Recursion

Genome 559: Introduction to Statistical and Computational Genomics

Elhanan Borenstein

The merge sort algorithm

1. *Split your list into two halves*
2. *Sort the first half*
3. *Sort the second half*
4. *Merge the two sorted halves, maintaining a sorted order*

Divide-and-conquer

- The basic idea behind the merge sort algorithm is to divide the original problem into two halves, **each being a smaller version of the original problem**.

- This approach is known as ***divide and conquer***
  - Top-down technique
  - Divide the problem into **independent** smaller problems
  - Solve smaller problems
  - Combine smaller results into a larger result thereby “conquering” the original problem.

Merge sort – the nitty gritty

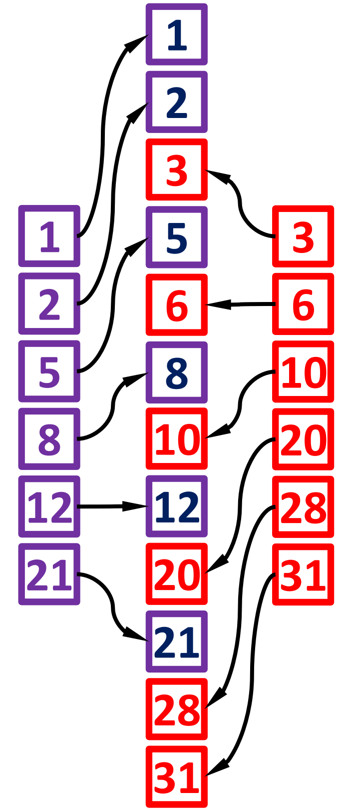
The merge sort algorithm

1. *Split your list into two halves*
2. *Sort the first half*
3. *Sort the second half*
4. *Merge the two sorted halves, maintaining a sorted order*

That's simple

???

Careful bookkeeping, but still simple



If I knew how to sort, I wouldn't be here in the first place?!?

Merge sort – the nitty gritty

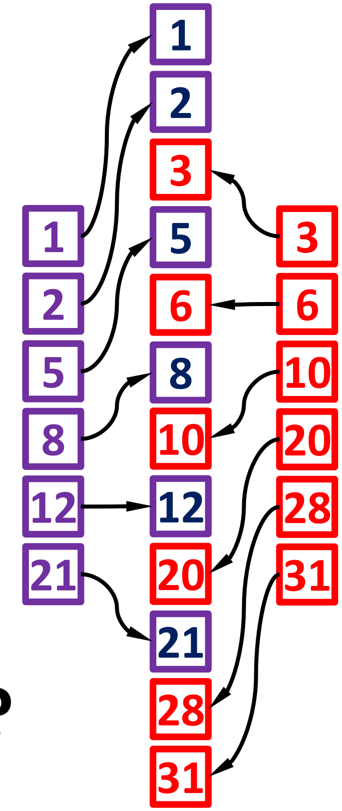
The merge sort algorithm

1. *Split your list into two halves*
2. *Sort the first half*
3. *Sort the second half*
4. *Merge the two sorted halves, maintaining a sorted order*

That's simple

???

Careful bookkeeping, but still simple



So …

how are we going to sort the two smaller lists?

Here's a crazy idea:

let's use merge sort

to do this

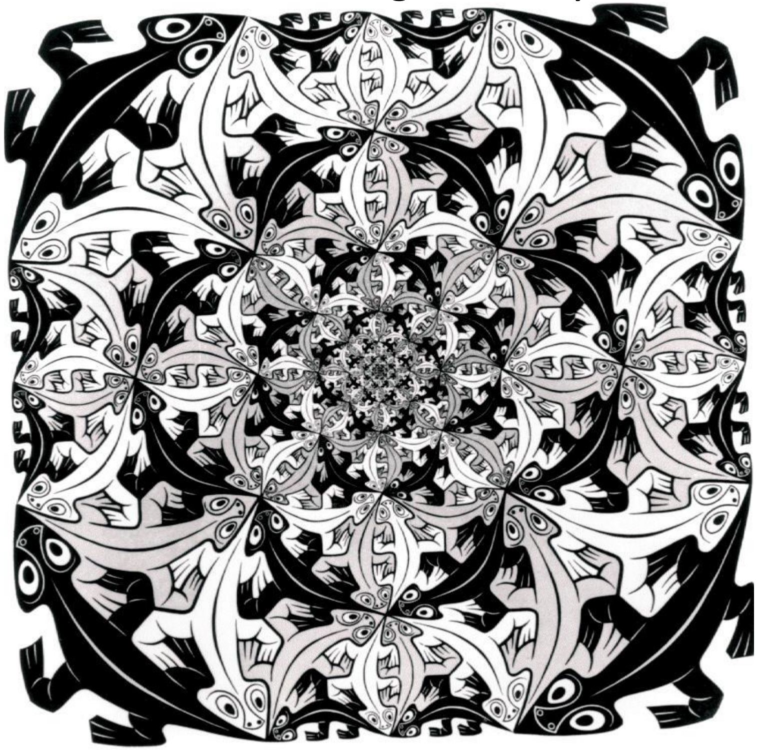
```
def mergeSort(list):
    half1 ← first half of list
    half2 ← second half of list
    half1_sorted = mergeSort(half1)
    half2_sorted = mergeSort(half2)
    list_sorted = merge(half1_sorted,half2_sorted)
    return list_sorted
```

You must be kidding, right?

- WHAT?
- This function has no loop?
- It seems to refer to itself!
- Where is the actual sort?
- What's going on???

```
def mergeSort(list):
    half1 ← first half of list
    half2 ← second half of list
    half1_sorted = mergeSort(half1)
    half2_sorted = mergeSort(half2)
    list_sorted = merge(half1_sorted,half2_sorted)
    return list_sorted
```

this is making me dizzy!



Let’s take a step back …

Factorial

- A simple function that calculates n!

```
# This function calculated n!
def factorial(n):
    f = 1
    for i in range(1,n+1):
        f *= i
    return f
```

```
>>> print factorial(5)
120
>>> print factorial(12)
479001600
```

- This code is based on the standard definition of factorial: $n! = \prod_{k=1}^{n} k$

Factorial

- But … there is an alternative **recursive** definition:

$$n! = \begin{cases} 1 & if \quad n = 0 \\ (n-1)! \times n & if \quad n > 0 \end{cases}$$

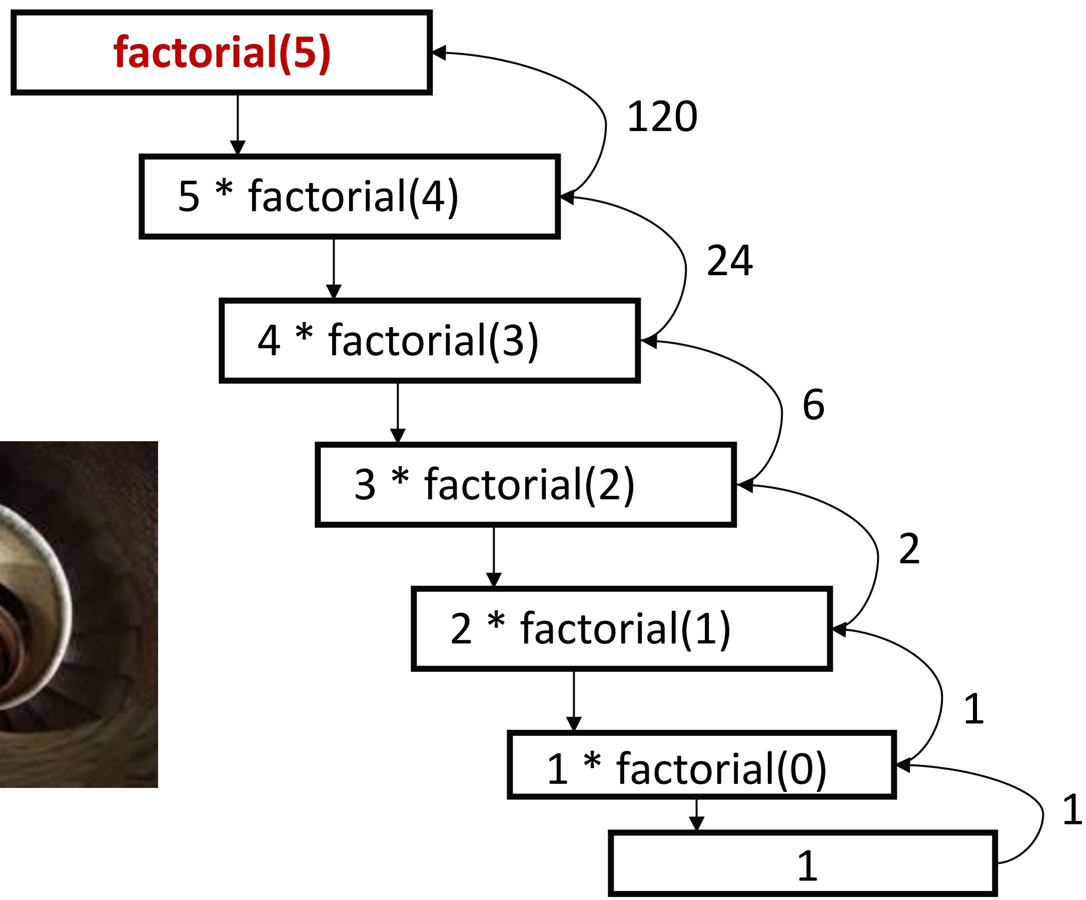
- So … can we write a function that calculates n! using this approach?

```
# This function calculated n!
def factorial(n):
    if n==0:
        return 1
    else:
        return n * factorial(n-1)
```

- **Well …**
  We can! It works! And it is called a *recursive* function!

Why is it working?

```
# This function calculated n!
def factorial(n):
    if n==0:
        return 1
    else:
        return n * factorial(n-1)
```



Recursion and recursive functions

- **A function that calls itself**, is said to be a **recursive** function (and more generally, an algorithm that is defined in terms of itself is said to use recursion or be recursive)

(A call to the function "recurs" within the function; hence the term "recursion")

- In may real-life problems, recursion provides an intuitive and natural way of thinking about a solution and can often lead to very elegant algorithms.

mmm…

- If a recursive function calls itself in order to solve the problem, isn't it circular?
  (in other words, why doesn't this result in an infinite loop?)

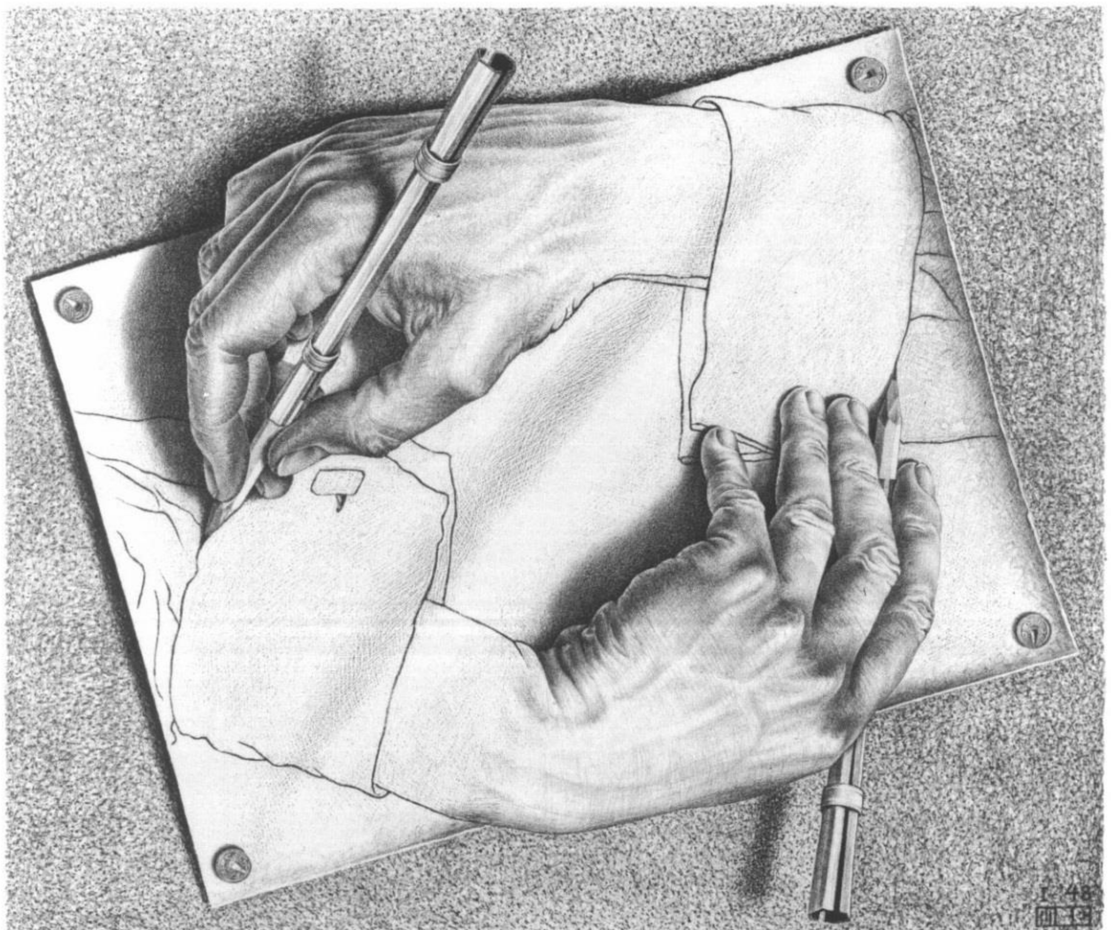
- Factorial, for example, is not circular because we eventually get to 0!, whose definition **does not rely** on the definition of another factorial and is simply 1.
  - This is called a <u>**base case**</u> for the recursion.
  - When the base case is encountered, we get a closed expression that can be directly computed.

Defining a recursion

- Every recursive algorithm must have two key features:
  1. There are one or more ***base cases*** for which no recursion is applied.
  2. All recursion chains eventually end up at one of the base cases.

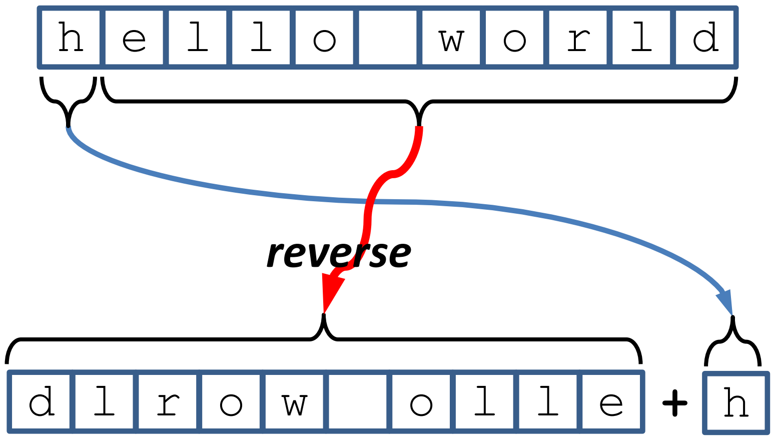
The simplest way for these two conditions to occur is for each recursion to act on a ***smaller*** *version of the original problem. A very small version of the original problem that can be solved without recursion then becomes the base case.*

This is fun!
Let’s try to solve (or at least think of) other problems using recursion



String reversal

- Divide the string into ***first*** *character* and ***all the rest***
- Reverse the "rest" and append the first character to the end of it



See how simple and elegant it is! No loops!

```
# This function reverses a string
def reverse(s):
    return reverse(s[1:]) + s[0]
```

s[1:] returns all but the first character of the string. We ***reverse*** *this part (s[1:]) and then concatenate the first character (s[0]) to the* ***end****.*

String reversal - D’oh!

```
# This function reverses a string
def reverse(s):
    return reverse(s[1:]) + s[0]
```

```
>>> print reverse(“hello world”)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in reverse
  File "<stdin>", line 2, in reverse
  File "<stdin>", line 2, in reverse
  File "<stdin>", line 2, in reverse
  File "<stdin>", line 2, in reverse
  File "<stdin>", line 2, in reverse
  .
  .
  .
  File "<stdin>", line 2, in reverse
  File "<stdin>", line 2, in reverse
  File "<stdin>", line 2, in reverse
  File "<stdin>", line 2, in reverse
  File "<stdin>", line 2, in reverse
RuntimeError: maximum recursion depth exceeded
```

What just happened? There are 1000 lines of errors!

String reversal – Duh!

- **Remember**: To build a correct recursive function, we need a **base case** that doesn't use recursion!

  We forgot to include a base case, so our program is an infinite recursion. Each call to "reverse" contains another call to reverse, so none of them return.

  Each time a function is called it takes some memory. Python stops it at 1000 calls, the default "maximum recursion depth."

- **What should we use for our base case?**

String reversal - Yeah

- Since our algorithm is creating shorter and shorter strings, it will eventually reach a stage when *S* is of length 1 (one character).
- Since a string of length 1 is its own reverse, we can use it as the base case.

```
# This function reverses a string
def reverse(s):
    if len(s) == 1:
        return s
    else:
        return reverse(s[1:])+s[0]
```

```
>>> print reverse("hello world")
"dlrow olleh"
```

Search a sorted list

(think phonebook)

- How would you search a sorted list to check whether a certain item appears in the list and where?
  - Random search *(yep, I know, this is a stupid algorithm)*
  - Serial search – *O(n)*
  - Binary search

hunting a lion in the desert

Binary search

<u>The binary-search algorithm</u>

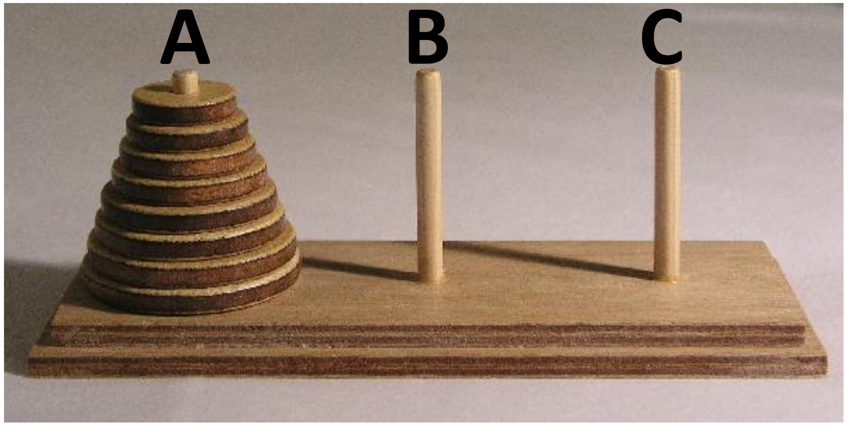
1. *If your list is of size 0, return "not-found".*
2. *Check the item located in the middle of your list.*
3. *If this item is **equal** to the item you are looking for:*
   ***you're done!** Return "found".*
4. *If this item is **bigger** than the item you are looking for:*
   *do a <u>binary-search</u> on the **first half** of the list.*
5. *If this item is **smaller** than the item you are looking for:*
   *do a <u>binary-search</u> on the **second half** of the list.*

How long does it take for this algorithm to find the query item (or to determine it is not in the list)?

Towers of Hanoi

- There are three posts and 64 concentric disks shaped like a pyramid.
- The goal is to move the disks from post ***A*** to post ***B***, following these three rules:

  1. You can move only one disk at a time.
  2. A disk may not be "set aside". It may only be stacked on one of the three posts.
  3. A larger disk may never be placed on top of a smaller one.

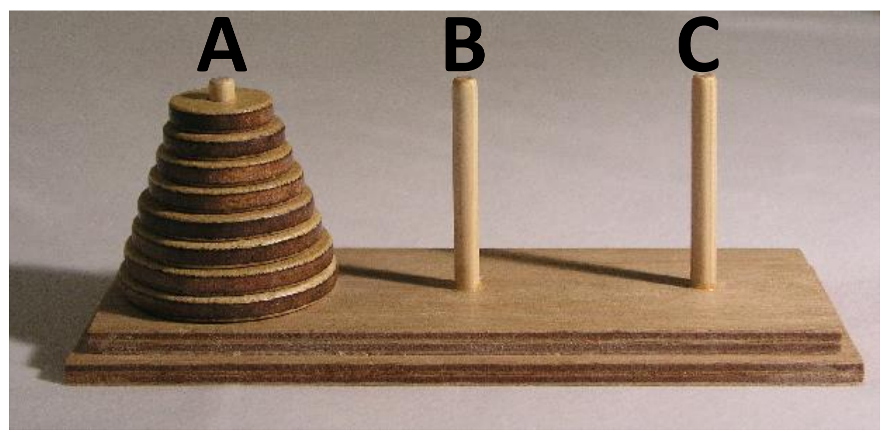


Towers of Hanoi

<u>Towers-of-Hanoi algorithm (for an “n disk tower”)</u>

1. *Move an “n-1 disk tower” from source-post to resting-post (use the tower-of-hanoi algorithm)*
2. *Move 1 disk from source-post to destination-post*
3. *Move an “n-1 disk tower” from resting-post to destination-post (use the tower-of-hanoi algorithm)*

- **What should the base case be?**
- Assuming each disk move takes 1 second, how long would it take to move a 64 disk tower?



Finally,
let’s get back to our merge sort

The merge sort algorithm

1. *Split your list into two halves*
2. *Sort the first half (**using merge sort**)*
3. *Sort the second half (**using merge sort**)*
4. *Merge the two sorted halves, maintaining a sorted order*

The merge sort algorithm

1. *Split your list into two halves*
2. *Sort the first half (**using merge sort**)*
3. *Sort the second half (**using merge sort**)*
4. *Merge the two sorted halves, maintaining a sorted order*

4 helper function

```
# Merge two sorted lists
def merge(list1, list2):
    merged_list = []
    i1 = 0
    i2 = 0

    # Merge
    while i1 < len(list1) and i2 < len(list2):
        if list1[i1] <= list2[i2]:
            merged_list.append(list1[ii])
            i1 += 1
        else:
            merged_list.append(list2[i2])
            i2 += 1

    # One list is done, move what's left
    while i1 < len(list1):
        merged_list.append(list1[i1])
        i1 += 1
    while i2 < len(list2):
        merged_list.append(list2[i2])
        i2 += 1

    return merged_list
```

```
# merge sort recursive
def sort_r(list):
    if len(list) > 1: # Still need to sort
```

1

```
        half_point = len(list)/2
        first_half = list[:half_point]
        second_half = list[half_point:]
```

2

```
        first_half_sorted = sort_r(first_half)
```

3

```
        second_half_sorted = sort_r(second_half)
```

4

```
        sorted_list = merge \
            (first_half_sorted, second_half_sorted)
        return sorted_list
    else:
        return list
```

List of size 1.
Base case

Recursion vs. Iteration

- There are usually similarities between an iterative solutions (e.g., looping) and a recursive solution.
  - In fact, anything that can be done with a loop can be done with a simple recursive function!
  - In many cases, a recursive solution can be easily converted into an iterative solution using a loop (but not always).

- Recursion can be very costly!
  - Calling a function entails overhead
  - Overhead can be high when function calls are numerous (stack overflow)

Recursion - the take home message

- **Recursion is a great tool to have in your problem-solving toolbox.**
- In many cases, recursion provides a natural and elegant solution to complex problems.
- If the recursive version and the loop version are similar, prefer the loop version to avoid overhead.
- Yet, even in these cases, recursion offers a creative way to **think** about how a problem could be solved.

Sample problem #1

- Write a function that calculates the sum of the elements in a list using a recursion

  Hint: your code should not include ANY for-loop or while-loop!

- Put your function in a module, import it into another code file and use it to sum the elements of some list.

Solution #1

utils.py

```
def sum_recursive(a_list):
    if len(a_list) == 1:
        return a_list[0]
    else:
        return a_list[0] + sum_recursive(a_list[1:])
```

my_prog.py

```
my_list = [1, 3, 5, 7, 9, 11]

from utils import sum_recursive
print sum_recursive(my_list)
```

Sample problem #2

- Write a **recursive** function that determines whether a string is a palindrome. Again, make sure your code does not include any loops.

  A palindrome is a word or a sequence that can be read the same way in either direction.

  For example:

  - *“d*etartrated”
  - “olson in oslo*”*
  - *“s*tep on no pets”

Solution #2

```
def is_palindrome(word):
    l = len(word)
    if l <= 1:
        return True
    else:
        return word[0] == word[l-1] and is_palindrome(word[1:l-1])
```

```
>>>is_palindrome("step on no pets")
True
>>>is_palindrome("step on no dogs")
False
>>>is_palindrome("12345678987654321")
True
>>>is_palindrome("1234")
False
```

Challenge problems

1. **Write a recursive function that prime factorize s an integer number.**
   (The prime factors of an integer are the prime numbers that divide the integer exactly, without leaving a remainder).
   Your function should print the list of prime factors:

```
>>> prime_factorize(5624)
2 2 2 19 37
>>> prime_factorize(277147332)
2 2 3 3 3 3 3 7 7 11 23 23
```

   Note: you can use a for loop to find a divisor of a number but the factorization process itself should be recursive!

2. Improve your function so that it “returns” a list containing the prime factors. Use pass-by-reference to return the list.

3. Can you do it without using ANY loops whatsoever?

Challenge solution 1

```
import math

def prime_factorize(number):

    # find the first divisor
    divisor = number
    for i in range(2,int(math.sqrt(number))+1):
        if number % i == 0:
            divisor = i
            break

    print divisor,

    if divisor == number: # number is prime. nothing more to do
        return
    else:                 # We found another divisor, continue
        prime_factorize(number/divisor)

prime_factorize(277147332)
```

Challenge solution 2

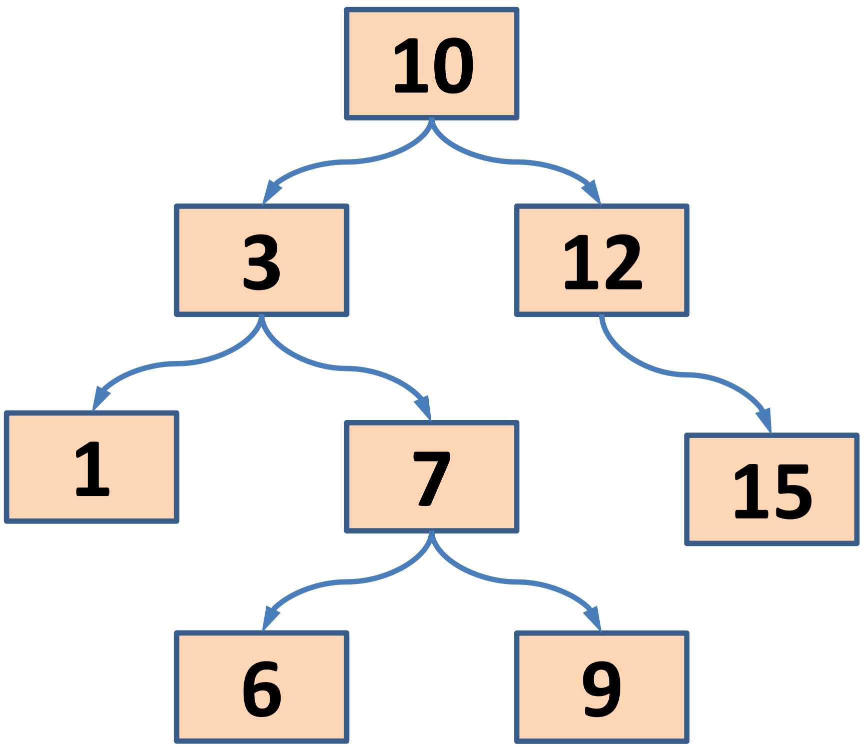
```
import math

def prime_factorize(number, factors=[]):

    # find the first divisor
    divisor = number
    for i in range(2,int(math.sqrt(number))+1):
        if number % i == 0:
            divisor = i
            break

    factors.append(divisor)

    if divisor == number: # number is prime. nothing more to do
        return
    else:                 # We found another divisor, continue
        prime_factorize(number/divisor, factors)


factors = []
prime_factorize(277147332,factors)
print factors
```

Traversing a phylogenetic tree

- Recursion is extremely useful when processing a data structure that is recursive by nature.

```
def preorder(node):
    if node == None:
        return

    print node.value,
    preorder(node.left)
    preorder(node.right)
```

```
preorder(root)
10 3 1 7 6 9 12 15
```



Traversing a phylogenetic tree

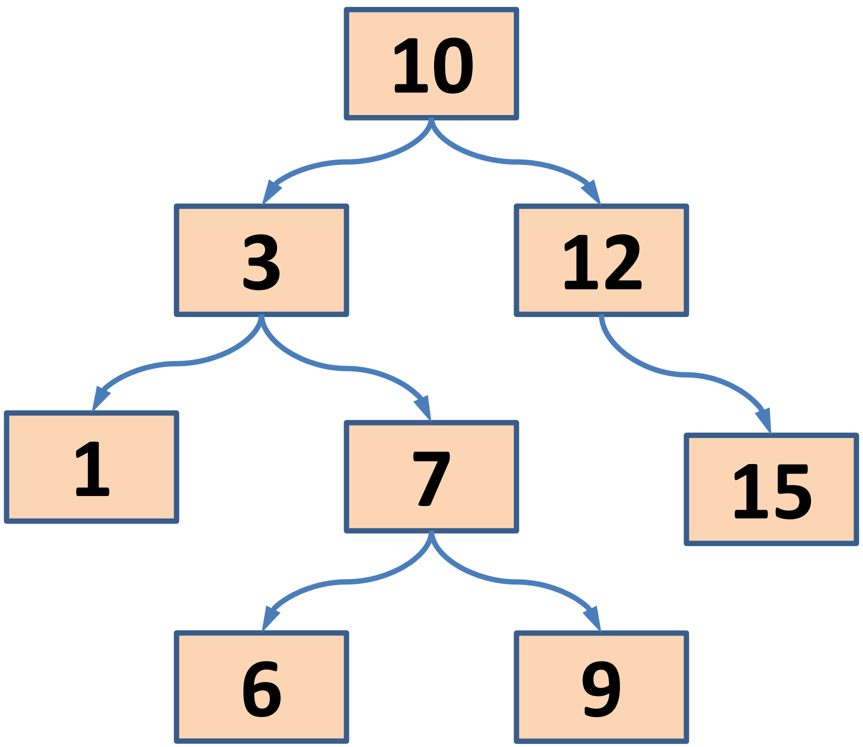
- Recursion is extremely useful when processing a data structure that is recursive by nature.

```
def preorder(node):
    if node == None:
        return

    print node.value,
    preorder(node.left)
    preorder(node.right)
```

```
preorder(root)
10 3 1 7 6 9 12 15
```

```
def postorder(node):
    if node == None:
        return

    postorder(node.left)
    postorder(node.right)
    print node.value,
```

```
postorder(root)
1 6 9 7 3 15 12 10
```



Recursion vs. Iteration

- Check the following two solutions for the Fibonacci sequence:

```
# Returns the n-th Fibonacci number
def fib_loop(n):
    prev = 0
    curr = 1
    for i in range(n-2):
        curr, prev = curr+prev, curr
    return curr
```

```
# Returns the n-th Fibonacci number
def fib_rec(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fib_rec(n-1)+fib_rec(n-2)
```

Which one would you prefer?