks in Hanoi spend their free time moving heavy gold disks to and from three poles made of black wood.

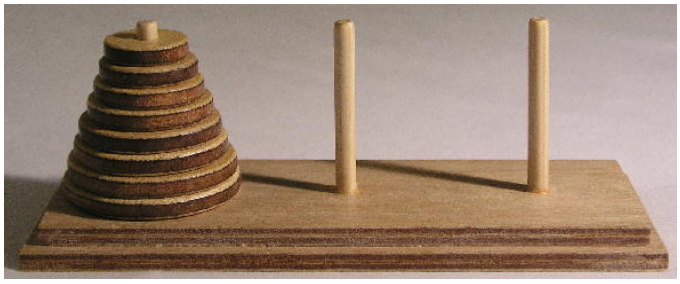

Figure 6: Tower of Hanoi Puzzle

The disks are all different in size and are numbered from 1 to n according to their sizes. Each disk has a hole at the center to fit the poles. In the beginning, all n disks are stacked on one pole in sequence, with disk 1, the smallest, on top, and disk n, the biggest, at the bottom (Figure 6). The task at hand is to move the disks one by one from the first pole to the third pole, using the middle pole as a resting place, if necessary. There are only three rules to follow:

1. A disk cannot be moved unless it is the top disk on a pole. Only one disk can be moved at a time.

2. A disk must be moved from one pole to another pole directly. It cannot be set down some place else.

3. At any time, a bigger disk cannot be placed on top of a smaller disk.

To simplify our discussion, let us label the first pole A (source pole), the second pole B (the parking pole), and the third pole C (the target pole). If you have not seen the solution before, you might like to try a small example first, say, $n = 3$. It does not take long to figure out the following sequence.

move disk 1 from A to C
move disk 2 from A to B
move disk 1 from C to B
move disk 3 from A to C
move disk 1 from B to A
move disk 2 from B to C
move disk 1 from A to C

So it turns out that you need 7 moves for the case $n = 3$. As you get a feel of how to do three disks, you are tempted to do four disks, and so on. But you will soon find that there seems to be no rule to follow, and the problem becomes much harder with each additional disk. Fortunately, the puzzle becomes very easy if you think about it recursively.

Let us apply our recursion engine to this puzzle in order to generate a sequence of correct moves for the problem: *Move n disks from pole A to C through B.*

(i) Termination condition: If $n = 1$, then move disk 1 from A to C and return.

(ii) Subproblems: For $n > 1$, we shall do three smaller problems:

1. Move $n - 1$ disks from A to B through C
2. Move disk n from A to C
3. Move $n - 1$ disks from B to C through A

There are two smaller subproblems of the same kind, plus a trivial step.

(iii) Recombination of answers: This problem is solved after the subproblems are solved. No recombination is necessary.

Let's write down the recursive function for the solution. Two recursive calls and a move of disk n is all that it takes.

Algorithm `hanoi(n, a, b, c)`:
Input: Integer n (the number of disks), `a` (name of source pole), `b` (name of parking pole), `c` (name of target pole)
Output: Displays a sequence of moves

1. If $n = 1$, display "Move disk 1 from a to c" and return.

2. `hanoi(n-1, a, c, b)`

3. Display "Move disk n from a to c"

4. `hanoi(n-1, b, a, c)`

Each of steps 2 and 4 makes a recursive call. This looks almost too simple, doesn't it? But it works. To obtain a solution for 7 disks, say, we make the call

```
hanoi(7, 'A', 'B', 'C')
```

During the course of the solution, different poles are used as the source, middle and target poles. This is the reason why the `hanoi` function has the `a, b, c` parameters in addition to `n`, the number of disks to be moved at any stage. Figure 7 shows the three-step recursive solution for `n = 5`:

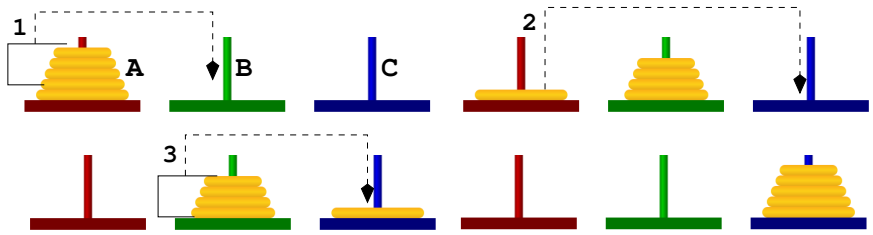

Figure 7: Tower of Hanoi (Five Disks)

1. Move 4 disks from pole `A` to pole `B` (`hanoi(4, 'A', 'C', 'B')`)

2. Move disk 5 from pole `A` to pole `C`

3. Move 4 disks from pole `B` to pole `C` (`hanoi(4, 'B', 'A', 'C')`)

Now, what is the number of moves needed for the `hanoi` algorithm for n disks? Let the move count be $mc(n)$. Then we have:

- If $n = 1$, then $mc(n) = 1$.
- If $n > 1$, then $mc(n) = 2 \times mc(n-1) + 1$

The above definition for $mc(n)$ is in the form of a *recurrence relation*, and it allows us to compute $mc(n)$ for any given n.

But, we can also seek a close-form formula for $mc(n)$.

$n = 1 \quad mc(1) = 1$
$n = 2 \quad mc(2) = 2 + 1$
$n = 3 \quad mc(3) = 2^2 + 2 + 1$
$n = 4 \quad mc(4) = 2^3 + 2^2 + 2 + 1$
...

Generally:

$$mc(n) = 2^{n-1} + 2^{n-2} + ... + 2 + 1 = 2^n - 1$$

An interactive Tower of Hanoi game (**Demo:** `Hanoi`) can be found at our companion website. Because $2^n - 1$ moves will be needed for n disks, you should test the program only with small values of n. But the monks in Hanoi are not so fortunate, they have 200 heavy gold disks to move, and the sun may burn out before they are finished!

But the logic behind the *recursion engine* provides a problem solving strategy unparalleled by other methods. Many seemingly complicated problems can be solved easily with recursive thinking.

More to Come

This article has given several important examples of problem solving using algorithmic thinking. The recursion paradigm is especially amazing and wonderful.

Computational thinkers can take a lot away from the discussions here in this article. However, problem solving is central to computational thinking and further discussion can be found in the article *Problem Solving: Backtracking & Heuristics*.