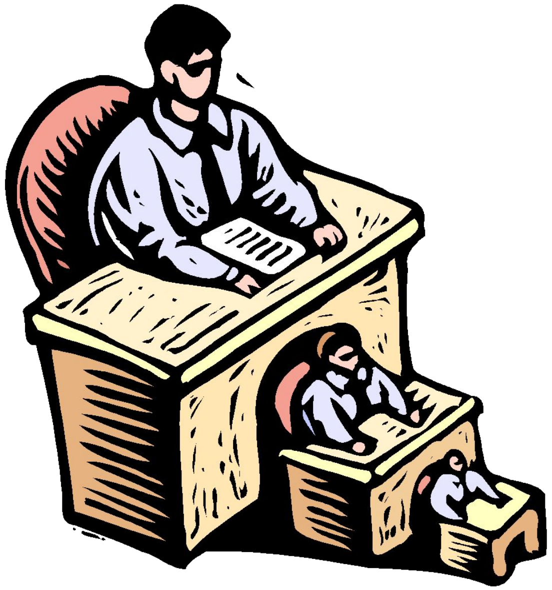
COMP 125 Programming with Python
Recursion



Mehmet Sayar

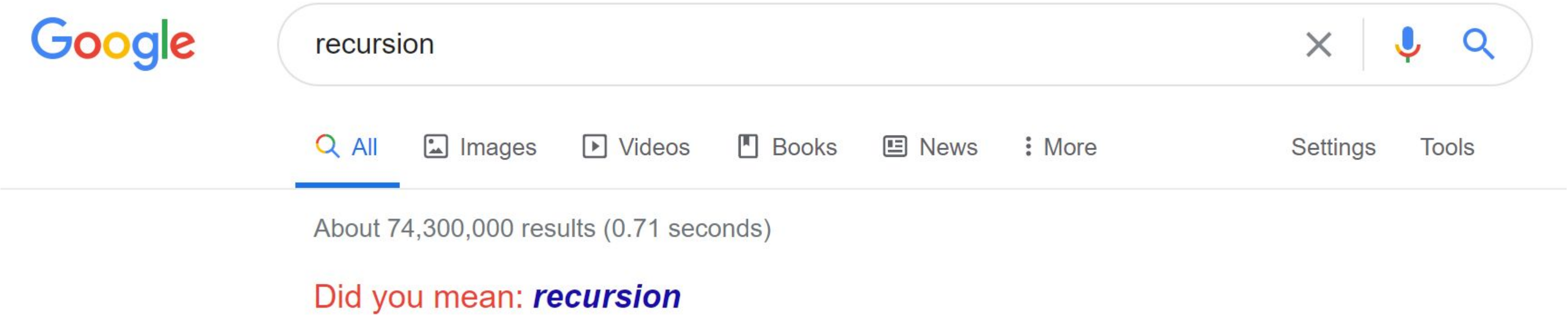
Koç University

Recursion

- “The repeated application of a **recursive** procedure or definition”
- Occurs when something is defined in terms of itself or its type
- Any non-CS examples?
- Sourdough Ingredients:
  - 300g water, 100g **sourdough**, …

- Matryoshka Dolls:

Recursion in CS



- "To understand recursion, you must first understand recursion."
- A method of defining a function in terms of its own definition, i.e., when a function calls itself.

Recursion in CS

- Classic Example– the factorial function:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

- Recursively

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

- Applied when the solution of a problem depends on solutions to the smaller instances of the same problem. One of the central ideas in CS!
- Let's us implement "a lot of computation" with very few lines of code!
- There are downsides too …

Factorial Example

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

```
def factorial(n):
    if n < 0:
        raise ValueError("Negative values not allowed in factorial")
    elif n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Base case

Recursive case

Content of a Recursive function

- **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 or we would have *infinite recursion*

- **Recursive calls**
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Exercise: Sum of numbers from 1 to n

Iteratively

```
def sum(n):
  s = 0
  for i in range(n+1):
    s += i
  return s
```

Recursively

```
def sum(n):
  if n < 1:
    return 0
  else:
    return n + sum(n - 1)
```

Solving A Problem Recursively

- Break into smaller problems
- Solve each sub-problem recursively
- Repeat until the problems “easy or small enough” (base case)
- Assemble sub-solutions

```
Algorithm recursiveAlgorithm(input)
  if isBaseCase(input)
    return baseSolution(input) //May have more than 1 base cases
  else   //Recursive case
    input1, input2, … ← divideInput(input)
    solution1 ← recursiveAlgorithm(input1)
    solution2 ← recursiveAlgorithm(input2)
    …
    solution ← assembleSolutions(solution1, solution2, …)
    return solution
```

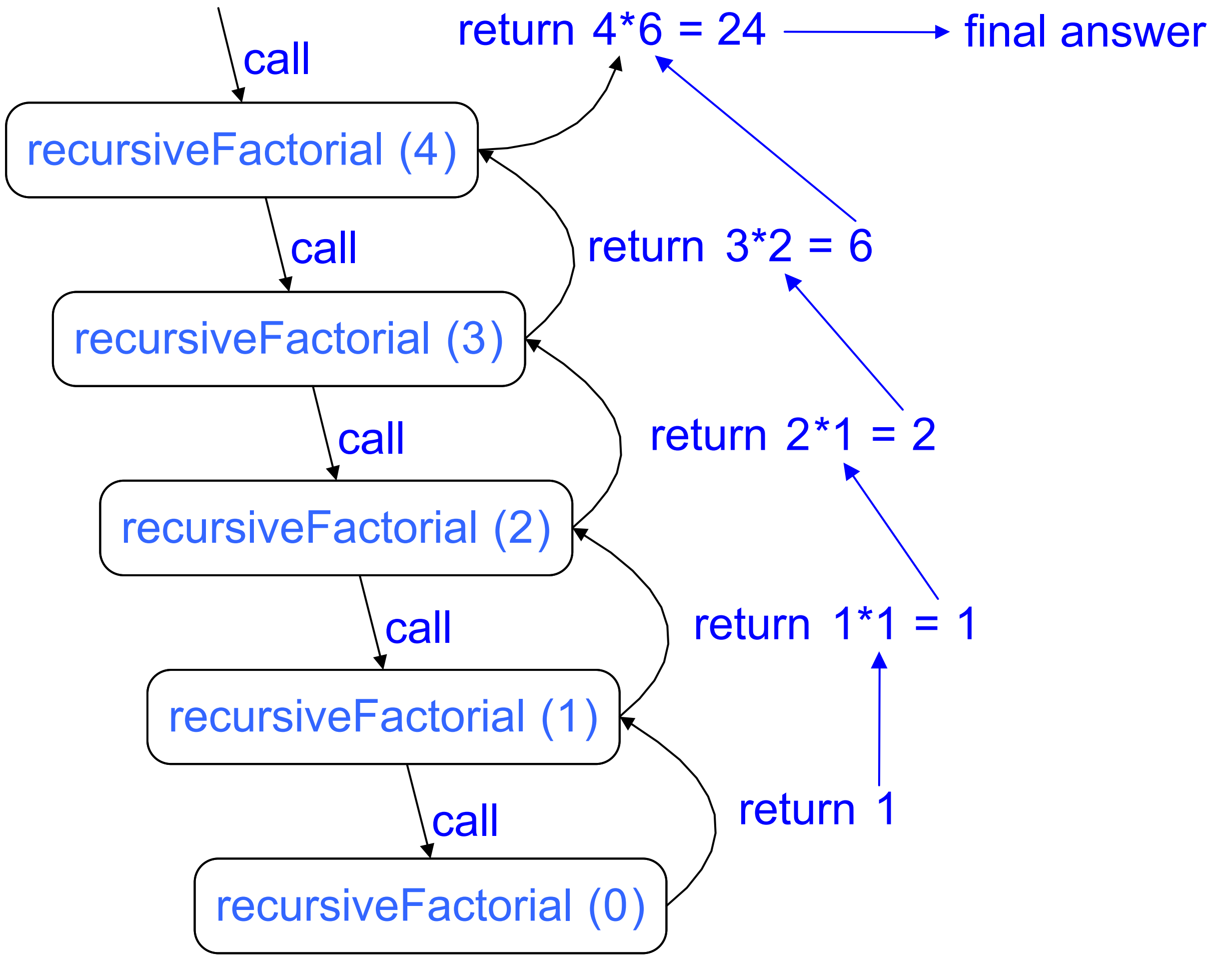
Three-Question Verification for Recursive Algorithms

- **Base-Case Question:**
  - Is there a non-recursive way out of the function?
  - Is the base case solution correct?
- **Smaller-Caller Question:**
  - Does each recursive function call involve a smaller case of the original problem?
  - Is this leading to the base case?
- **General-Case Question:**
  - Assuming each recursive call works correctly, does the whole function work correctly?

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
- How would this look like for the recursive factorial function?

`recursiveFactorial(4)`



Linear Recursion

- Nothing special about the base case, may have one or more base-cases.
- As always, each recursive call must make progress towards one of the base cases and eventually reach one

- In the recursive cases, the function only calls itself once per run
- May have a test that decides which of several possible recursive calls to make, but it should <u>ultimately make just one of these calls</u>

Linear Recursion Example

- Linear Sum: Sum the entries of an array

```
def linearSum(A, n):
    if n < 1:
        return 0
    else
        return linearSum(A, n-1)+ A[n-1]
```

Side note: Did we really need to have **n** as an input parameter?

Reversing an Array

```
def reverseArray(A, start, end)
    if start < end :
        A[start], A[end] = A[end], A[start] #Swapping
        reverseArray(A, start+1, end-1)


#First call
reverseArray(A,0,len(A)-1)
```

Side Note: Base case?

Defining Arguments for Recursion

- In creating recursive methods, it is important to <u>define the methods in ways that facilitate recursion</u>.
- This sometimes requires we define additional parameters that are passed to the method.
- For example, we defined the array reversal method as `reverseArray(A,start,end)`, not `reverseArray(A)`
    - Note that the latter was also possible to do due to slicing in Python

Binary Search

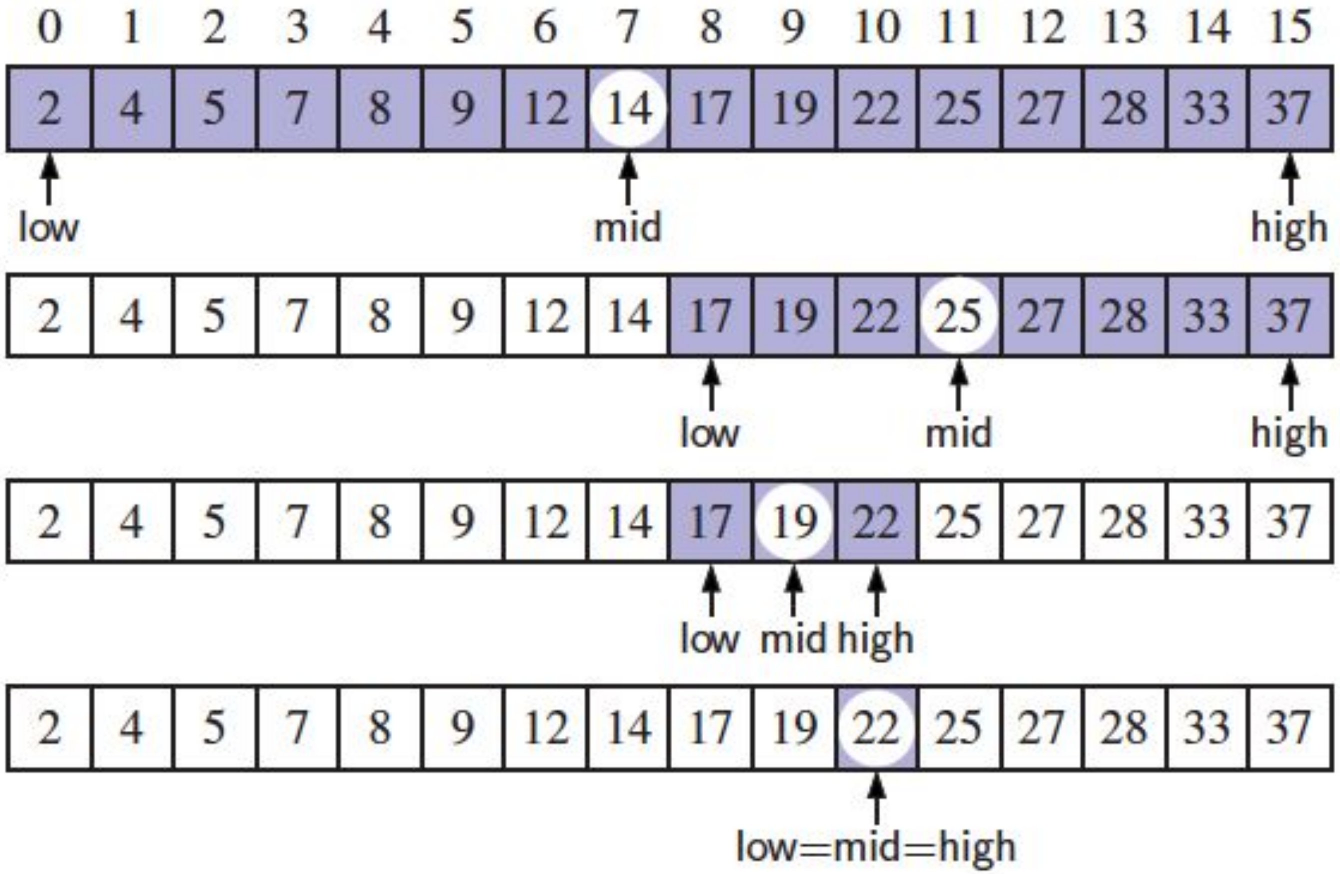
- Remember guess the number game. What was your strategy?
- What if the tables were reversed?

- The computer would need to search for an integer in an ordered list. Ideas?
- If the target equals lst[mid], then we have found the target.
- If target < lst[mid], then we recur on the first half of the sequence.
- If target > lst[mid], then we recur on the second half of the sequence.

Binary Search

- 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.

Find 22



Binary Search

```
def binarySearch(lst, target):
    if len(lst) == 0:
        return False
    else:
        mid = len(lst) // 2 #integer division
        if target == lst[mid]:
            return True
        elif target < lst[mid]:
            return binarySearch(lst[:mid], target)
        else:
            return binarySearch(lst[mid+1:], target)
```

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources, compilers sometimes do this by default). Example:

```
def IterativeReverseArray(A, i, j)
    while i < j:
        A[start], A[end] = A[end], A[start]
         start  += 1
         end  += 1
```

Binary Recursion

- Binary recursion: Two recursive calls for each non-base case.

- Example: Add all numbers in an integer array

```
def BinarySum(A, i, n):
    n = len(A)
    if n == 1:
        return A[0]
    # Two recursive calls, integer division
    return BinarySum(A[:n//2]) + BinarySum(A[n//2:])
```

Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

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

- Recursive algorithm (first attempt):

```
def BinaryFib(k):
    if k <= 1:
        return k
    else
        return BinaryFib(k - 1) + BinaryFib(k - 2)
```

A Better Algorithm

- No need to repeat the work! We can have a linearly recursive algorithm by returning two numbers instead of one

```
def LinearFibonacciHelper(k):
    if k == 1:
        return (1, 0)
    else:
        (i, j)  = LinearFibonacciHelper(k - 1)
        return (i+j, i)

def LinearFibonacci (k):
    if k == 0:
        return 0
    else:
        (i, j)  = LinearFibonacciHelper(k)
        return i+j
```

Indirect Recursion

- **Direct Recursion**: The function calls itself
  - Function foo(…) has a call to foo(…)
  - Previous examples was of this type!

- **Indirect Recursion**: The function calls another function which in turn calls the first function
  - Function foo(…) calls function bar(…) which in turn calls foo(…)
  - Also called mutual recursion
  - Not restricted to two functions, can have a longer chain

An (Contrived) Indirect Recursion Example

```
def isEven(n)
    if n = 0
        return True
    else
        return isOdd(n-1)


def isOdd(n)
    if n = 0
        return False
    else
        return isEven(n-1)
```

Keeping State

- Sometimes you need to keep the state of your problem
- For example:
  - Current depth in recursion
  - Some cumulative value
- Preferable way is to pass along a variable in recursion
- However, global variables can also be used here (with extreme caution, often times there is a better alternative)