COMP1819
Algorithms and Data Structures

Lecture 09: Recursive algorithms and analysis

Dr. Tuan Vuong

16/03/2021

Content

- Lab 08 Walk-through
- Recursive algorithms & Analysis
- Linear and Binary recursion
- Visualising recursion
- Reinforcement
- CW Q&A

Walk-through

Note: please follow the instructions in the coursework instruction in Moodle. Missing programming exercises, here we go.

---

5. Palindromes

Write a program to read a text file and list out all palindrome words in the file but removing the duplications. A palindrome is a word or sentence which is read the same backwards as it is forwards, such as the name, "Hannah", a word “civic” or the sentence, "Never odd or even"

Just go through the process…

Today

https://www.amazon.co.uk

Three laws of recursion

- A recursive algorithm must
  - Have a base case.
  - Change its state and move forward the base case.
  - Call itself, recursively.

The Recursion Pattern

- **Recursion**: when a method calls itself
- Classic example--the factorial function:
  - n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \textit{else} \end{cases}$$

- As a Python method:

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

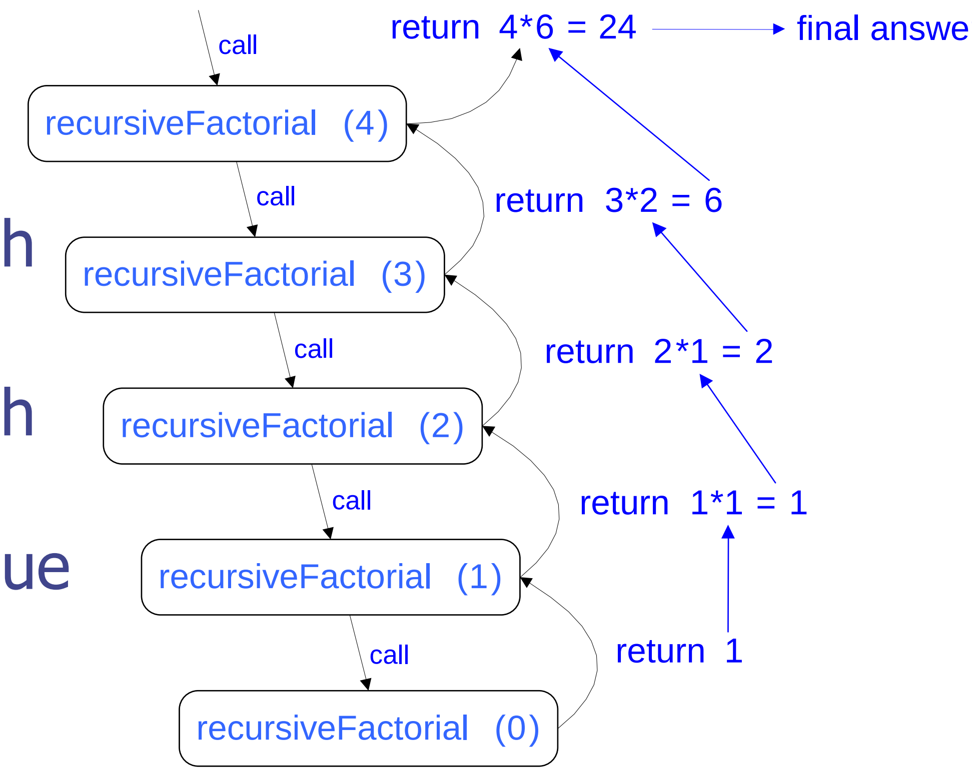
Content of a Recursive Method

- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Visualizing Recursion

- Recursion trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value

- Example



Analyzing computing factorials

- Runs in O(n) time.
  - n + 1 activations, from n, n-1, …,0 (base).
  - Each activation executes a constant number of operations.

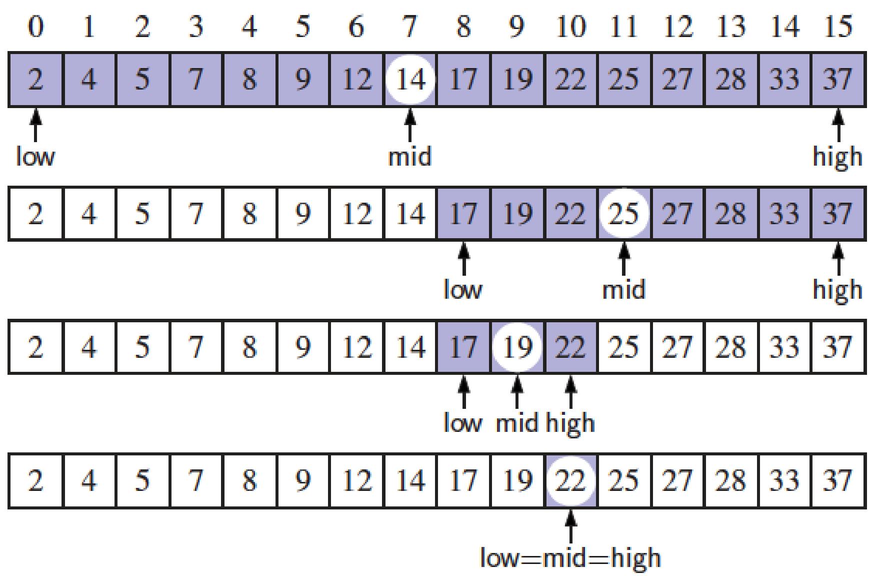
Binary Search

- Search for an integer, target, in an ordered list.

```python
def binary_search(data, target, low, high):
  """Return True if target is found in indicated portion of a Python list.

  The search only considers the portion from data[low] to data[high] inclusive.
  """
  if low > high:
    return False                      # interval is empty; no match
  else:
    mid = (low + high) // 2
    if target == data[mid]:           # found a match
      return True
    elif target < data[mid]:
      # recur on the portion left of the middle
      return binary_search(data, target, low, mid - 1)
    else:
      # recur on the portion right of the middle
      return binary_search(data, target, mid + 1, high)
```

Visualizing Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



Analyzing Binary Search

- Runs in O(log n) time.
  - The remaining portion of the list is of size high – low + 1.
  - After one comparison, this becomes one of the following:

$$(\mathsf{mid}-1)-\mathsf{low}+1=\left\lfloor\frac{\mathsf{low}+\mathsf{high}}{2}\right\rfloor-\mathsf{low}\leq\frac{\mathsf{high}-\mathsf{low}+1}{2}$$

$$\mathsf{high}-(\mathsf{mid}+1)+1=\mathsf{high}-\left\lfloor\frac{\mathsf{low}+\mathsf{high}}{2}\right\rfloor\leq\frac{\mathsf{high}-\mathsf{low}+1}{2}.$$

  - Thus, each recursive call divides the search region in half; hence, there can be at most log n levels.

Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

Example of Linear Recursion

Algorithm LinearSum(A, n):

Input:

A integer array A and an integer n = 1, such that A has at least n elements

Output:

The sum of the first n integers in A

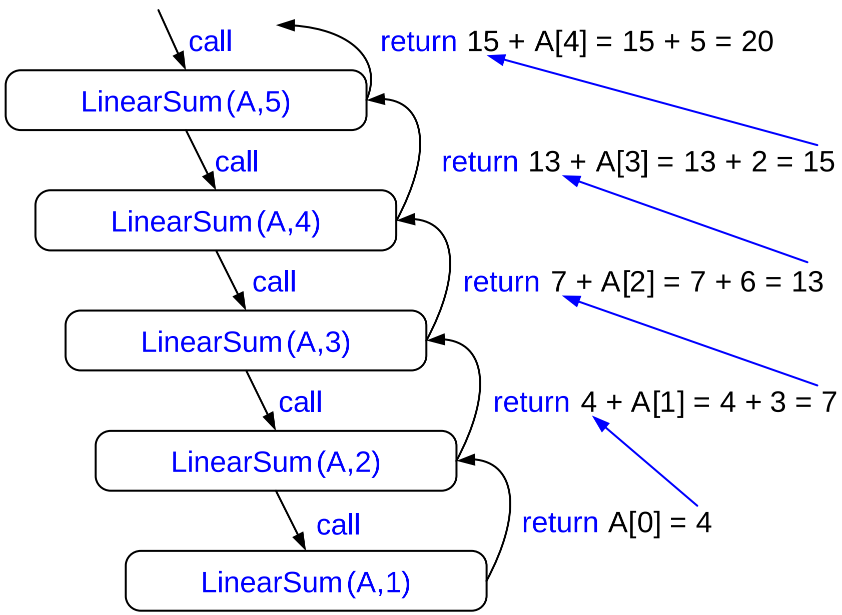
if $n = 1$ **then**

  return $A[0]$

else

  return LinearSum(A, n - 1) + $A[n - 1]$

Example recursion trace:



Reversing an Array

Algorithm ReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

if $i < j$ **then**

Swap $A[i]$ and $A[j]$

ReverseArray(A, $i + 1$, $j - 1$)

return

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional paramaters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(A, i, j), not ReverseArray(A).
- Python version:

```
def reverse(S, start, stop):
  """Reverse elements in implicit slice S[start:stop]."""
  if start < stop − 1:                          # if at least 2 elements:
    S[start], S[stop−1] = S[stop−1], S[start]   # swap first and last
    reverse(S, start+1, stop−1)                 # recur on rest
```

Binary Recursion

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.

Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1.$$

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):
 Input: Nonnegative integer k
 Output: The kth Fibonacci number F_k
 if $k <= 1$ **then**
  return ***k***
 else
  return BinaryFib(k - 1) + BinaryFib(k - 2)

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

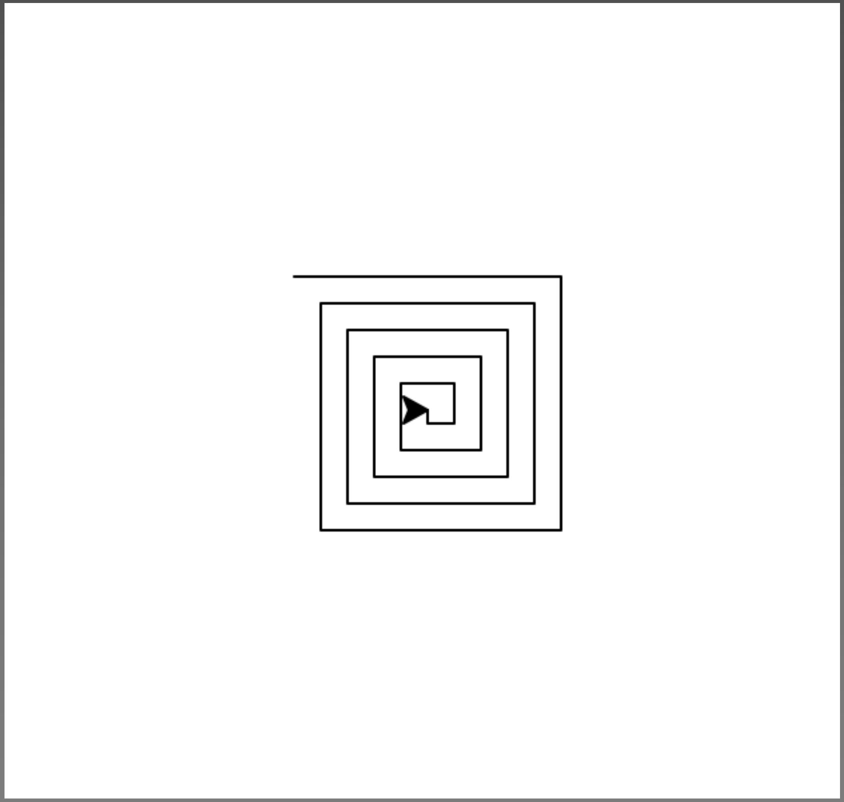
A Better Fibonacci Algorithm

- Use linear recursion instead

Algorithm LinearFibonacci(k):
  Input: A nonnegative integer k
  Output: Pair of Fibonacci numbers (F_k, F_{k-1})
  if $k = 1$ **then**
      return (1, 0)
  else
      (i, j) = LinearFibonacci($k - 1$)
      return (i +j, i)

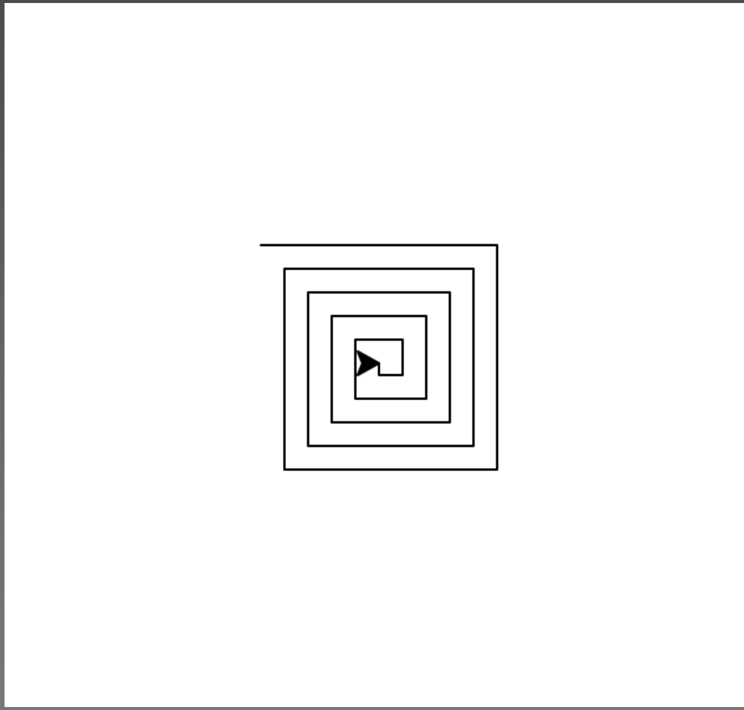
- LinearFibonacci makes k−1 recursive calls
- Hence, instead of exponential, it is linear.

Introduction: Visualizing Recursion



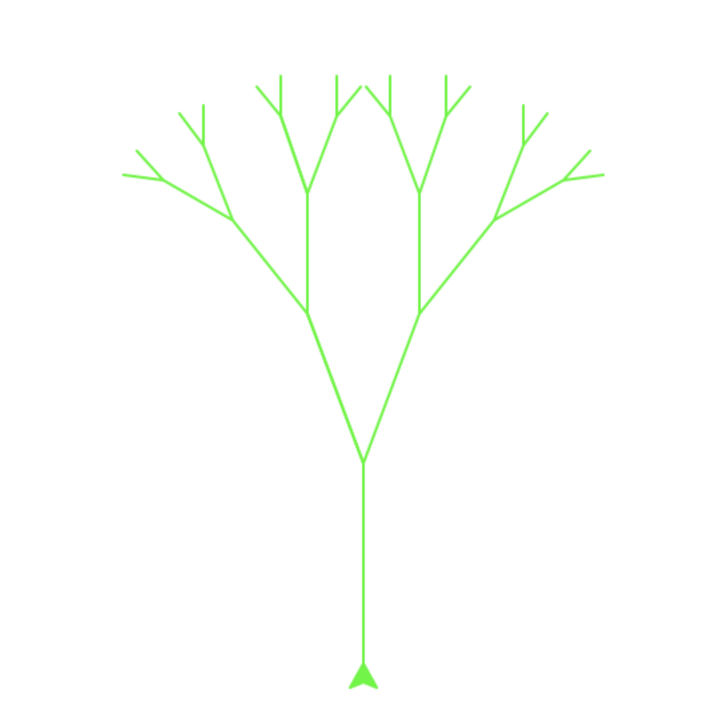
Drawing Spiral

```
import turtle

myTurtle = turtle.Turtle()
myWin = turtle.Screen()

def drawSpiral(myTurtle, lineLen):
    if lineLen > 0:
        myTurtle.forward(lineLen)
        myTurtle.right(90)
        drawSpiral(myTurtle,lineLen-5)

drawSpiral(myTurtle,100)
myWin.exitonclick()
```

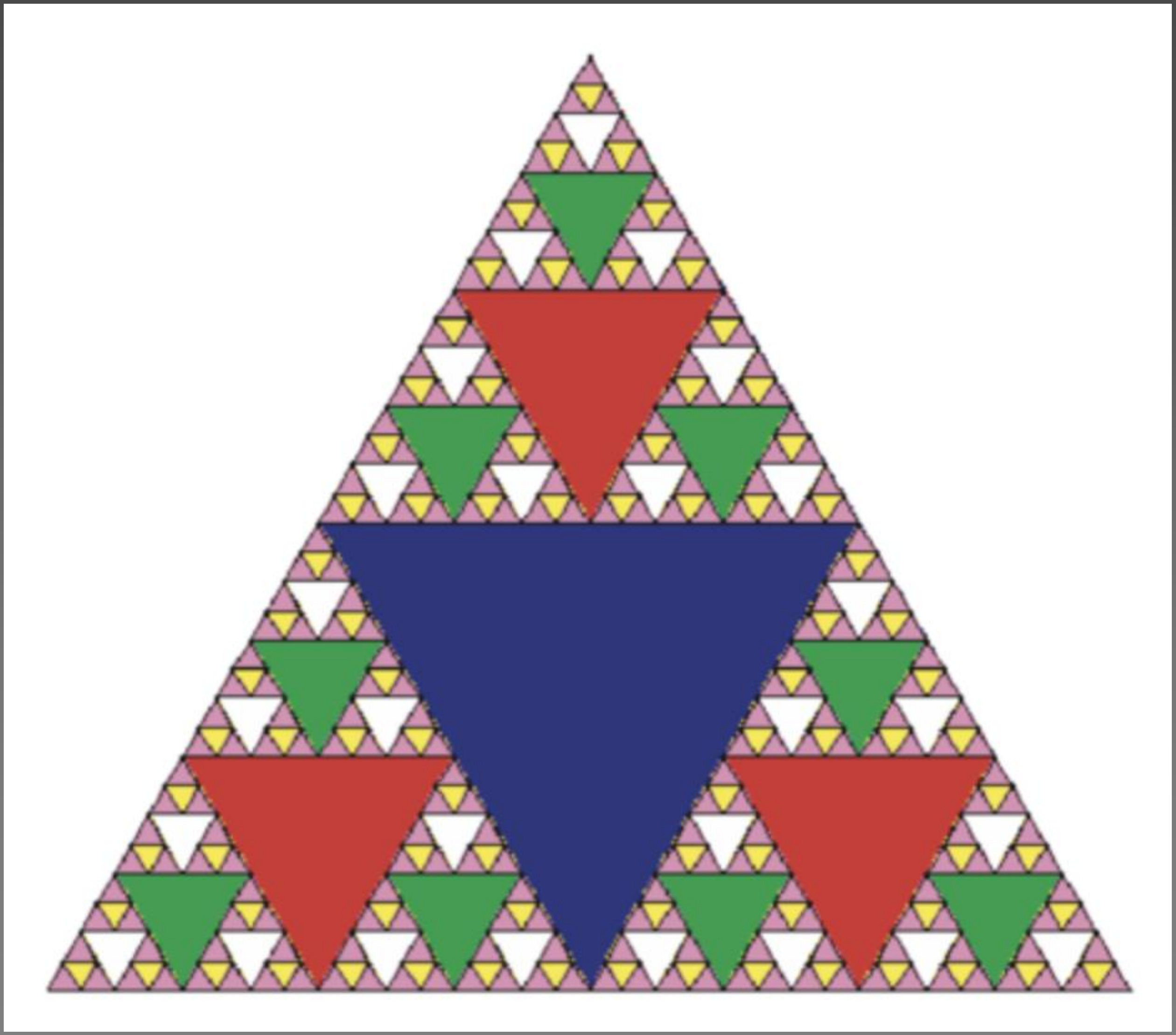


Drawing Tree

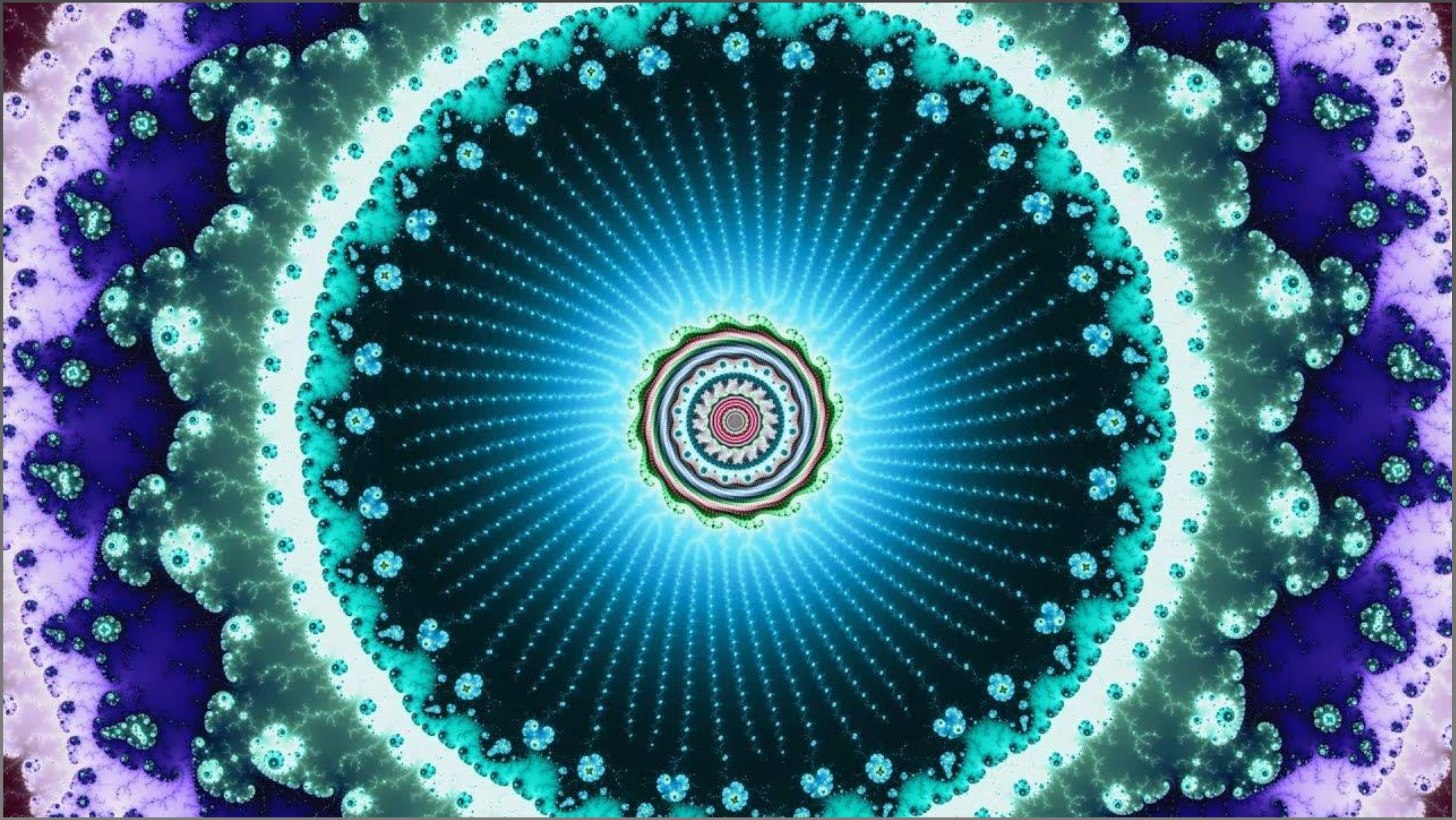
```python
import turtle

def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-15,t)
        t.right(20)
        t.backward(branchLen)

def main():
    t = turtle.Turtle()
    myWin = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75,t)
    myWin.exitonclick()

main()
```



Sierpinski Triangle



Eye of the Universe



https://youtu.be/pCpLWbHVNhk

Reinforcement

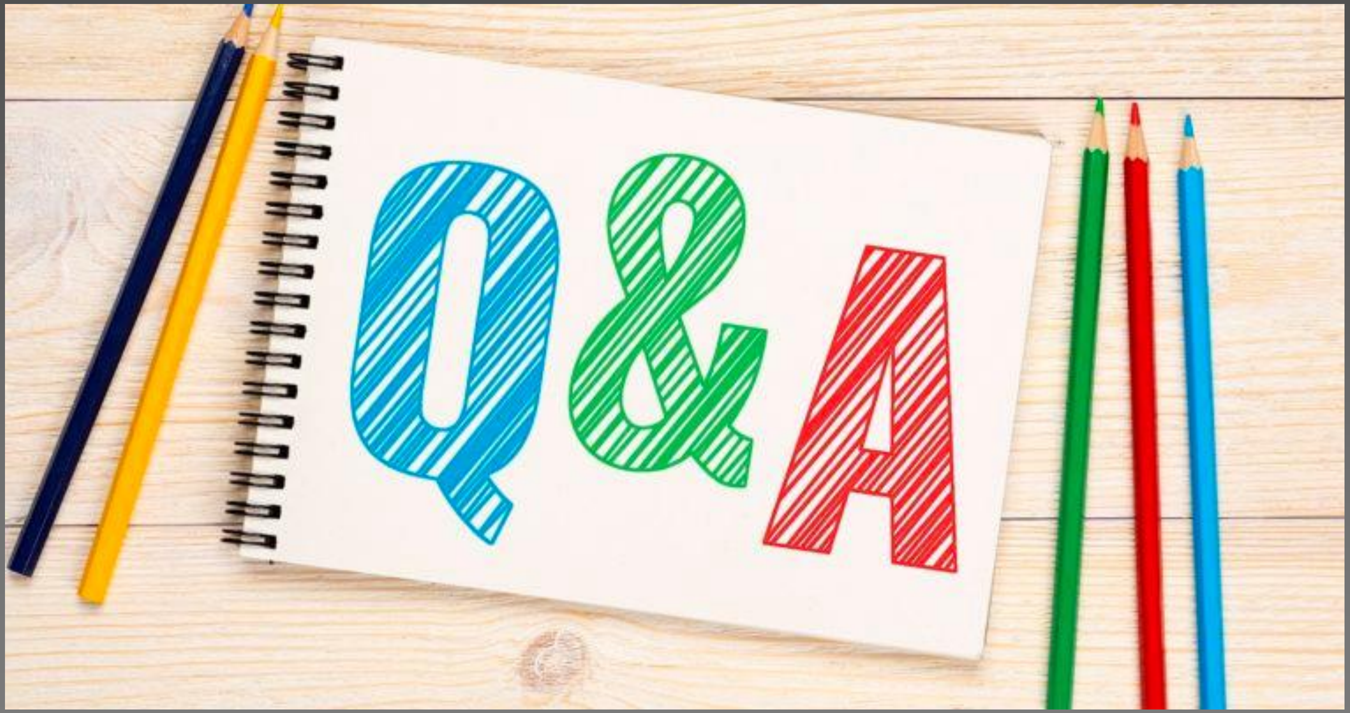
Question 1

Describe a recursive algorithm for finding the maximum element in a sequence. What is the running time and space usage?

Question 2

Describe a recursive function for converting a string of digits into the integer it represents. For example, ’13531’ represents the integer 13,531.

CW Q&A

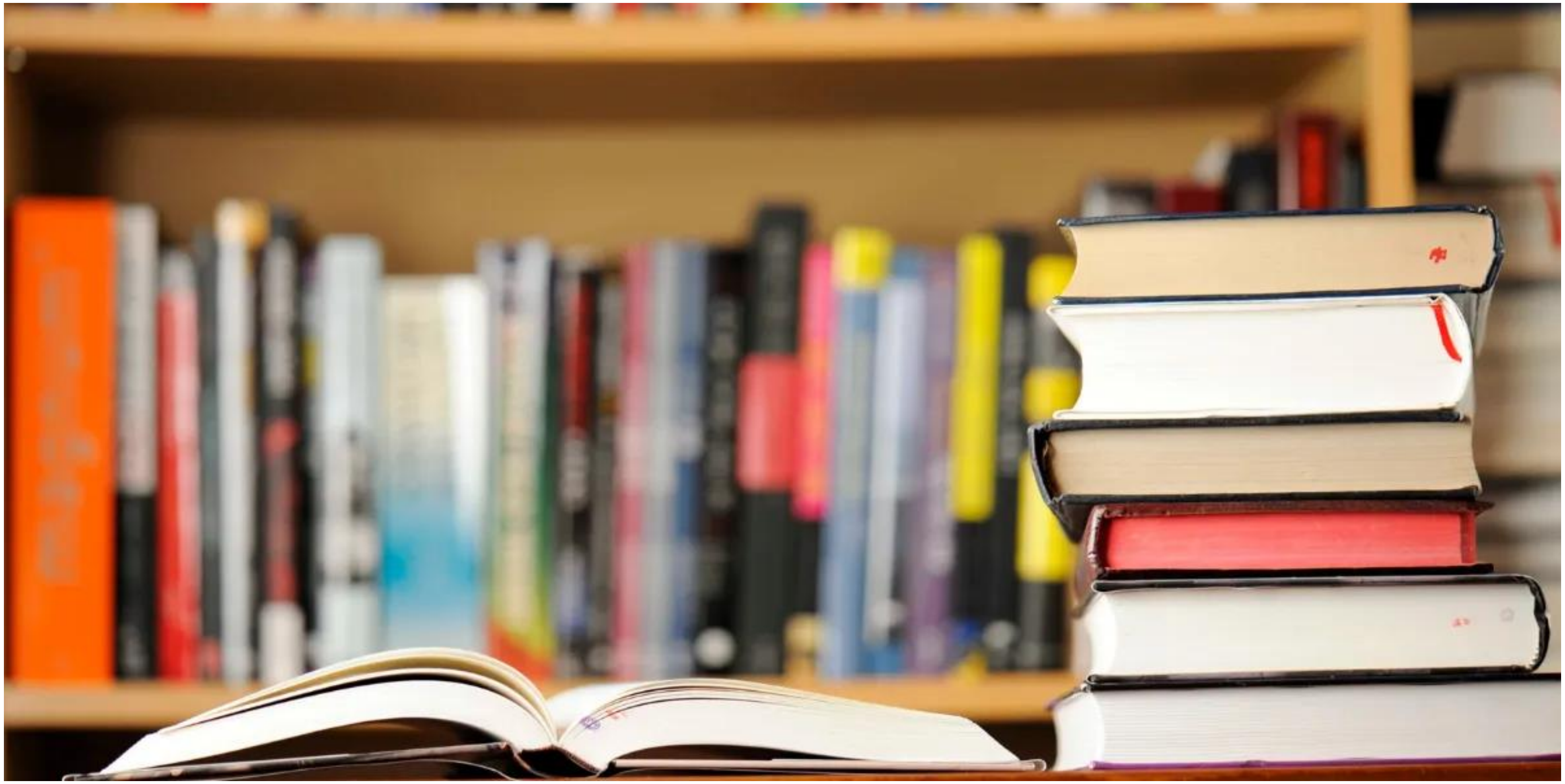


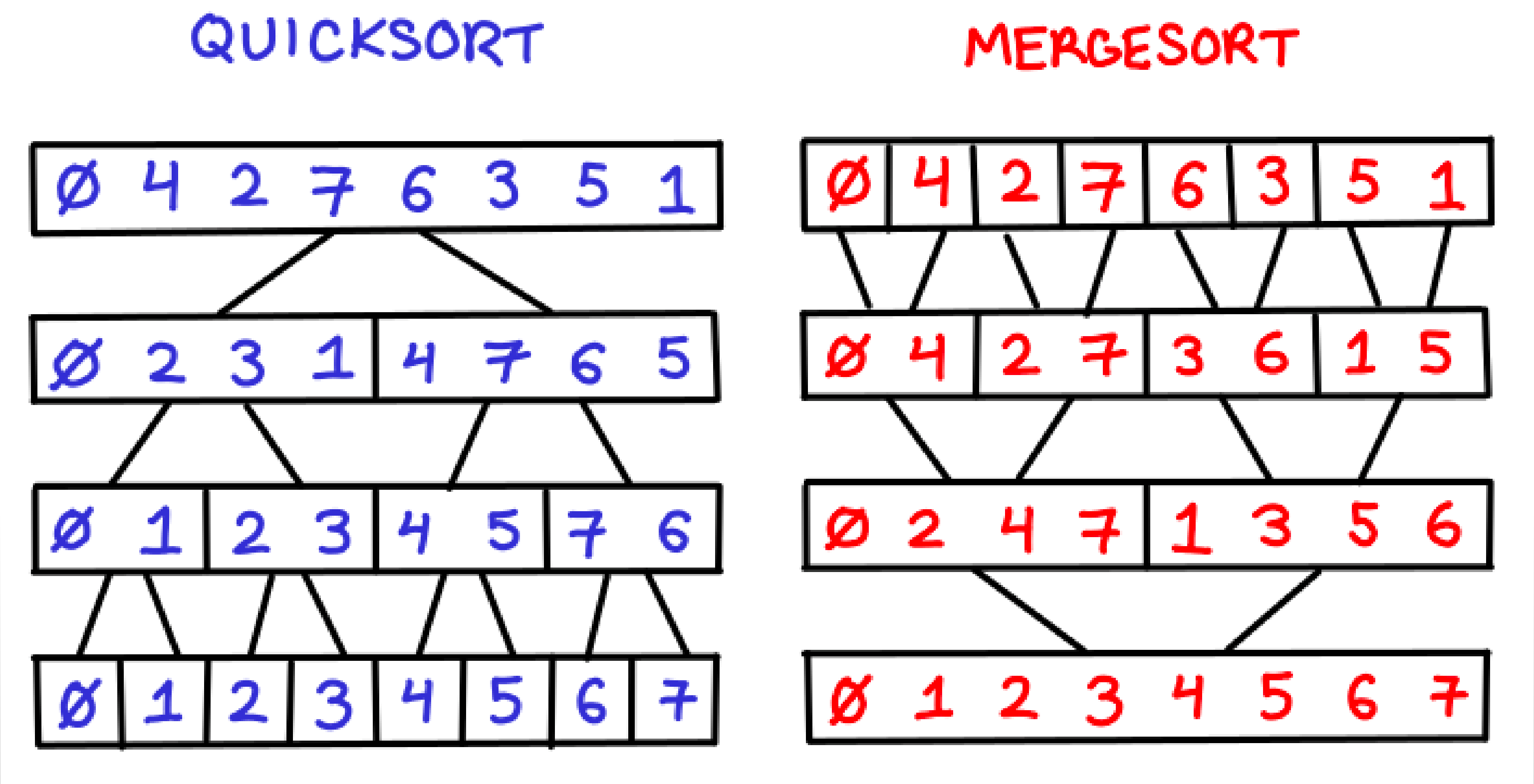
Quick overview

- Three laws of recursion
- Linear Recursion
- Binary Recursion
- Recursion analysis

Extra reading

- Tower of Hanoi
- Exploring a Maze
- Dynamic programming





https://hackernoon.com/

Next week!