# Towers of Hanoi

The Towers of Hanoi puzzle is a classic problem, often used to demonstrate the power and simplicity of recursion. The game consists of three rods and a tower of discs in decreasing diameter. The goal is to move all the discs from one end rod to the other end rod using the middle rod as a helper. You can move one disc at a time and you can only place a disc of smaller diameter on top of a disc of larger diameter. Here's some background from Wikipedia:

> The puzzle was invented by the French mathematician Edouard Lucas in 1883. There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma, since that time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, when the last move of the puzzle will be completed, the world will end. It is not clear whether Lucas invented this legend or was inspired by it.

## Recursive Solution

Suppose we have $n$ discs, labelled $1 \dots n$ in increasing order by diameter. If we knew how to move the first $n-1$ discs to the middle rod (call it the *helper* rod), then we could move disc $n$ to to the target rod, and then use the same procedure to again move the $n-1$ discs from the middle rod to the target rod. This is exactly the recursive solution we hoped for. Here is one version that prints directions for the movement.

```
def hanoi(n, src, trgt, helper):
  if (n > 0):
    hanoi(n−1, src, helper, trgt)
    print("Moving disk {} from {} to {}".format(n,src,trgt))
    hanoi(n−1, helper, trgt, src)
```

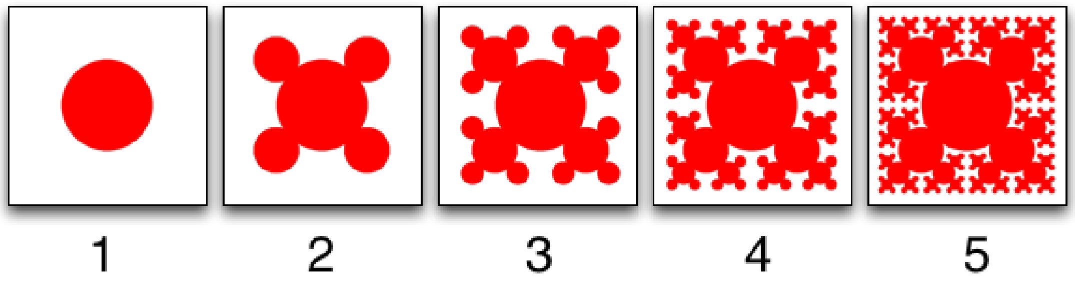
We might also be interested in solving the *end of the world* problem. Let's count how many *moves* are required to transfer the stack from one end to the other. We can do this by thinking about the `print` statement as a single move. Our solution just adds 1 to the number of moves it takes to transfer the first $n-1$ discs to the middle rod as well as the number of moves it takes to transfer them over to the target rod.

```
def hanoi_count(n, src, trgt, helper):
  if (n > 0):
    c1 = hanoi_count(n−1, src, helper, trgt)
    c2 = hanoi_count(n−1, helper, trgt, src)
    return c1 + 1 + c2
  else:
    return 0
```

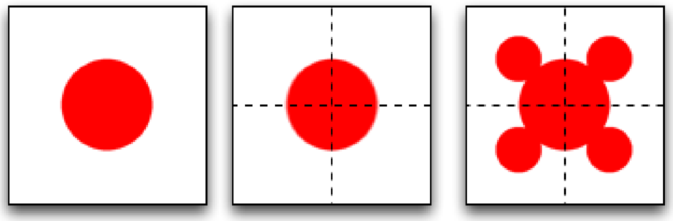
Notice that to move $n$ discs, it takes $2^n - 1$ moves. Suppose that it takes one second to move one disc. To move 64 discs would take $2^{64} - 1$ seconds, which is over 500 Billion years.

## Bubbles

Suppose you wanted to create one of the bubble images below for a given natural number $n$.



Limiting ourselves to the unit square where the bottom left is $(0, 0)$, we could generate a circle (in the form of a center point and radius) and then recursively generate four more circles. Each level of the recursion decreases $n$ by 1 until we reach $n = 0$, which means stop.



Here is a function to recursively compute all the circles (centers and radii) in Python and return them in a list.

```
def bubble(n, coords=((0, 0), (1, 1))):
  if n==0:
    return []
  else:
    (x0, y0), (x1, y1) = coords
    diam = (x1-x0)/2
    bubbles = [((x0+diam,y0+diam),diam/2)]
    bubbles.extend(bubble(n-1, ((x0, y0), (x0+diam, y0+diam))))
    bubbles.extend(bubble(n-1, ((x0, y0+diam), (x0+diam, y1))))
    bubbles.extend(bubble(n-1, ((x0+diam, y0), (x1, y0+diam))))
    bubbles.extend(bubble(n-1, ((x0+diam, y0+diam), (x1, y1))))
    return bubbles
```

Some comments:

- To write this function recursively, we need to know `n`, but we also need a rectangle to provide context. Thus, we also include optional `coords` parameters.
- The base case for our function is when `n==0`. Here we don't have any points, so we return an empty list.
- The inductive case computes the centroid of the square along with the diameter and then creates a list `bubbles` with that single pair. Now we assume that the four recursive calls correctly calculate all the smaller circles for the four quadrants, extend `bubbles` appropriately, and return the list.

Generating the image now requires iterating over the circles, interpolating the coordinates, and drawing them appropriately. Here is the Python code. The function takes a depth `n`, image dimensions `w` and `h`, and an output file name `output`. Notice that our interpolation function `inter` switches the $y$-coordinate appropriately. Also, the `ellipse` method assumes the first coordinate is the top-left corner of a bounding box and the second coordinate is the bottom-right corner of the bounding box.

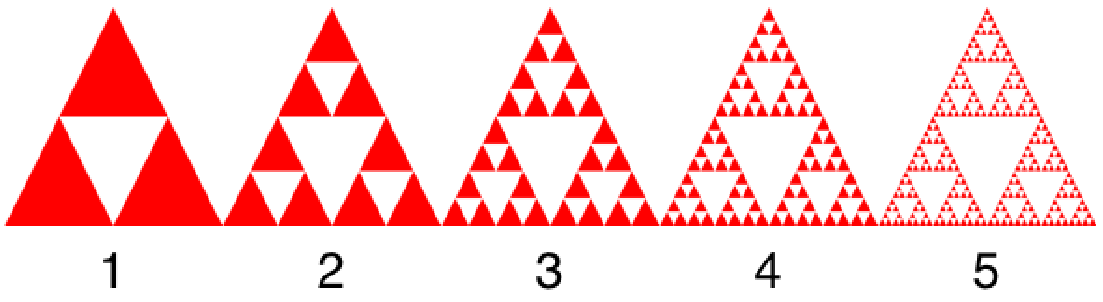
```python
def draw(n, w, h, output):

  def inter(x, y):
    return (int(x*w), int(h-y*h))

  im = Image.new("RGB", (w, h), (255, 255, 255))
  for (x,y), radius in bubble(n):
    coords = [inter(x-radius, y+radius), inter(x+radius,y-radius)]
    ImageDraw.Draw(im).ellipse(coords, fill=(255,0,0))
  im.save(output, "PNG")
```

## Sierpinksi Triangles

Below are five Sierpinksi Triangles. You should write a program called triangle.py that, when called from the command line with argument N, produces a single Sierpinksi Triangle of depth N. You should also pass it an output file name and an image dimension. Here is some example syntax for creating a triangle of depth 8.



```
from PIL import Image, ImageDraw
import sys


def triangles(n, x0=0.5, y0=1.0, x1=0.0, y1=0.0, x2=1.0, y2=0.0):

  if (n == 0):
    return []
  else:
    x3 = x1 + ((x0 − x1) / 2)
    y3 = y1 + ((y0 − y1) / 2)
    x4 = x0 + ((x2 − x0) / 2)
    y4 = y3
    x5 = x1 + ((x2 − x1) / 2)
    y5 = y1
    tris = [((x3, y3), (x4, y4), (x5, y5))]
    tris.extend(triangles(n − 1, x3, y3, x1, y1, x5, y5))
    tris.extend(triangles(n − 1, x4, y4, x5, y5, x2, y2))
    tris.extend(triangles(n − 1, x0, y0, x3, y3, x4, y4))
    return tris


def draw(n, w, h, output):

  def inter(x, y):
    return (x ∗ w, h − y ∗ h)

  im = Image.new(”RGB”, (w, h), (255, 255, 255))
  coords = [inter(0.5, 1), inter(0, 0), inter(1, 0)]
  ImageDraw.Draw(im).polygon(coords, fill=(255, 0, 0))
  for coords in triangles(n):
    coords = [inter(∗c) for c in coords]
    ImageDraw.Draw(im).polygon(coords, fill=(255, 255, 255))
  im.save(output, ”PNG”)


if __name__ == '__main__':
  draw(int(sys.argv[1]), int(sys.argv[3]), int(sys.argv[3]), sys.argv[2])
```