Recursion

Data Structures & Algorithms with Python

Lecture 4

Outline

- Introduction
- Recursion Examples
- Analyzing the time-complexity of recursions
- Types of Recursion
- Designing Recursive Algorithms
- Tower of Hanoi Problem
- Summary

Introduction

Two ways to define repetition:

- Using loops: for-loop, while-loop etc.
- Using recursion.

Recursion: *It is a technique by which a function makes one or more calls to itself during execution, or by which a data structure relies upon smaller instances of the very same type of structure in its presentation.*

Few real examples of recursion in nature:

➔ Fractal patterns
➔ Russian Matryoshka Dolls

Illustrative Examples of recursion

- Factorial function
- English Ruler
- Binary Search
- File System

The Factorial Function

Definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n.(n-1).(n-2)\cdots 3.2.1 & \text{if } n \geq 1 \end{cases}$$

Example:

$$5! = 5.4.3.2.1 = 120$$

Recursive Definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n.(n-1)! & \text{if } n \geq 1 \end{cases}$$

A recursive definition contains one non-recursive ***base case*** and one or more ***recursive cases***.

The number of ways in which n distinct items can be arranged into a sequence = n!. In other words, ***permutation*** of n items = n!.

For example, the characters a, b and c can be arranged in 3! = 3.2.1 = 6 ways: abc, acb, bac, bca, cab and cba.

```
1 # Factorial Function
2
3 def factorial(n):
4   if n == 0:
5     return 1
6   else:
7     return n*factorial(n-1)
8
9 print(factorial(5))
```

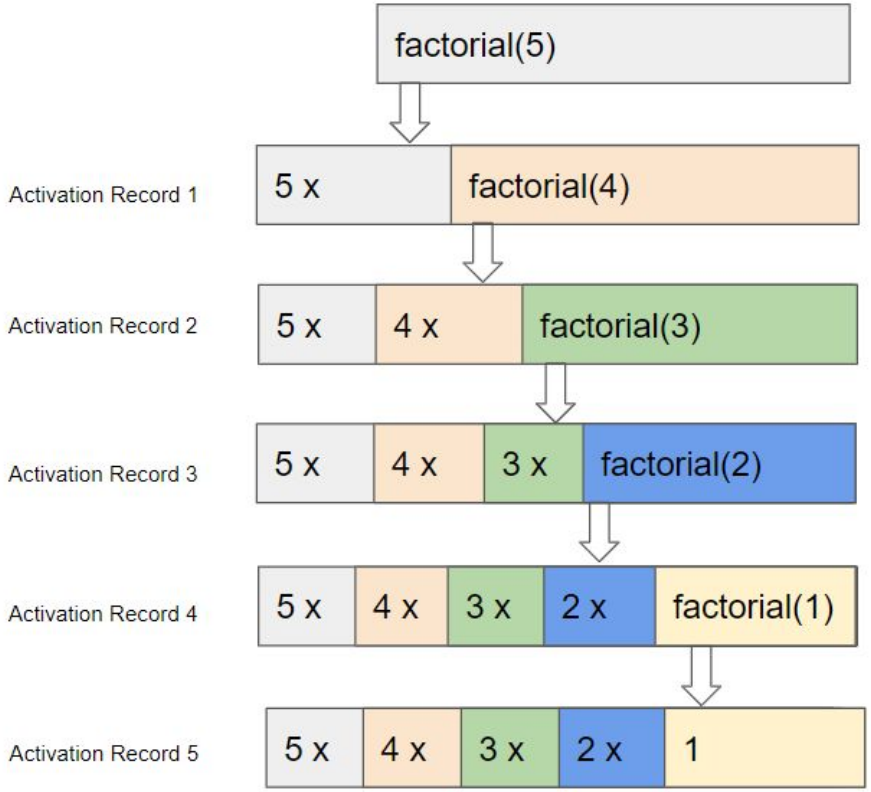
```
120
```

- Each time a function is called, a structure known as an ***activation record*** or ***frame*** is created to store information about the progress of that invocation of the function.

- Activation record includes a namespace for storing the function call parameters and local variables and information about which command in the body of the function is currently executing.

- When the execution of a function leads to a nested function call, the execution of the former function call is suspended and its activation record stores the place in the source code at which the flow of control should continue upon return of the nested call.

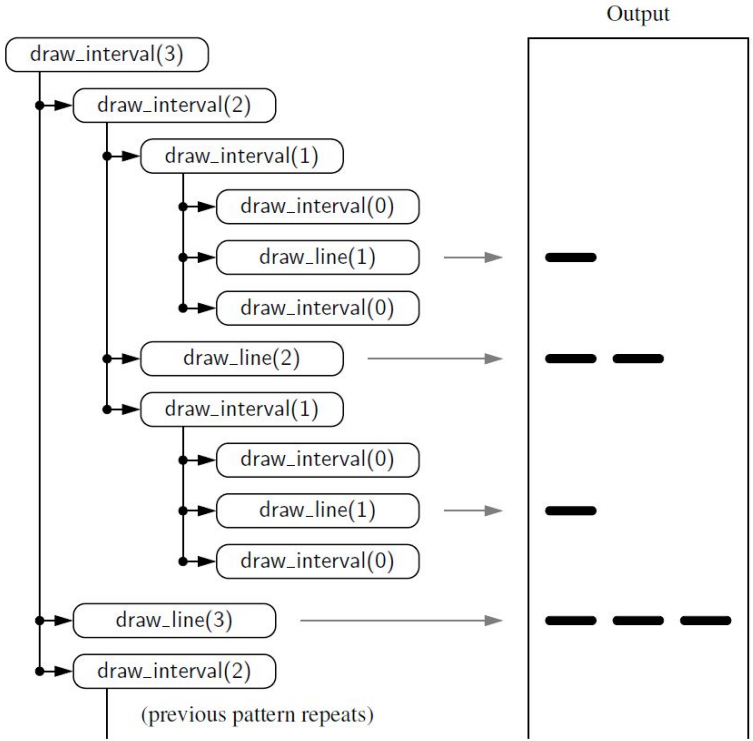
- There is a different activation record for each active call.

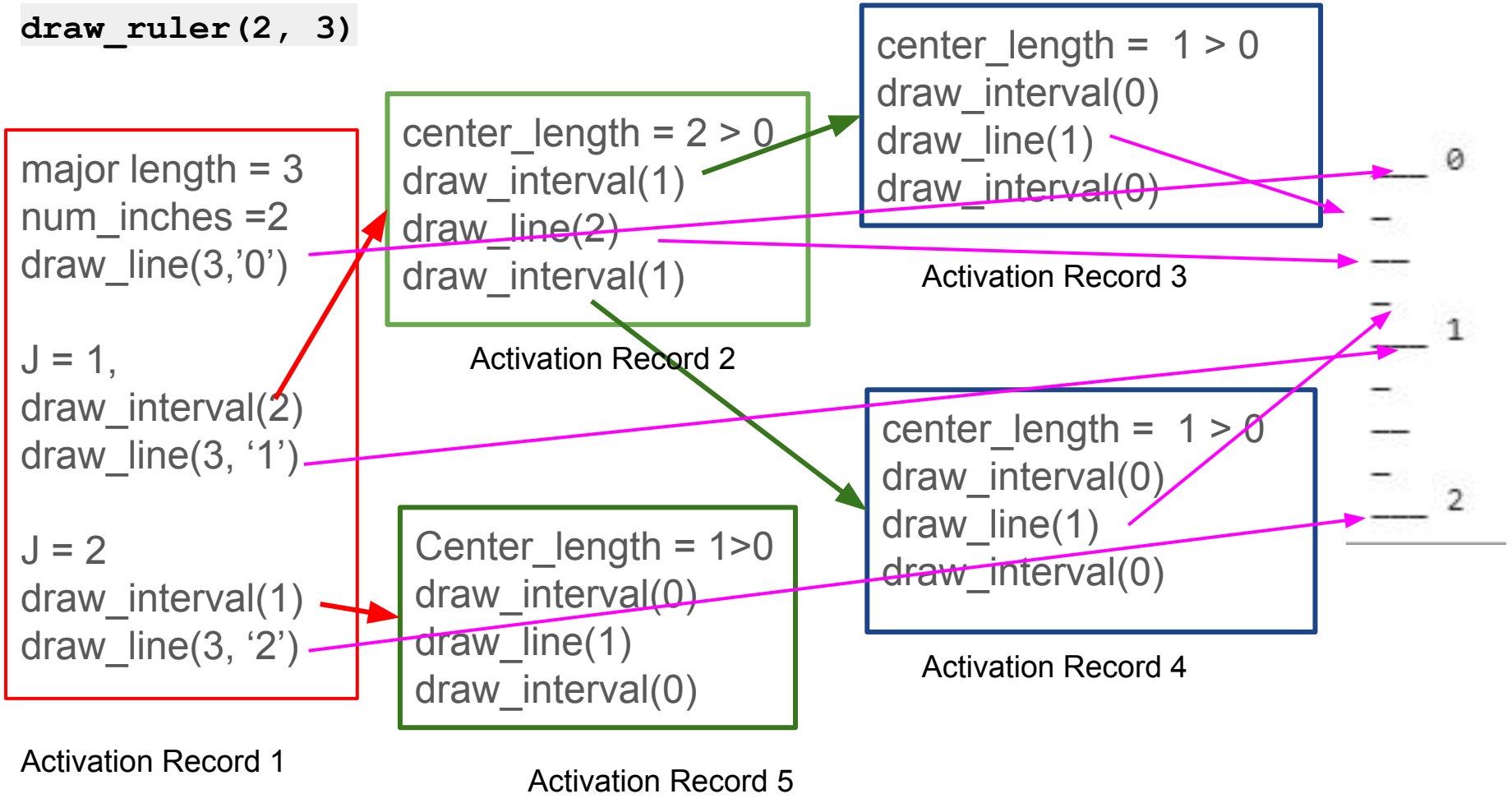


Drawing an English Ruler

```
def draw_line(tick_length, tick_label=''):
  '''
  Draw one line with given tick length (followed by optional label)
  '''
  line = '_'*tick_length
  if tick_label:
    line += ' ' + tick_label
  print(line)

def draw_interval(center_length):
  '''
  Draw tick interval based upon a central tick length
  '''
  if center_length > 0:  # stop when length drops to 0
    draw_interval(center_length - 1) # recursively draw top tics
    draw_line(center_length) # draw center tick
    draw_interval(center_length - 1) # recursively draw bottom ticks

def draw_ruler(num_inches, major_length):
  '''
  Draw English ruler with given number of inches, major tick length
  '''
  draw_line(major_length, '0')     # draw inch 0 line
  for j in range(1, 1+num_inches):
    draw_interval(major_length - 1)  # draw interior ticks for inch
    draw_line(major_length, str(j))  # draw nch j line and label

draw_ruler(2, 4)
#draw_ruler(2,3)
#draw_ruler(2,5)
```

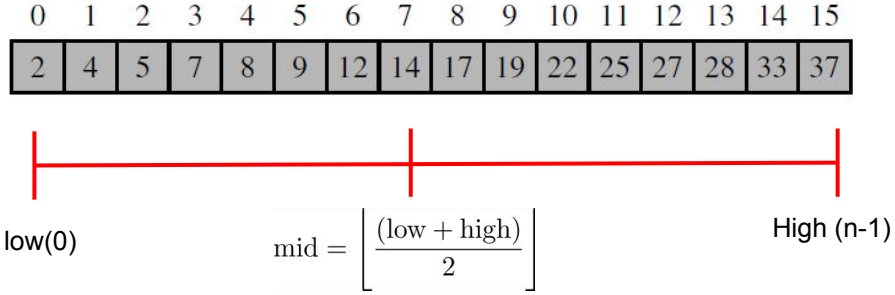
```
___ 0
_
__
_
___
_
__
_
___ 1
_
__
_
___
_
__
_
___ 2
```




draw_ruler(2, 3)
major length = 3
num_inches =2
draw_line(3,'0')
J = 1,
draw_interval(2)
draw_line(3, '1')
J = 2
draw_interval(1)
draw_line(3, '2')
Activation Record 1
center_length = 2 > 0
draw_interval(1)
draw_line(2)
draw_interval(1)
Activation Record 2
Center_length = 1>0
draw_interval(0)
draw_line(1)
draw_interval(0)
Activation Record 5
center_length = 1 > 0
draw_interval(0)
draw_line(1)
draw_interval(0)
Activation Record 3
center_length = 1 > 0
draw_interval(0)
draw_line(1)
draw_interval(0)
Activation Record 4
0
1
2

Binary Search

- ***Sequential search*** is used for ***unsorted*** sequence - Loop over each and every element to find the target $\sim O(n)$
- ***Binary search*** is used to efficiently locate a target value within a ***sorted*** sequence of n elements $\sim O(\log n)$
- For any index j in a sorted array, all the values stored at indices 0, 1, … j-1 are less than or equal to the value at index j.
- In BS, the sequence is partitioned into two sub-sequences and then the search is performed only in one of the sub-sequence.



Consider three cases:

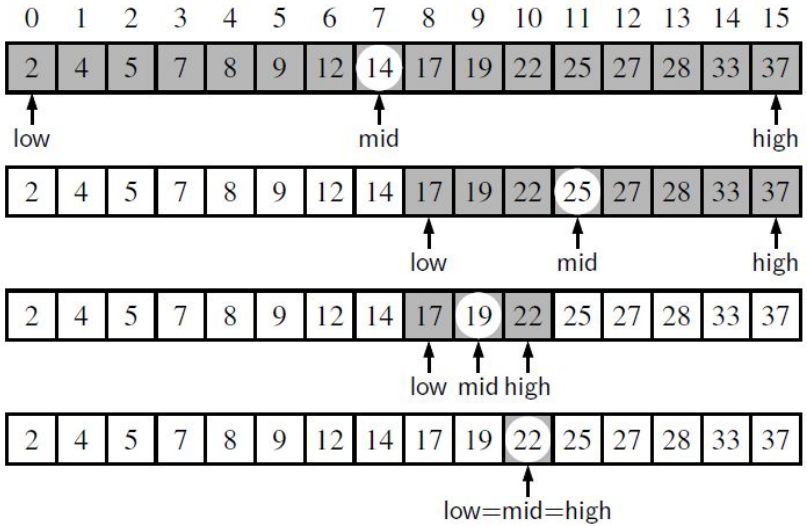
➔ If `target == data[mid]`, target is found, search ends.

➔ If `target < data[mid]`, recur the search in the first half of the sequence.

➔ If `target > data[mid]`, recur the search in the second half of the sequence.

➔ Repeat these steps until the interval `[low, high]` is empty or `low > high`.

Search ‘22’ in the array

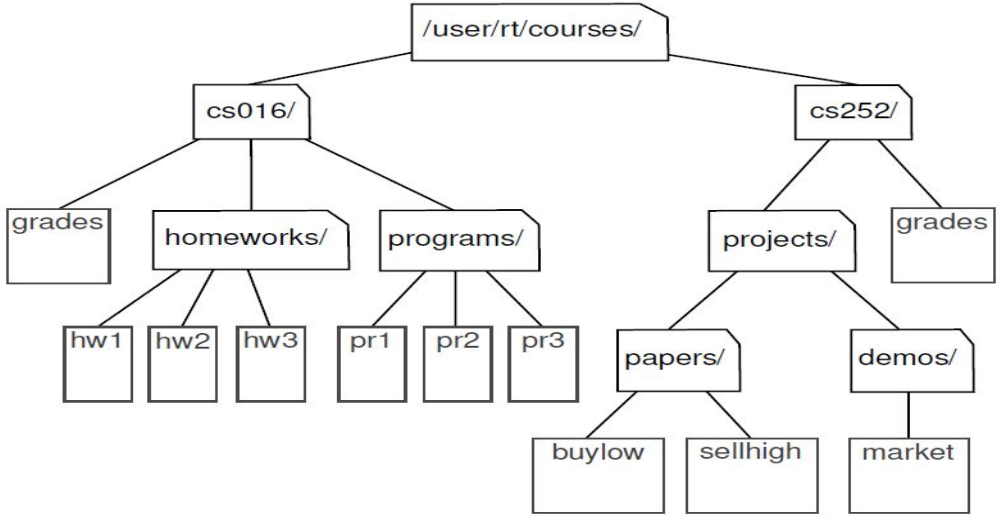



```python
# Binary Search algorithm through recursion

def binary_search(data, target, low, high):
  '''
  Return True if target is found in the indicated interval [low, high]
  '''
  if low > high:
    return False  # interval is empty, no match found
  else:
    mid = (low + high) // 2
    if target == data[mid]:
      return mid, True # Match found
    elif target < data[mid]:
      # recur on the first half of the interval
      return binary_search(data, target, low, mid-1)
    else:
      # recur on the second half of the interval
      return binary_search(data, target, mid+1, high)


a = [2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]

print(binary_search(a, 22, 0, 15))

```

```
(10, True)
```

File Systems

- In modern operating system, the file-system directories (or folders) are defined in a recursive way.
- Many OS functions such as copying or deleting implement recursive algorithms.
- Example: cumulative disk usage for all files and directories within a particular directory.



Algorithm DiskUsage(path):
Input: A string designating a path to a file-system entry
Output: The cumulative disk space used by that entry and any nested entries

```
total = size(path)                    {immediate disk space used by the entry}
if path represents a directory then
    for each child entry stored within directory path do
        total = total + DiskUsage(child)          {recursive call}
return total
```

Following functions from Python's os module will be used:

- **`os.path.getsize(path)`**:
  - Return the immediate disk usage (measured in bytes) for the file or directory that is identified by the string path
- **`os.path.isdir(path)`**:
  - Return True if entry designated by string path is a directory; False otherwise
- **`os.listdir(path)`**:
  - Return a list of strings that are the names of all entries within a directory designated by string path
- **`os.path.join(path, filename)`**:
  - Compose the path string and filename string using an appropriate operating system separator between the two ('/' in Linux)

```python
# Disk Usage Example
import os
def disk_usage(path):
  '''
  Returns the number of bytes used by a file/folder and any descendents
  '''
  total = os.path.getsize(path)
  if os.path.isdir(path):  # if this is a directory
    for filename in os.listdir(path):
      childpath = os.path.join(path,filename)
      total += disk_usage(childpath)

  print('{0:<7}'.format(total),path)
  return total

disk_usage('./')
```

```
49      ./.config/configurations/config_default
4145    ./.config/configurations
8168    ./.config/logs/2019.08.27/16.17.17.407104.log
394     ./.config/logs/2019.08.27/16.17.07.367707.log
21809   ./.config/logs/2019.08.27/16.17.00.530950.log
394     ./.config/logs/2019.08.27/16.17.21.663235.log
630     ./.config/logs/2019.08.27/16.17.22.686467.log
35491   ./.config/logs/2019.08.27
39587   ./.config/logs
0       ./.config/config_sentinel
134     ./.config/.last_update_check.json
7       ./.config/active_config
5       ./.config/gce
32      ./.config/.metricsUUID
48006   ./.config
930     ./sample_data/README.md
1697    ./sample_data/anscombe.json
18289443 ./sample_data/mnist_test.csv
1706430 ./sample_data/california_housing_train.csv
36523880 ./sample_data/mnist_train_small.csv
301141  ./sample_data/california_housing_test.csv
56827617 ./sample_data
56879719 ./
56879719
```

~ 56.8 MB

Analyzing Recursive Algorithms

- Computing Factorials
  - Each individual activation of factorial() executes a constant number of operations.
  - To compute factorial(n), there are n+1 activations. The main function that calls the factorial() function has its own activation record.
  - Time complexity ~ O(n)

Analyzing the English Ruler Example

- Drawing an English Ruler
  - For c >= 0, a call to draw_interval(c) results in precisely $2^c - 1$ lines of output.

    Proof by Induction:
    Base case: For c = 0, $2^0 - 1 = 1 - 1 = 0$ (no lines are drawn)

    Assume that above proposition holds true for $c' = c - 1 < c$. Hence, draw_interval(c-1) prints $2^{c-1} - 1$ lines of output

    Inductive step: Each call to draw_interval(c) for c>0 spawns two calls to draw_interval(c-1) and a single call to draw_line().
    For any general c > 0, the number of lines printed:

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

Analyzing binary search algorithm

Proposition: The binary search algorithm runs in ***O(log n)*** time for a sorted sequence with n elements.

Proof: The number of candidates to be search with each recursive call = high - low +1

The number of candidates to be searched is reduced by at least one half with each recursive call. In general, after the jth call in a binary search, the number of candidates remaining is at most $n/2^j$

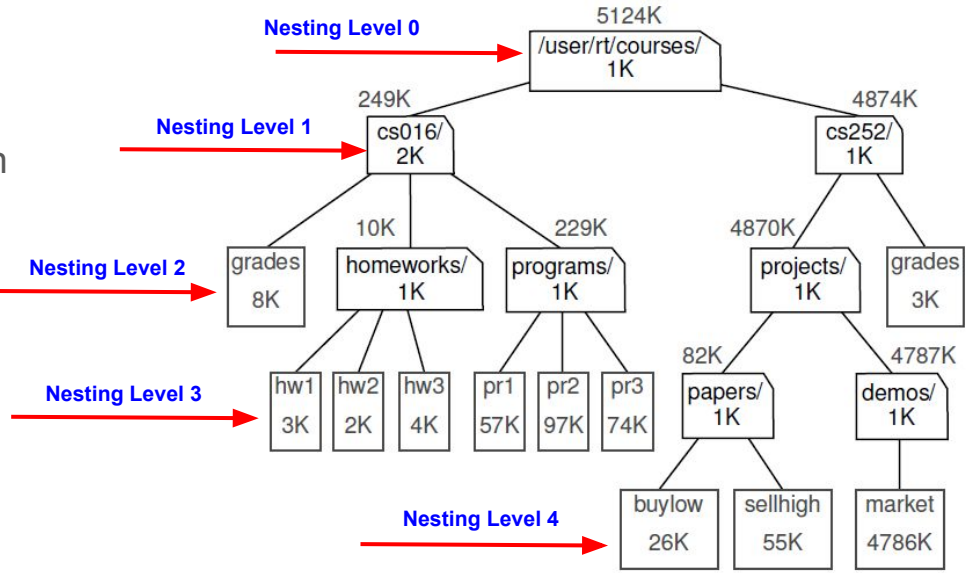
The maximum number of recursive calls to be performed is the smallest integer r such that

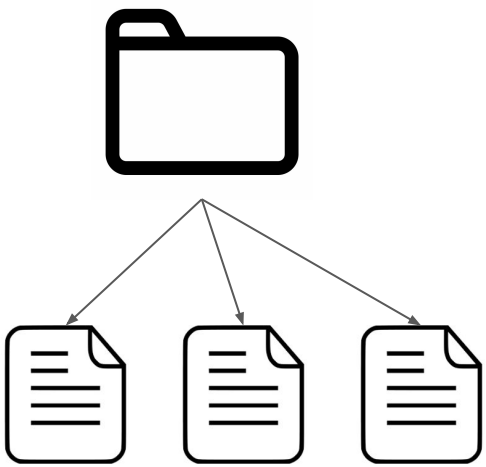
$$\frac{n}{2^r} < 1 \Rightarrow r > \log n \qquad \therefore r = \lfloor \log n \rfloor + 1$$

This implies that binary search runs in O(log n) time.

Analyzing disk space usage algorithm

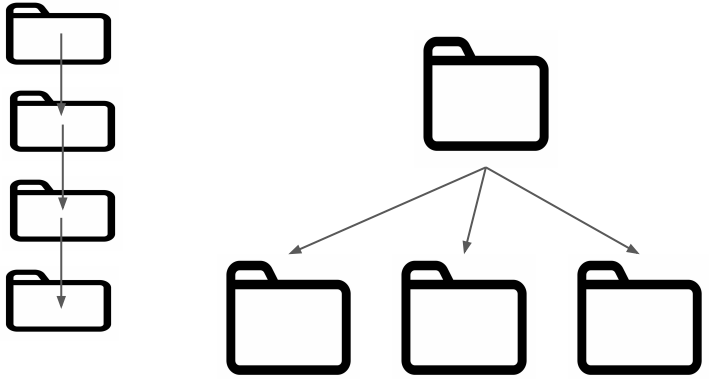
- Total number of file-system entries in a given part of the file system is n (n = 19, in this case).
- It can be proved by induction that there is exactly one recursive invocation of `disk_usage()` upon each entry at the nesting level k.
- In the worst case, it has running time of order $O(k^2) \sim O(n^2)$
- A tighter bound will be $O(n)$ where n is the total number of entries in the file-system.
- ***Amortization:*** We can sometimes get a tighter bound on a series of operations by considering the cumulative effect, rather than assuming that each achieves a worst case.





calls to disk_usage() function: 4

No of entries in the file system is n= 4.
Case I: The folder contains 3 files.



calls to disk_usage() function: 4

Case II: The folder contains 3 empty directories.

O(n) is a tighter bound on complexity for disk_usage() function.

Bad use of Recursion - Revisiting Element uniqueness problem

Let n = stop - start

If n = 1, running time of unique3() is O(1).

If general, a single call to unique3 for problem size n may result in two recursive calls on problems of size n-1.

Those two calls with size n-1 will lead to four calls with a size of n-2 leading to 8 calls in turn with size n-3 and so on …

Total number of function calls is given by the geometric summation:

$$1 + 2 + 4 + 8 + \ldots + 2^{n-1} = 2^n - 1$$

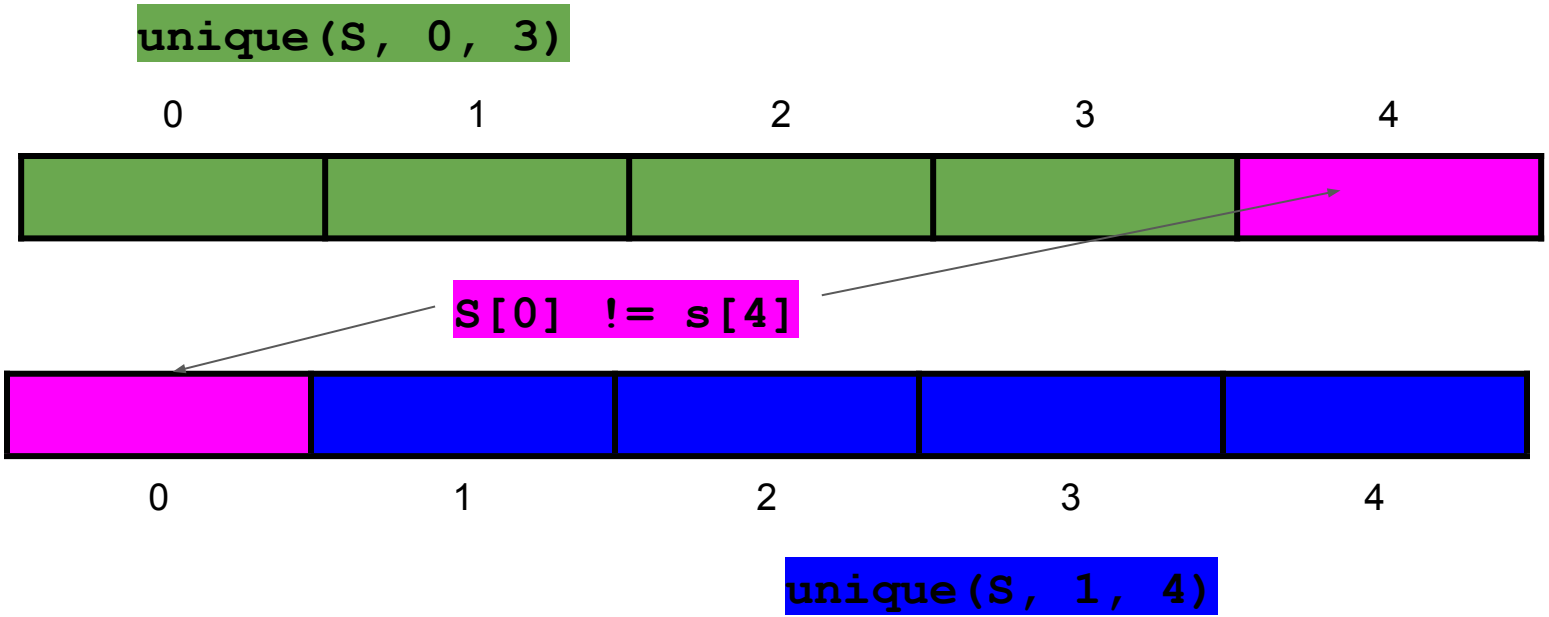
```python
# Element uniqueness problem
def unique3(S, start, stop):
  '''
  Return True if there is no duplicate elements in the array
  '''
  if stop - start <= 1: return True  # at most one item
  elif not unique3(S, start, stop-1): return False # first part has duplicate
  elif not unique3(S, start+1, stop): return False # second part has duplicate
  else: return S[start] != S[stop-1] # do first and last differ


a = [2, 5, 7, 6, 8]
print(a[0:5])
print(unique3(a, 0,5))
```

```
[2, 5, 7, 6, 8]
True
```

Run time of function unique3 is $O(2^n)$

```
unique(S, 0, 5)
```

Each function call leads to two more function calls to itself leading to exponential time complexity.



Another bad use of Recursion: Bad_fibonacci

- Let C_n be the number of calls performed in the execution of `bad_fibonacci(n)`.
- The number of calls more than doubles for every two consecutive indices. In other words,

$$C_2 > 2 \times C_0$$
$$C_4 > 2 \times C_2$$
$$C_6 > 2 \times C_4$$
$$\vdots$$
$$C_n > 2 \times C_{n/2}$$

- This implies that the function bad_fibonacci has a time complexity of $O(2^n)$

$$C_8 > 2 \times C_6 > 2 \times 2 \times C_4 > 2 \times 2 \times 2 \times C_2 > 2 \times 2 \times 2 \times C_0$$
$$C_8 > 2^4 \times C_0$$

```
1 ## Bad fibonacci
2 def bad_fibonacci(n):
3   '''
4   Return the nth Fibonacci number
5   '''
6   if n <= 1:
7     return n
8   else:
9     return bad_fibonacci(n-2) + bad_fibonacci(n-1)
10
11 # ----------------------
12 bad_fibonacci(10)
13
```

```
55
```

$$\begin{aligned}
c_0 &= 1 \\
c_1 &= 1 \\
c_2 &= 1 + c_0 + c_1 = 1 + 1 + 1 = 3 \\
c_3 &= 1 + c_1 + c_2 = 1 + 1 + 3 = 5 \\
c_4 &= 1 + c_2 + c_3 = 1 + 3 + 5 = 9 \\
c_5 &= 1 + c_3 + c_4 = 1 + 5 + 9 = 15 \\
c_6 &= 1 + c_4 + c_5 = 1 + 9 + 15 = 25 \\
c_7 &= 1 + c_5 + c_6 = 1 + 15 + 25 = 41 \\
c_8 &= 1 + c_6 + c_7 = 1 + 25 + 41 = 67
\end{aligned}$$

An Efficient implementation of Fibonacci algorithm using recursion

- We modify the requirement by making it return two consecutive numbers instead of one.
- Each call leads to only one recursive call with problem size reducing by 1 at each call.
- So for n elements, n recursive calls are made.
- So the time-complexity of good_fibonacci(n) is O(n).

```
# Good Fibonacci
def good_fibonacci(n):
  '''
  return pair of fibonacci numbers F(n) and F(n-1)
  F(-1) = 0
  '''
  if n <= 1:
    return (n, 0)
  else:
    (a,b) = good_fibonacci(n-1)
    return (a+b, a)

#----------------------------
print(good_fibonacci(10))
```

```
(55, 34)
```

```
(1, 0)
(1, 1)
(2, 1)
(3, 2)
(5, 3)
(8, 5)
(13, 8)
(21, 13)
(34, 21)
(55, 34)
```

Maximum Recursive Depth in Python

- Each recursive call makes another recursive call without ever reaching a base case - ***infinite recursion***.

  Example:

```
def fib(n):
    return fib(n)
```

- Python limits the total number of function activations to some fixed value (default = 1000). If this number is exceeded, a RuntimeError or RecursionError exception is raised by the interpreter.

- The default recursion limit can be changed:

```
import sys
old = sys.getrecursionlimit( )
sys.setrecursionlimit(1000000)
```

```
1 # Binary Search algorithm through recursion
2
3 def binary_search(data, target, low, high):
4   '''
5   Return True if target is found in the indicated interval [low, high]
6   '''
7   if low > high:
8     return False  # interval is empty, no match found
9   else:
10    mid = (low + high) // 2
11    if target == data[mid]:
12      return mid, True # Match found
13    elif target < data[mid]:
14      # recur on the first half of the interval
15      return binary_search(data, target, low, mid-1)
16    else:
17      # recur on the second half of the interval
18      return binary_search(data, target, mid, high)
19
20
21 a = [2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]
22
23 print(binary_search(a, 22, 0, 15))
24
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-18-ea0cadb9254f> in <module>()
     20 a = [2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]
     21
---> 22 print(binary_search(a, 22, 0, 15))

 3 frames
... last 1 frames repeated, from the frame below ...

<ipython-input-18-ea0cadb9254f> in binary_search(data, target, low, high)
     15     else:
     16       # recur on the second half of the interval
---> 17       return binary_search(data, target, mid, high)
     18
     19

RecursionError: maximum recursion depth exceeded in comparison
```

Further Examples of Recursion

- ***Linear Recursion*** - one recursive call starts at most one other.
- ***Binary recursion*** - one recursive call may start two others.
- ***Multiple recursion*** - one recursive call may start three or more others.

Linear Recursion

Linear Recursion - one recursive call starts at most one other.

Examples:

- ➔ Factorial
- ➔ Good_fibonacci
- ➔ Binary search

Linear recursion terminology reflects the structure of the recursion trace, not the asymptotic analysis of the running time.

Summing the elements of a sequence

n = 4	linear_sum(S,4)	S[3] + linear_sum(S,3)	20 + 30 = 50
n = 3	linear_sum(S,3)	S[2] + linear_sum(S,2)	15 + 15 = 30
n = 2	linear_sum(S,2)	S[1] + linear_sum(S,1)	10 + 5 = 15
n = 1	linear_sum(S,1)	S[0] + linear_sum(S, 0)	5 + 0 = 5
n = 0	linear_sum(S,0)	0	0

Run time $\sim O(n)$

```
1 def linear_sum(S, n):
2   '''
3   Return the sum the first n numbers of a sequence S
4   '''
5   if n == 0:
6     return 0
7   else:
8     return linear_sum(S, n-1) + S[n-1]
9
10 # Same as above but does not take the additional argument
11 def linear_sum2(S):
12   n = len(S)
13
14   if n == 0:
15     return 0
16   else:
17     return linear_sum2(S[0:n-1]) + S[n-1]
18
19 #----------------------------------------
20 b = [5, 10, 15, 20]
21 print(linear_sum(b, 4))
22 print(linear_sum2(b))
```

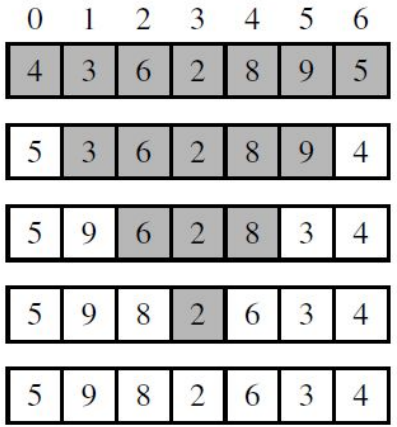
```
50
50
```

Reversing a Sequence with Recursion

- The code is guaranteed to terminate after a total of $1 + \lfloor \frac{n}{2} \rfloor$ recursive calls.
- Runtime $\sim O(n)$

```
1 # Reversing a sequence with recursion
2
3 def reverse(S, start, stop):
4   '''
5   Reverse the elements in implicit slice S[start:stop]
6   '''
7   if start < stop - 1:     # if at least two elements
8     S[start], S[stop-1] = S[stop-1], S[start]  # swap first and last
9     reverse(S, start+1, stop-1)
10
11 a = [4,3,6,2,8,9,5]
12 reverse(a, 0, 7)
13 print(a)
```

```
[5, 9, 8, 2, 6, 3, 4]
```



Recursive algorithms for computing Powers

- Trivial recursion definition

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot power(x, n-1) & \text{otherwise.} \end{cases}$$

- Runtime $\sim O(n)$

```
1 # Recursive algorithms for computing powers
2
3 def power(x,n):
4   '''
5   Compute the value x**n for integer n
6   '''
7   if n ==0:
8     return 1
9   else:
10    return x * power(x, n-1)
11
12 #----------------------------
13 print(power(2,13))
```

```
8192
```

Let $k = \left\lfloor \frac{n}{2} \right\rfloor$

If n is even, $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n}{2}$

$\Rightarrow (x^k)^2 = (x^{\frac{n}{2}})^2 = x^n$

If n is odd, $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n-1}{2}$

$\Rightarrow (x^k)^2 = (x^{\frac{n-1}{2}})^2 = x^{n-1}$

$\Rightarrow x^n = x.(x^k)^2$

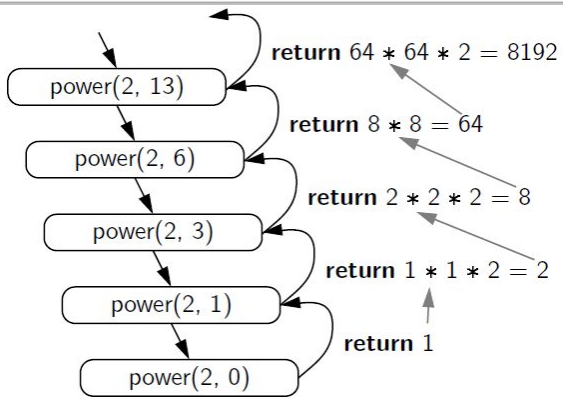
Improved Recursion algorithm:

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 & \text{if } n > 0 \text{ is odd} \\ \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

Run-time $\sim O(\log n)$

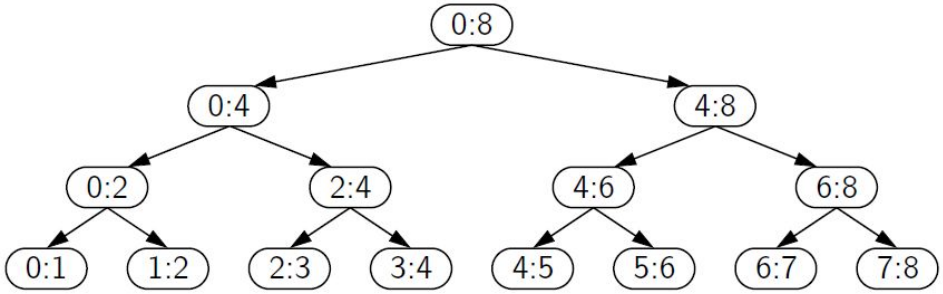
```python
# An efficient version of the power algorithm
def power2(x, n):
  '''
  Compute the value x**n for integer n
  '''
  if n ==0:
    return 1
  else:
    partial = power2(x, n//2)
    result = partial * partial
    if n%2 == 1: # if n is odd, include extra factor of x
      result *= x
    return result
#---------------------------------
print(power2(2,13))
```

```
8192
```



Binary Recursion

- One functions make two recursive calls.
- Examples:
  - English ruler
  - Bad_fibonacci
  - Binary_sum



Recursion trace of binary_sum(0,8)

```
# Binary sum - Summing n elements of a sequence

def binary_sum(S, start, stop):
  '''
  Return the sum of the numbers in implicit slice S[start:stop]
  '''
  if start >= stop:
    return 0
  elif start == stop-1:
    return S[start]
  else:
    mid = (start+stop) // 2
    return binary_sum(S, start, mid) + binary_sum(S, mid, stop)

a = [5, 10, 3, 7, 9, 2, 4, 12]
print(binary_sum(a, 0, 8))
```

```
52
```

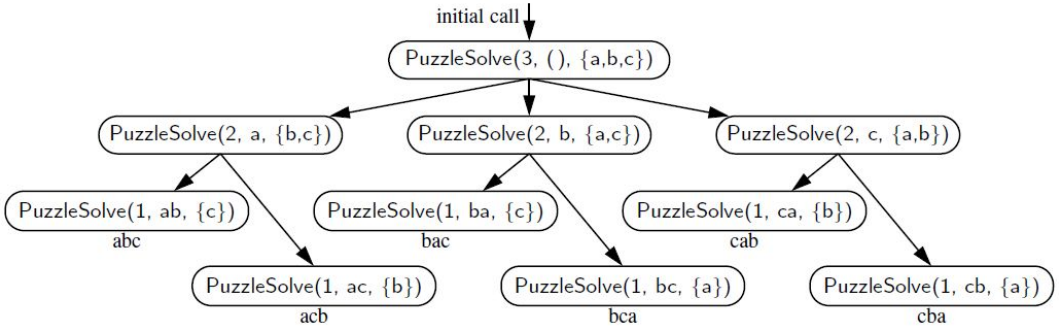
- For n elements the depth of recursion is $1 + \log_2 n$
- Binary sum uses $O(\log n)$ of additional space - activation frames
- However, the running time is $O(n)$ as there are $2n - 1$ function calls.

Multiple recursion

- Multiple recursion is a process in which a function may take more than two recursive calls.
- Examples:
  - Disk space usage of a file system.
  - Solving combinatorial puzzles.

Algorithm PuzzleSolve(k,S,U):
Input: An integer k, sequence S, and set U
Output: An enumeration of all k-length extensions to S using elements in U without repetitions

```
for each e in U do
    Add e to the end of S
    Remove e from U                          {e is now being used}
    if k == 1 then
        Test whether S is a configuration that solves the puzzle
        if S solves the puzzle then
            return "Solution found: " S
    else
        PuzzleSolve(k−1,S,U)                 {a recursive call}
    Remove e from the end of S
    Add e back to U                          {e is now considered as unused}
```



Designing Recursive Algorithms

Recursive algorithms usually have the following form:

- **Test the base case**.
  - Begin by testing for a set of base cases (there should be at least one)
  - These base cases are defined so that every possible chain of recursive calls will eventually reach a base case, and the base case should not use recursion.
- **Recur:** If not a base case, we perform one or more recursive calls.

Parameterizing a Recursion:

- Sometimes it becomes necessary to introduce parameters to define a recursive algorithm.
- Examples:

```
binary_search(data, target, low, high)
binary_sum(S, start, stop)
linear_sum(S,n)
reverse(S, start, stop)
```

Advantages and Disadvantages of Recursion

- Advantages of Recursion
  - provides a succinct way of achieve repetition by avoiding nested loops.
  - Makes the code readable and efficient.

- The usefulness of recursion comes at a modest cost.
  - Python interpreter must maintain ***activation records*** that keep track of the state of each nested call.
  - Such calls should be avoided where memory is at a premium.
  - Some forms of recursion can be eliminated without any use of auxiliary memory. e.g - ***tail recursions***.

Eliminating Tail Recursion

- A recursion is a ***tail recursion*** if any recursive call that is made from one context is the very last operation in that context, with the return value of the recursive call immediately returned by the enclosing recursion.

- By necessity, a tail recursion must be a linear recursion.

- Examples:

```
binary_search(data, target)
reverse(S, start, stop)
```

- Following are not valid examples of tail recursions (Why?):

```
factorial(n)
linear_sum(S,n)
```

- Any tail recursion can be re-implemented non recursively by enclosing the body in a loop for repetition, and by replacing a recursive call with new parameters by a reassignment of the existing parameters to those values.

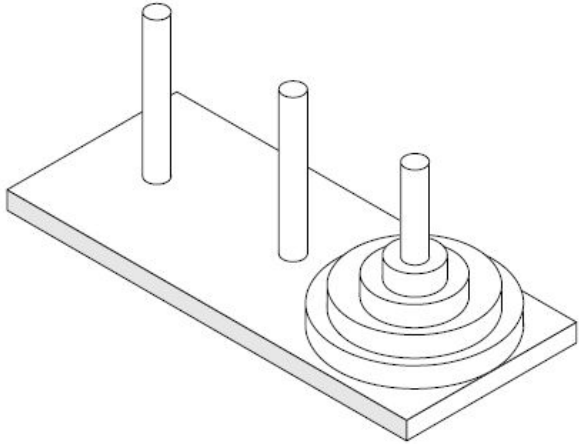
```python
# Eliminating tail recursion
# Non-recursive implementation of binary_search

def binary_search_iterative(data, target):
  '''
  Return True if target is found in the given python list
  '''
  low = 0
  high = len(data) - 1
  while low <= high:
    mid = (low + high) // 2
    if target == data[mid]:
      return True
    elif target < data[mid]:
      high = mid-1
    else:
      low = mid+1
  return False
```

```python
# Non recursive implementation of reverse algorithm
def reverse_iterative(S):
  '''
  Reverse elements in sequence S
  '''
  start, stop = 0, len(S)
  while start < stop - 1:
    S[start], S[stop-1] = S[stop-1], S[start] #swap first and last
    start, stop = start+1, stop-1 # narrow the range
```

Few More Examples of Recursion

Tower of Hanoi

- It consists of 3 rods or pegs or towers a number of disks of different sizes which can slide onto any rod.
- The puzzle starts with the disks on one rod in ascending order of its size, smallest at the top, thus making a conical tower shape.
- The objective is to move the entire stack to another rod, satisfying the following rules:
    - Only one disk may be moved at a time.
    - Each move consists of taking the uppermost disk in one rod and sliding onto another rod that may or may not have disks in them.
    - No disk may be placed on the top of a smaller disk.



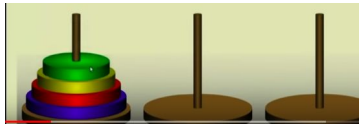
Algorithm:

- Move top n-1 disks from source to Auxiliary tower.
- Move the nth disk from source to destination tower.
- Move the n-1 disks from auxiliary tower to the destination tower.
- Transferring the top n-1 disks from source to auxiliary tower can again be considered as a fresh problem which can be solved by the above 3 steps.

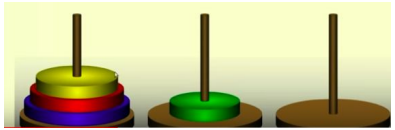
```python
# Tower of Hanoi Problem with 4 disks

def towers_of_hanoi(numberOfDisks, startPeg=1, endPeg=3):
  if numberOfDisks:
    towers_of_hanoi(numberOfDisks-1, startPeg, 6-startPeg-endPeg)
    print("Move disk {:d} from peg {:d} to peg {:d}".format(numberOfDisks, startPeg, endPeg))
    towers_of_hanoi(numberOfDisks-1, 6-startPeg-endPeg, endPeg)

#-----------------------------
towers_of_hanoi(4)
```

- Runtime complexity $\sim O(2^n)$
- Binary recursion - one call leads to two recursive calls.
- 4 disks requires 15 steps to solve
- 5 disks require 31 steps to solve

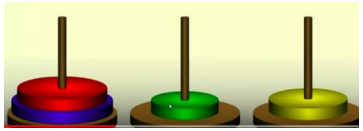
```
Move disk 1 from peg 1 to peg 2
Move disk 2 from peg 1 to peg 3
Move disk 1 from peg 2 to peg 3
Move disk 3 from peg 1 to peg 2
Move disk 1 from peg 3 to peg 1
Move disk 2 from peg 3 to peg 2
Move disk 1 from peg 1 to peg 2
Move disk 4 from peg 1 to peg 3
Move disk 1 from peg 2 to peg 3
Move disk 2 from peg 2 to peg 1
Move disk 1 from peg 3 to peg 1
Move disk 3 from peg 2 to peg 3
Move disk 1 from peg 1 to peg 2
Move disk 2 from peg 1 to peg 3
Move disk 1 from peg 2 to peg 3
```



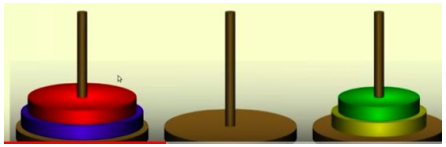
Move 0



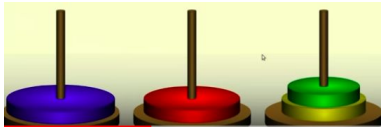
Move 1



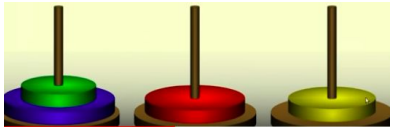
Move 2



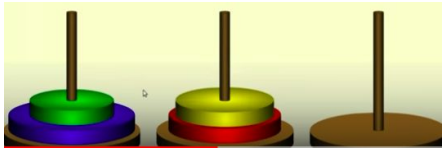
Move 3



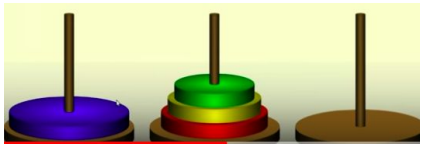
Move 4



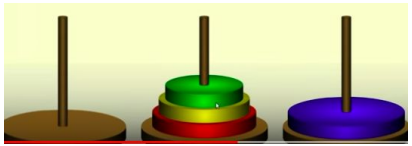
Move 5



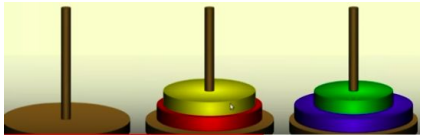
Move 6



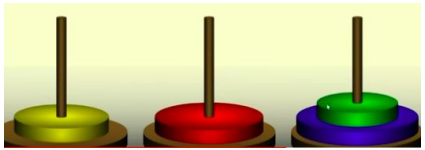
Move 7



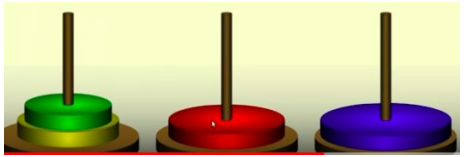
Move 8



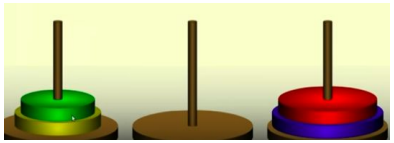
Move 9



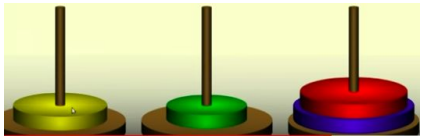
Move 10



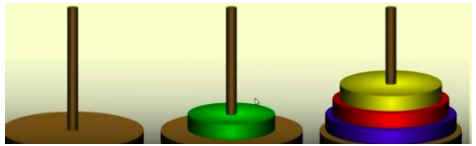
Move 11



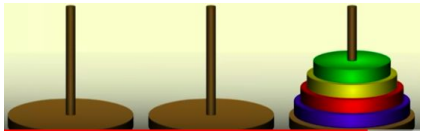
Move 12



Move 13



Move 14



Move 15

Check for a sorted array

- Given an array, check whether the array is in sorted order
- Linear Recursion
- Runtime complexity is O(n)

```python
# Check if an array is sorted

def is_array_in_sorted_order(A):
  # Base case
  if len(A) < 1:
    raise ValueError("array can not be empty")
  elif len(A) == 1:
    return True
  else:
    return A[0] <= A[1] and is_array_in_sorted_order(A[1:])

a = [127, 220, 246, 277, 321, 454, 534, 565, 933]
b = [4, 3, 1, 5, 7, 8, 0]
print(is_array_in_sorted_order(a))
print(is_array_in_sorted_order(b))

```

```
True
False
```

Summary

We cover the following topics in this lecture:

- What is recursion and why is it needed?
- Some examples of recursive algorithm
- Analyzing their runtime complexity & memory complexity
- Different types of recursion
- Downside of recursion