CS240
Fall 2014

Mike Lam, Professor

recursion

n. [ri-**kur-**zh*uh* n]

1. See “recursion”

Recursion

Recursion

- The expression of a problem solution in a way that depends on solutions to smaller instances of the same problem
- For some problems, a recursive solution is cleaner than the corresponding iterative solution
- Classics:
  - A **list** is either 1) an “empty list” or 2) an item followed by a **list**
  - **fact**(n) = 1 *if n ≤ 1*, n * **fact**(n-1) *if n>1*
  - Tower of Hanoi / Brahma

Recursion

- The language runtime handles the actual semantics of recursive behavior
- Usually, it tracks recursive calls using a stack
- Every function call pushes a new entry (called an “activation record” or “frame”) to the stack
- A record is popped when a function returns, and execution returns to the function on the top of the stack

Recursion

- "Call stack"
- Details are machine- and language-dependent
- More info in CS430

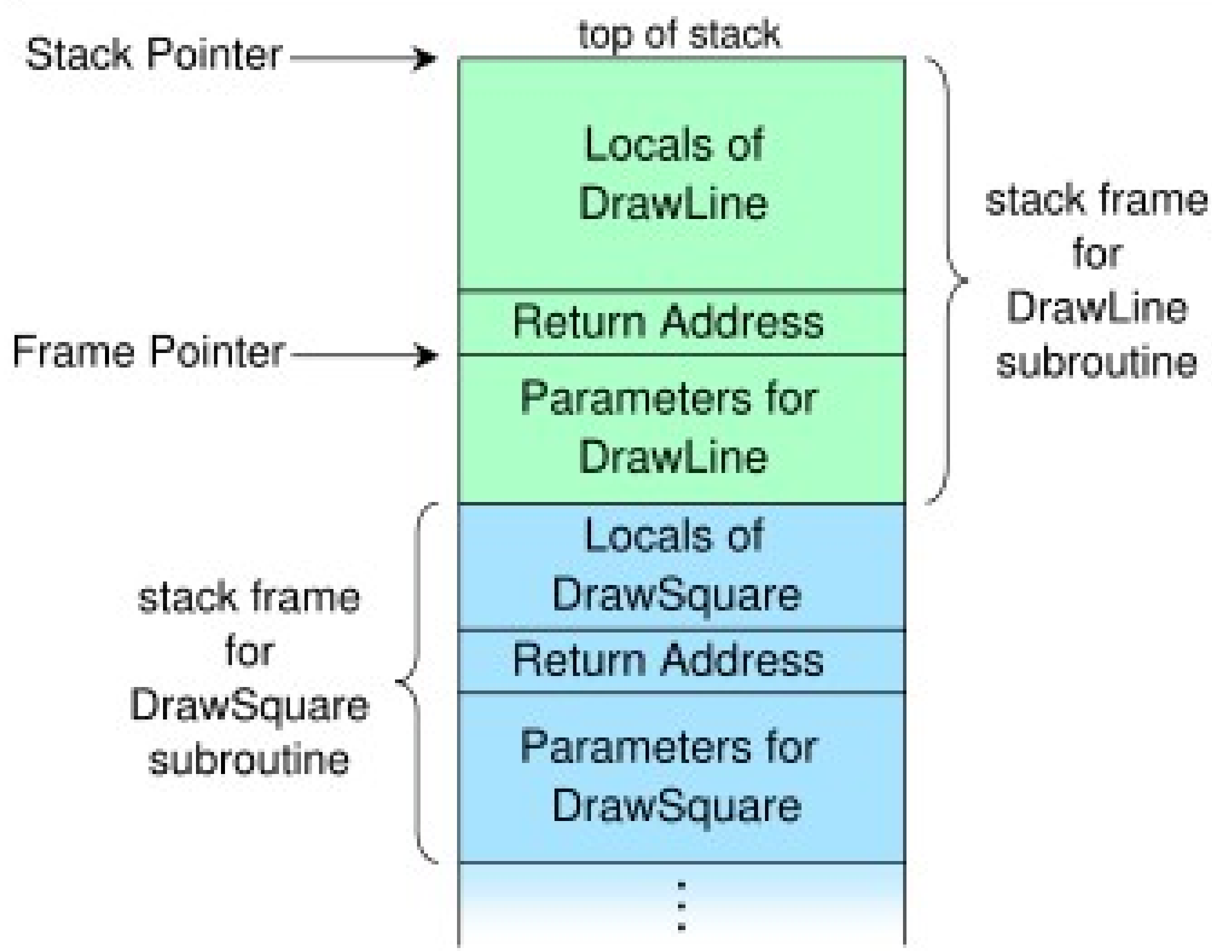


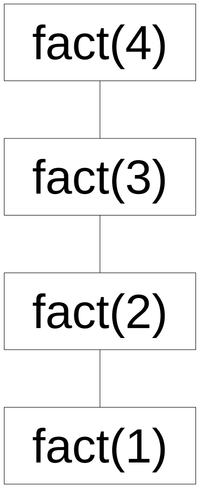
Image from Wikipedia article "Call Stack"

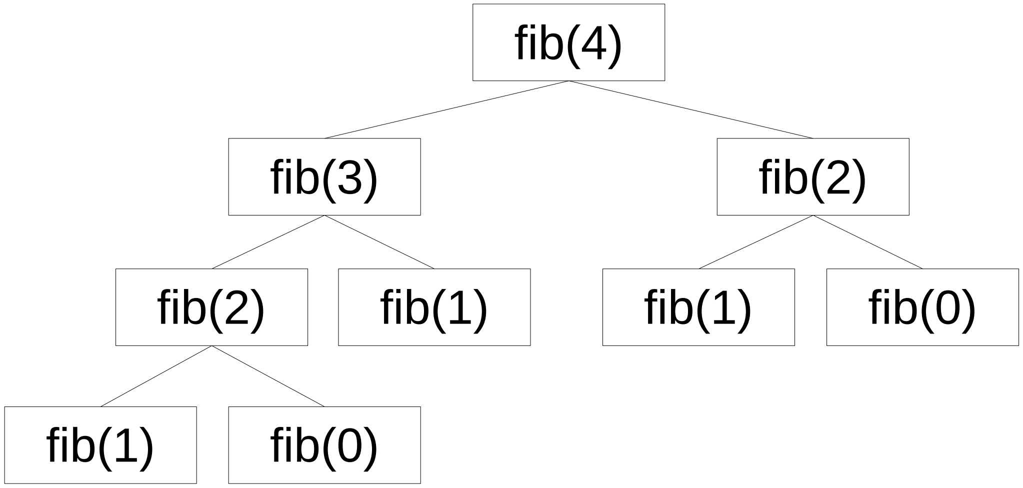
Recursion

- Single vs. binary vs. multiple recursion
  - **fact**(n) = 1 *if* $n \leq 1$, n ***fact**(n-1) *if* $n > 1$
  - **fib**(n) = 1 *if* $n \leq 1$, **fib**(n-1) +**fib**(n-2) *if* $n > 1$
- Trace: fact(4) vs. fib(4)

Recursion

- Single vs. binary vs. multiple recursion
  - **fact**(n) = 1 *if* $n \leq 1$, n ***fact**(n-1) *if* $n > 1$
  - **fib**(n) = 1 *if* $n \leq 1$, **fib**(n-1) +**fib**(n-2) *if* $n > 1$
- Trace: fact(4) vs. fib(4)





Binary Search

```
def find(array, item):
    return helper(array, item, 0, len(array))

def helper(array, item, left, right):
    mid = (right-left)//2 + left
    if array[mid] > item:
        return helper(array, item, left, mid)
    elif array[mid] < item:
        return helper(array, item, mid+1, right)
    else:
        return mid < len(array) and array[mid] == item
```

Binary Search

- Trace: find([1,4,5,7,9,11,15], 5)

Binary Search

- Trace: find([1,4,5,7,9,11,15], 5)

left = 0
right = 7

Binary Search

- Trace: find([1,4,5,7,9,11,15], 5)

left = 0
right = 7

mid = 3

[1, 4, 5, 7, 9, 11, 15]

Binary Search

- Trace: find([1,4,5,7,9,11,15], 5)

left = 0
right = 7

mid = 3

[1, 4, 5, 7, 9, 11, 15]

[1, 4, 5, 7, 9, 11, 15]

left = 0
right = 3

Binary Search

- Trace: find([1,4,5,7,9,11,15], 5)

left = 0
right = 7

mid = 3

[1, 4, 5, 7, 9, 11, 15]

[1, 4, 5, 7, 9, 11, 15]

left = 0
right = 3

mid = 1

[1, 4, 5, 7, 9, 11, 15]

Binary Search

- Trace: find([1,4,5,7,9,11,15], 5)

left = 0
right = 7

mid = 3 [1, 4, 5, 7, 9, 11, 15]

[1, 4, 5, 7, 9, 11, 15]

left = 0
right = 3

mid = 1 [1, 4, 5, 7, 9, 11, 15]

[1, 4, 5, 7, 9, 11, 15]

left = 2
right = 3

Binary Search

- Trace: find([1,4,5,7,9,11,15], 5)

left = 0
right = 7

mid = 3

[1, 4, 5, 7, 9, 11, 15]

[1, 4, 5, 7, 9, 11, 15]

left = 0
right = 3

mid = 1

[1, 4, 5, 7, 9, 11, 15]

[1, 4, 5, 7, 9, 11, 15]

left = 2
right = 3

mid = 2

[1, 4, 5, 7, 9, 11, 15]

A note on the word "binary"

- Binary search is not binary recursion!
  - Only recurses on one half of the list
  - So it is single recursion
- Binary sum is binary recursion
  - Recurses on both sides of the list

Binary Search

- What is the running time of a binary search?

Binary Search

- What is the running time of a binary search?
- Need a way to express recursion costs mathematically
- Write a function!
  - Express T(n) in terms of itself

Binary Search

- What is the running time of a binary search?
- Need a way to express recursion mathematically
- Write a function!
  - Express $T(n)$ in terms of itself
- For binary search: $T(n) = 1 + T(n/2)$
  - To search n items, do one comparison then recurse on the appropriate half-list

Recurrences

- Recursive formulas are called “recurrences”
- We still want to find a “closed-form” descriptio
  - Something like “2^n” or “$\log_2 n$” or “$5n^2$”
- We will talk more on Wednesday about how to solve recurrences
- But first, we need to be comfortable tracing recursive code

Exercise 1

- Given the following code:

```
def foo(n):
    if n < 2:
        return 1
    else:
        return n * foo(n-1)
```

- Trace the following call:

```
print(str(foo(4)))
```

Exercise 2

- Given the following code:

```
def bar(text):
    if len(text) <= 1:
        return True
    return text[0] == text[-1] and
           bar(text[1:-1])
```

- Trace the following call:

```
print(str(bar("abbaba")))
```

Exercise 3

- Given the following code:

```
def baz(x, n):
    if n == 0:
        return 1
    y = baz(x, n//2)
    if n % 2 == 1:
        return x * y * y
    else:
        return y * y
```

- Trace the following call:

```
print(str(baz(2, 10)))
```

Exercise 4

- Given the following code:

```
def hanoi(n, src, dst, tmp):
    if n == 1:
        print("move from " + str(src) +
              " to " + str(dst))
    else:
        hanoi(n-1, src, tmp, dst)
        hanoi(  1, src, dst, tmp)
        hanoi(n-1, tmp, dst, src)
```

- Trace the following call:

```
hanoi(3, “a”, “c”, “b”)
```