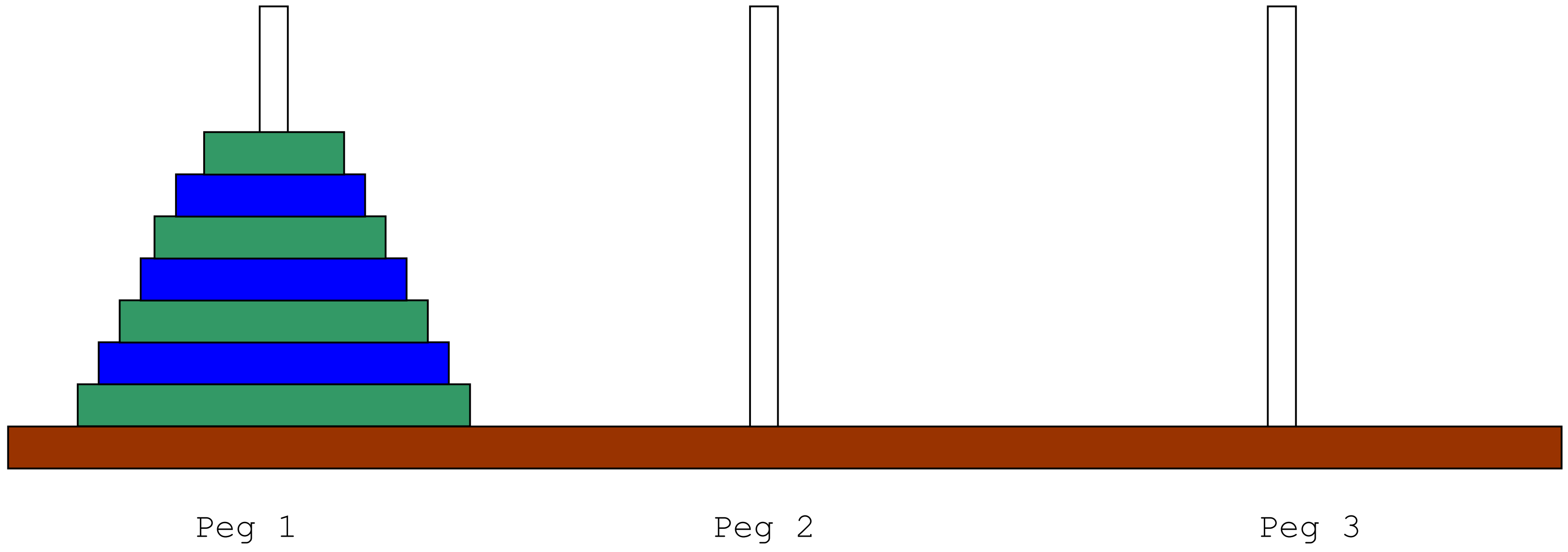
Recursion

Savitch chapter 13

Towers of Hanoi Puzzle

- This is a famous classic puzzle which involves having 3 pegs on which to store rings.
- The rings are originally all stacked on one peg, with each ring smaller than the one below it (think baby's toy!)
- The challenge is to move all the rings from one peg to an adjacent peg without ever putting a larger one on top of a smaller one
- And only moving one ring at a time
- There is one bit of help, a 3rd peg which can be used as an intermediate resting place for rings – but still never having a ring on top of one which is smaller than itself.
- Every budding computer scientist should be able to propose a programming solution which will print a precise sequence of peg to peg transfers (no matter how many rings there are in the stack)



Towers of Hanoi Puzzle – History!

- The "legend" which accompanied the game stated that in Benares, during the reign of the Emperor Fo Hi, there was a temple with a dome which marked the center of the world. Within the dome, priests moved golden disks between diamond needlepoints, a cubit high and as thick as the body of a bee. God placed 64 gold disks on one needle at the time of creation. It was said that when they completed their task, the universe would come to an end. (Since it would take at least 2^{64} -1 moves to complete the task, we're safe for now. Assuming one move per second, and no wrong moves, it would take almost 585,000,000,000 years to complete.)

Towers of Hanoi - 2

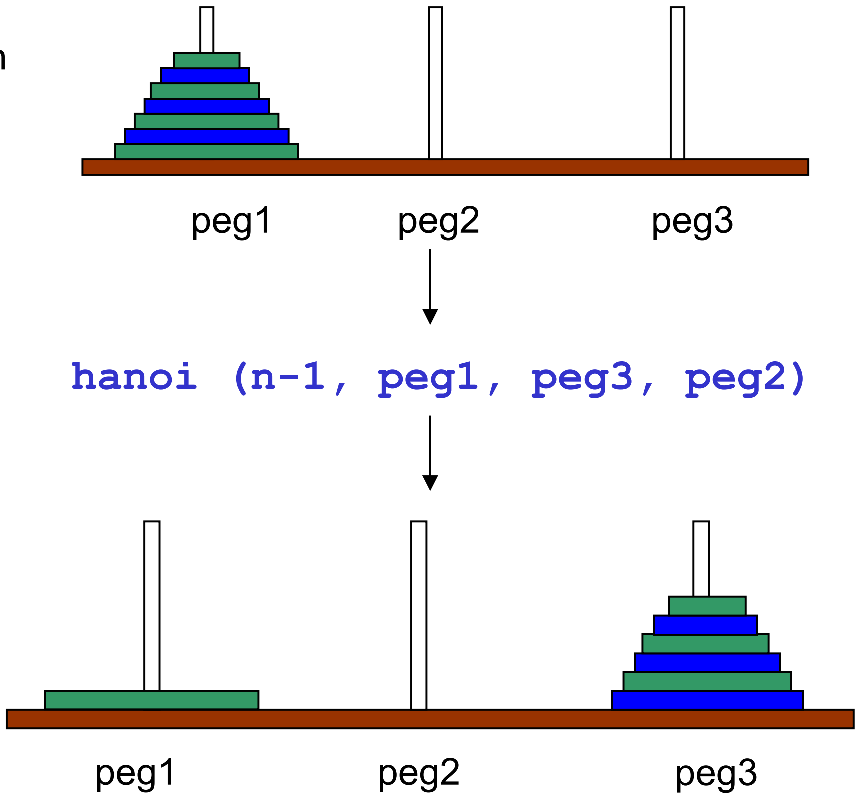
Lets say we want to code the solution in a function which is provided with arguments to tell it:

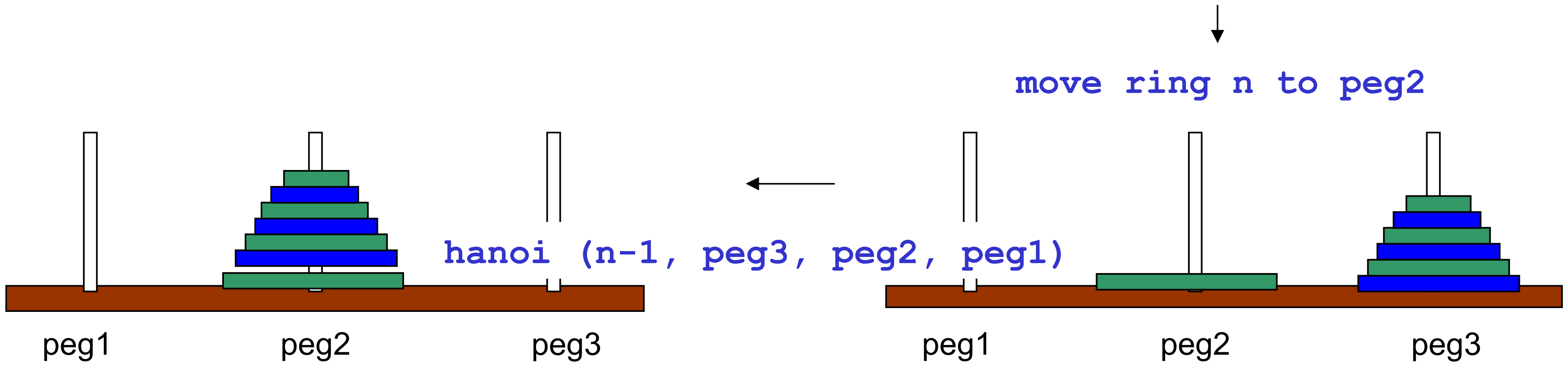
1. how many rings
2. the starting peg
3. the peg they are to finish on
4. the peg to use for temporary storage

hanoi(n, peg1, peg2, peg3)

Lets also assume that we ***already know how to do this for any smaller stacks of rings***

- Wow! Big assumption!!!
- So we can move n-1 rings to a spare peg only moving one ring at a time with the function call **hanoi (n-1, peg1, peg3, peg2)**
- Then move the bottom ring
- And finally move n-1 rings again to complete the solution





The recursive assumption

- The big assumption in this solution is that we can already solve the problem for simpler cases
  - This allows the recursive step
    "if we can solve this for 'the next simpler case', then we can solve it for this one"
  - So in our example:
    I can solve it for n if I can solve it for n-1
    And I can solve it for n-1 if I can solve it for n-2
    And I can solve it for n-2 if I can solve it for n-3
    And so on until we get right down to …
    And I can solve it for 2 if I can solve it for 1
  - And we can see that solving it for 1 ring would be trivial
  - so we can also solve it for 3, and therefore we can solve it for 4 , and so on right up to any number of rings n

- The recursive step can be used as long as we can show that the recursive assumption holds up – in other words there is a case so simple that we can solve it without another recursive step, and we will always reach that case

Towers of Hanoi – complete solution

```
hanoi (n, peg1, peg2, peg3)
BEGIN
    if (n == 1)
            print (move ring 1 to peg2)
    else
            hanoi(n-1, peg1, peg3, peg2)
            print (move ring n to peg2)
            hanoi(n-1, peg3, peg2, peg1)
    end else
END
```

We must have a really simple case that we are bound to come to

Keep calling simpler cases, which will call even simpler cases, which will call …

Recursion Discussion

- A function definition that includes a call to itself is said to be *recursive*.
- In general, using recursion is a bad idea.
  - It usually leads to infinite repetition of the code in the function (*infinite recursion*)
- But used with care, it can sometimes provide the possibility of simplifying the logic of a problem solution:
  1. Each call to the function MUST pass arguments that give a simpler version of the problem the function has to solve.
  2. There MUST be one or more versions that are so simple to solve that another recursive call will not be necessary. (*stopping cases*)
  3. Such a stopping case MUST *always* be reached.

- *Recursion should only be used in very clear situations, where it is plain that the problem is simplified with each call, and will definitely reach a base case.*

- Summary
  - A recursive function is called to solve a problem.
  - The function must actually know how to solve the simplest case(s) or so-called *base case(s)* of the problem,
  - it must always be possible to simplify the problem until a base case is reached.

Recursion – a (more mundane!) example

To calculate x to the power n, where n may be a large integer

$x^n = x * x * x * x * \ldots * x$ *(multiplied n times)*

We could use a for loop – start with x = 1, then multiply x by itself n times.

This is the iterative solution

- Or we could provide a recursive solution using the recursive step
  - $x^n = x * x^{n-1}$

  In other words, if we knew how to multiply x by itself n-1 times, then we could just multiply the result by x one more time.

  x^{n-1} is a slightly simpler problem than x^n
  - recursive call must be simpler
- x^1 is a really simple version – we would just return x

  x^0 is even simpler – we would just return 1
  - So these are base cases
- We can keep simplifying the problem until n is 1
  - Stopping case will always be reached : *as long as n started off positive* it will be reduced by one in each recursive call until it reaches 1

Working through our x^n example

- We will define the function as `pow(double x, int n)`
- If n is negative, return 0
  - to indicate problem can't be solved
- If this is a base case, return appropriately
  - If `n` is zero, return 1.
  - If `n` is 1, return `x`
- Otherwise return

  `x * pow(x, n -1)` (recursive step)

- `pow(x, 4)is  x * pow(x, 3)`
- `But pow(x, 3) is x * pow(x, 2)`
- `And pow(x, 2) is x * pow(x, 1)`
- `And pow(x, 1) is x`

$$\begin{tikzcd} & x * (x * (x * x)) \\ & (x * (x * x)) \arrow[u] \\ & (x * x) \arrow[u] \\ x \arrow[ur] & \end{tikzcd}$$

Now we can retrace our steps, passing back the result of each function call until we reach the initial function call

Recursion – more detail

- If the function is called with a base case, then the function simply returns a result.
- If the function is called with a more complex problem, the function divides the problem into two pieces
  - a piece that the function knows how to do
  - a piece that the function does not know how to do,
    - this latter piece must resemble the original problem, but must be a slightly simpler or smaller version of the original problem.
  - Because this new problem looks like the original problem, the function launches (calls) a fresh copy of itself to go work on the smaller problem, this is referred to as a recursive call and called the *recursive step*.
- The recursive step usually includes the keyword **return** because its result will be combined with the portion of the problem the function knew how to resolve to form a result that will be passed back to the original caller.
  - The recursive function call executes *while the original call to the function is still open*.
  - The recursive step can result in many more recursive calls being made as the function keeps dividing itself into simpler sub-problems until eventually the base case is met.
- When the base case is met, a result will be passed back up to the previous copy of the function and a sequence of returns follow to pass values up the line until the original call of the function eventually returns the final result to the calling function.

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Recursion Example 2: solving a palindrome

- A palindrome is a string of characters that read the same forward as backwards
    - ADA
    - ROTAVATOR
    - NEVER ODD OR EVEN *(OK, so we'd have to ignore white space!)*
- Write a recursive function which is passed a string, and figures whether it is a palindrome (return true or false) – you can use the following facts:
    - length of a string `str` is `str.length()`
    - Substring of length len starting at index pos i is `str.substr(i, len)`
    - First character is `str[0]`
    - last character is `str[str.length() - 1]`
    - String of length 1 is always a palindrome
    - String of length 2 is a palindrome if the 2 characters are identical

Recursion – real example

- Important problems often solved recursively include the binary search
  - Recall, the search is simplified at each step by cutting the array to search in half and determining if the top half or bottom half is to be searched.
- However it is not a method recommended if performance is important
  - Iterative versions run more efficiently, but recursive versions are often easier to understand
- Sometimes programmers start with a recursive algorithm, and then convert it to an iterative version.

Recap on Recursion

- What is recursion?
  - Calling a function from inside the same function
- Why is it usually a bad idea?
  - Leads to infinite repetition of the function code
- Why is it sometimes a good idea?
  - Sometimes makes the logic of a code easier to understand
- What are the questions to ask before using a recursive solution?
  - Is there a way to divide the problem to be solved by the function into a straightforward bit combined with a simpler version of the same problem?
  - Is there some case so simple that the solution is completely straightforward (don't need to combine with a simpler version of the same problem)
  - Can we *guarantee* that such a simple case will definitely be reached?

Practice

1. Try some of the self test exercises in Savitch chapter 13
2. Try writing a function that will test if a string passed to it is a palindrome. (see earlier overhead)