ECS 32B - Recursion

Aaron Kaloti

UC Davis - Summer Session #2 2020

Overview

- **recursion**: a function calls itself.
  - A function that calls itself *recurses* or is *recursive*.
- Alternative to iteration (loops).
  - Sometimes makes problem easier or cleaner to solve.

Example: Calculate List's Sum[1]

Iterative Implementation

```
def listsum(numList):
    theSum = 0
    for i in numList:
        theSum = theSum + i
    return theSum
```

Recursive Implementation

```
def listsum(numList):
   if len(numList) == 1:
        return numList[0]
   else:
        return numList[0] + listsum(numList[1:])
```

1. From section 5.3 of primary textbook.

Example: Calculate List's Sum

Recursive Implementation

```
def listsum(numList):
   if len(numList) == 1:
        return numList[0]
   else:
        return numList[0] + listsum(numList[1:])
```

Step-by-Step Example

`listsum([1,3,5,7,9])`

= `1 + listsum([3,5,7,9])`

= `1 + 3 + listsum([5,7,9])`

= `1 + 3 + 5 + listsum([7,9])`

= `1 + 3 + 5 + 7 + listsum([9])`

= `1 + 3 + 5 + 7 + 9`

Two Important Parts of Recursive Function

1. Base case(s).
2. Something that brings us closer to the base case.
    - Prevents infinite recursion.

Recursive Implementation

```
def listsum(numList):
   if len(numList) == 1:
        return numList[0]
   else:
        return numList[0] + listsum(numList[1:])
```

Infinite Recursion

Example

```
def foo():
    print("Hi")
    foo()

foo()
```

Output

```
Hi
Hi
Hi
Hi
Hi
Hi
Hi
Hi
Hi
Hi
Hi
Hi
Hi
... and so on ...
```

Example: Factorial

Iterative Implementation

```
def fact(n):
    prod = 1
    while n > 1:
        prod *= n
        n -= 1
    return prod
```

Recursive Implementation

```
def fact(n):
    if n == 0 or n == 1:
        return 1
    return n * fact(n - 1)
```

Output

```
>>> fact(3)
6
>>> fact(5)
120
```

Example: Prompt until Valid Integer

Iterative Implementation

```
def get_int():
    while True:
        try:
            val = int(input("Enter integer: "))
            return val
        except:
            print("That's not an integer.")
```

Recursive Implementation

```
def get_int():
    try:
        val = int(input("Enter integer: "))
        return val
    except:
        print("That's not an integer.")
        return get_int()
```

Output

```
>>> get_int()
Enter integer: 5
5
>>> get_int()
Enter integer: blah
That's not an integer.
Enter integer: hi there
That's not an integer.
Enter integer: 33
33
```

Maintaining Local Variables

Example

```
def foo(n, k):
    if n == 1:
        return
    print("n={}, k={}".format(n,k))
    foo(n - 1, k - 2)
    print("n={}, k={}".format(n,k))

foo(5, 3)
```

```
n=5, k=3
n=4, k=1
n=3, k=-1
n=2, k=-3
n=2, k=-3
n=3, k=-1
n=4, k=1
n=5, k=3
```

Maintaining Local Variables

Activation Stack[1]

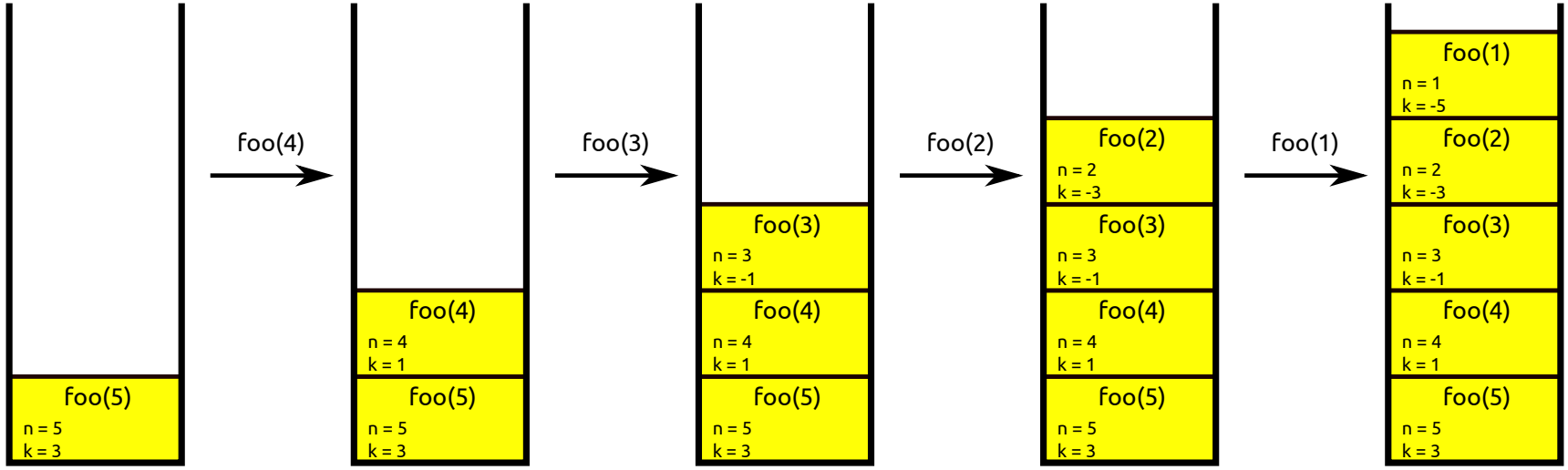
- Can think of "state" (*activation record*) of an unfinished function call as being in a stack.

Example #1

```
def foo(n, k):
    if n == 1:
        return
    print("n={}, k={}".format(n,k))
    foo(n - 1, k - 2)
    print("n={}, k={}".format(n,k))

foo(5, 3)
```

```
n=5, k=3
n=4, k=1
n=3, k=-1
n=2, k=-3
n=2, k=-3
n=3, k=-1
n=4, k=1
n=5, k=3
```

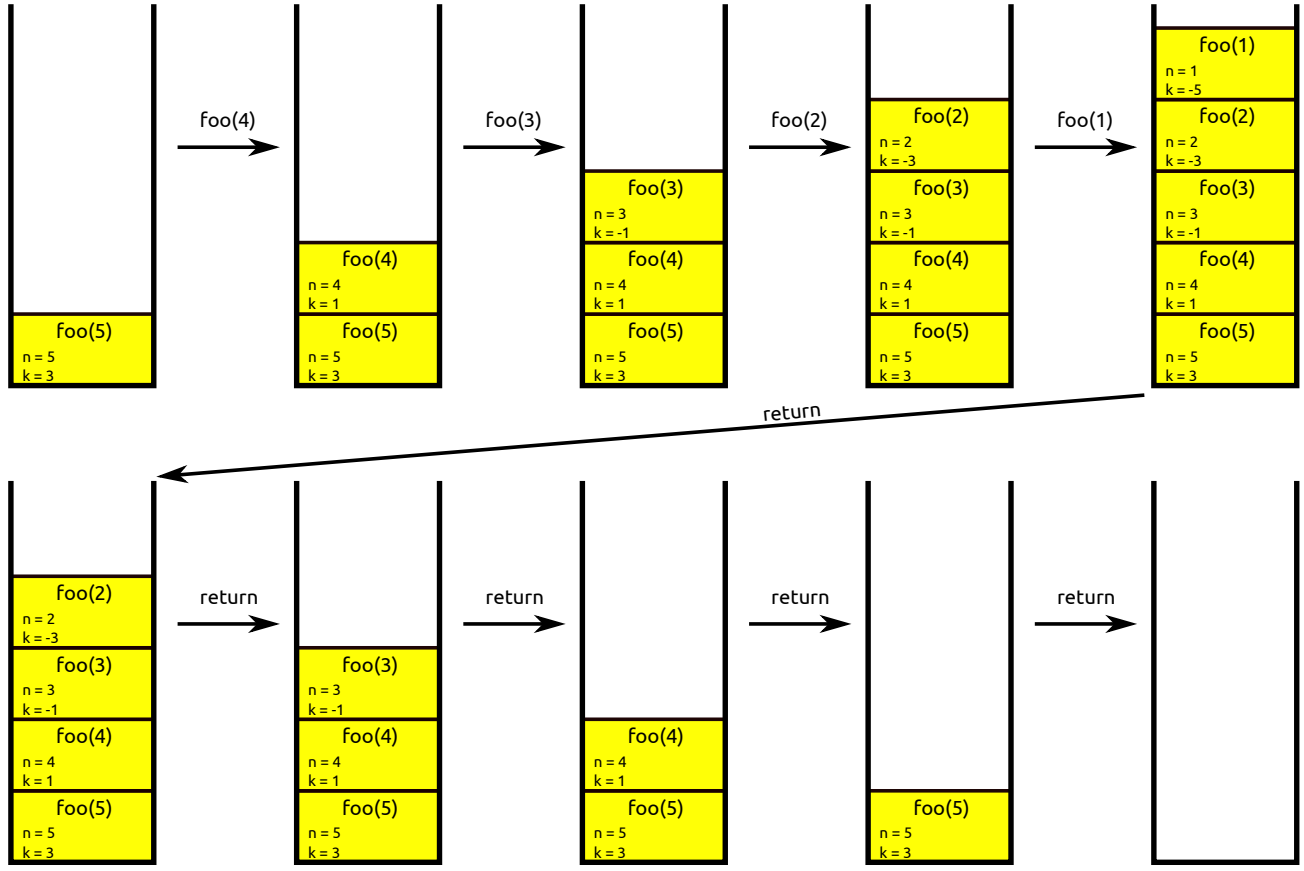


1. Could call it a "call stack" too. More details to come in ECS 50!

Maintaining Local Variables

Activation Stack

Example #1: The Entirety




```
n=5, k=3
n=4, k=1
n=3, k=-1
n=2, k=-3
n=2, k=-3
n=3, k=-1
n=4, k=1
n=5, k=3
```

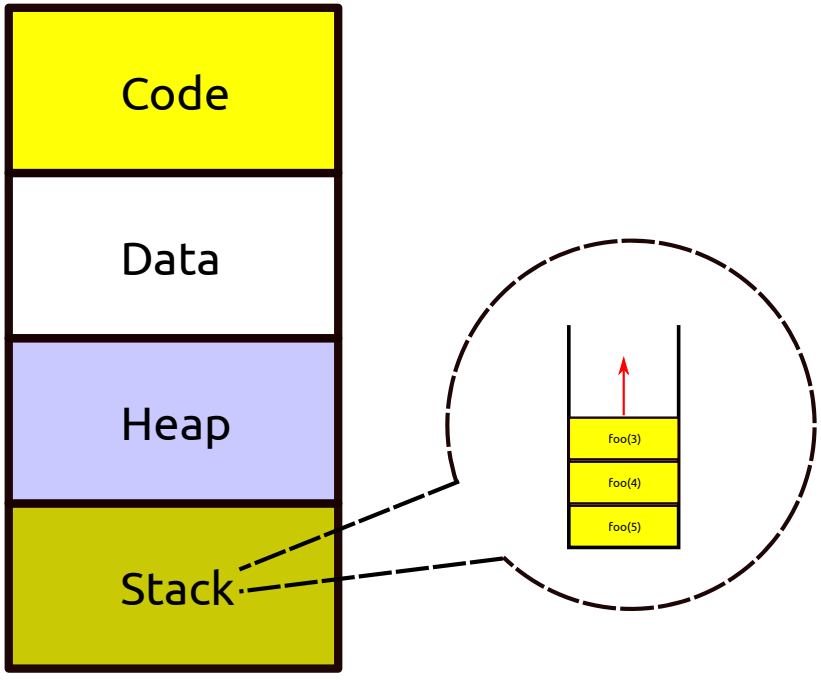
```
def foo(n, k):
    if n == 1:
        return
    print("n={}, k={}".format(n,k))
    foo(n - 1, k - 2)
    print("n={}, k={}".format(n,k))
```

Maintaining Local Variables

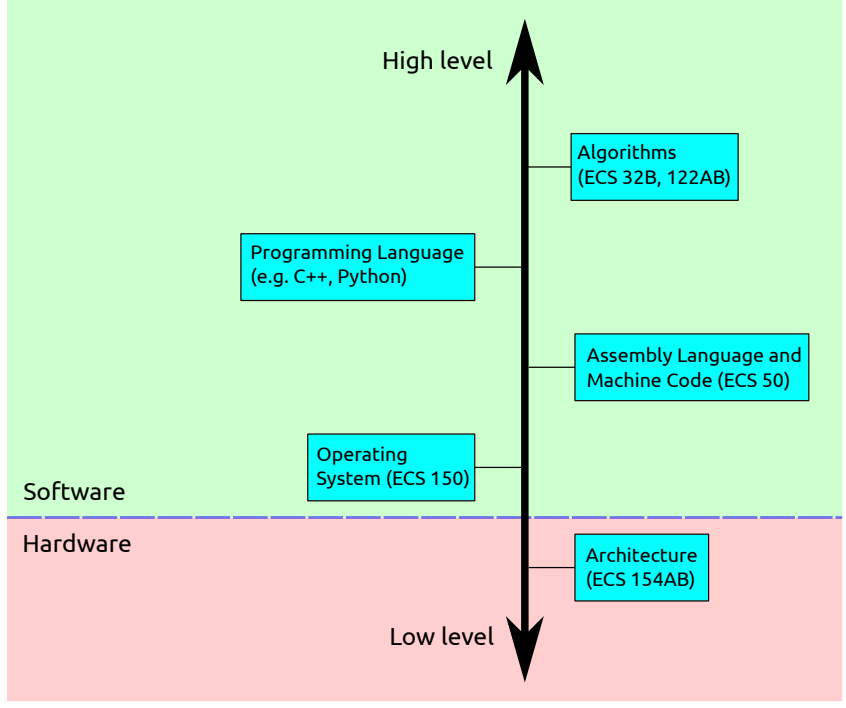
Activation Stack - Perspective of Operating System (OS)

- Activation stack is implicitly managed by your program and the operating system; you needn't worry about it.

Layout of Process Memory



Hardware/Software Stack



Example: Linear Search

Iterative Implementation

```
def lin_search(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return True
    return False
```

Example #1

`lin_search([14, -7, 15, 10], 15)`

= `lin_search([-7, 15, 10], 15)`

= `lin_search([15, 10], 15)`

= `True`

Recursive Implementation

```
def lin_search(lst, target):
    if len(lst) == 0:
        return False
    elif lst[0] == target:
        return True
    else:
        return lin_search(lst[1:],
                          target)
```

Example #2

`lin_search([18, 2, -3, 14, 7], -15)`

= `lin_search([2, -3, 14, 7], -15)`

= `lin_search([-3, 14, 7], -15)`

= `lin_search([14, 7], -15)`

= `lin_search([7], -15)`

= `lin_search([], -15)`

= `False`

Example: Binary Search

Iterative Implementation

```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    return found
```

Example: Binary Search

Recursive Implementation #1

```
def bin_search_rec(alist, item):
    if len(alist) == 0:
        return False
    first = 0
    last = len(alist) - 1
    midpoint = (first + last) // 2
    if alist[midpoint] == item:
        return True
    else:
        if item < alist[midpoint]:
            return bin_search_rec(alist[:midpoint], item)
        else:
            return bin_search_rec(alist[midpoint+1:], item)
```

Example

`bin_search_rec([8, 20, 35, 42, 44, 57, 60, 64, 100, 101, 105], 64)`

= `bin_search_rec([60, 64, 100, 101, 105], 64)`

= `bin_search_rec([60, 64], 64)`

= `bin_search_rec([64], 64)`

= `True`

Example: Binary Search

Recursive Implementation #2: More for C/C++[1]

```
def bin_search_rec(alist, item):
    return bin_search_aux(alist, item, 0, len(alist)-1)

def bin_search_aux(alist, item, first, last):  # @end is included
    if first > last:
        return False
    midpoint = (first + last) // 2
    if alist[midpoint] == item:
        return True
    else:
        if item < alist[midpoint]:
            return bin_search_aux(alist, item, first, midpoint-1)
        else:
            return bin_search_aux(alist, item, midpoint+1, last)
```

```
bin_search_rec([8, 20, 35, 42, 44, 57, 60, 64, 100, 101, 105], 64)
= bin_search_aux([8, 20, 35, 42, 44, 57, 60, 64, 100, 101, 105], 64, 0, 10)
= bin_search_aux([8, 20, 35, 42, 44, 57, 60, 64, 100, 101, 105], 64, 6, 10)
= bin_search_aux([8, 20, 35, 42, 44, 57, 60, 64, 100, 101, 105], 64, 6, 7)
= bin_search_aux([8, 20, 35, 42, 44, 57, 60, 64, 100, 101, 105], 64, 7, 7)
= True
```

1. These languages don't support splicing.

References / Further Reading

- **Primary textbook**: Chapter 5 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.
- Topics we won't cover:
  - Tail recursion.
  - Divide-and-conquer. (ECS 122A)
  - Dynamic programming. (ECS 122A)