CSC148H Week 5

February 2, 2015

Announcements

- Assignment 1
  - Due this Wednesday at 22:00
  - Submit using MarkUs (link posted on Portal and Piazza)
- Term Test 1
  - Next Friday at 17:00 in IB110
- Office Hours
  - DH3097 B

Maximum Element

```
def maximum_element(xs):
    '''(list) -> int
    Find the maximum element in a list of integers.
    The list does not contain any nested lists.

    >>> maximum_element([5, 1, 2])
    5

    >>> maximum_element([5, 1012, 21])
    1012
    '''
```

Maximum Element

```
def maximum_element(x):
    if len(x)==0:
        raise Exception("empty list")
    elif len(x)==1:
        return x[0]
    else:
        return max(x[0], maximum_element(x[1:]))
```

Reversing Strings

```
def reverse_string(xs):
    '''(str) -> str
    Get the reverse of a given string.

    >>> reverse_string(Hello World)
    dlroW olleH

    >>> reverse_string(I love Toronto)
    otnoroT evol I
    '''
```

Reversing Strings

```
def reverse_string(x):
    if len(x)==0:
        raise Exception("Empty list")
    if len(x)==1:
        return x[0]
    else:
        return x[-1] + reverse_string(x[:-1])
```

Zip Lists

```
def zip_lists(x, y):
    '''(list, list) -> tuple
    Zip up two lists into a list of tuples

    >>> zip_lists([1,2,3], [4,5,6])
    [(1, 4), (2, 5), (3, 6)]

    >>> zipL_lists([1,2,3], [4,5,6,9,10,11])
    [(1, 4), (2, 5), (3, 6)]

    >>> zip_lists([1,2,3,100,101,102], [4,5,6,9,10,11])
    [(1, 4), (2, 5), (3, 6), (100, 9), (101, 10), (102, 11)]

    >>> zip_lists([Jack','Matt','Jane'], [45,51,68])
    [('Jack', 45), ('Matt', 51), ('Jane', 68)]
    '''
```

Zip Lists

```
def zip_lists(x, y):
    if len(x)==0 or len(y)==0:
        return []
    return [(x[0],y[0])] + zip_lists(x[1:],y[1:])
```

Power Function

```
def power(x, n):
    '''(int, int) -> int
    Computing the power function.

    >>> power(2, 3)
    8
    >>> power(4, 2)
    16
    >>> power(5, 0)
    1
    '''
```

Power Function

```
def power(x, n):
    if n==0:
        return 1
    else:
        return x*power(x, n-1)
```

Tracing Recursion

The recommended way to trace a recursive function:

- Trace the simplest (non-recursive) case
- Trace the next-most complex case; **plug in known results**
- Keep doing this until you trace the required (most-complex) case

Tracing Recursion...

- What is the base case?
- What is the recursive structure?

```
def rec_max(lst):
    '''(list of int, can be nested) -> int
    Return max number in possibly nested list of numbers.

    >>> rec_max([17, 21, 0])
    21
    >>> rec_max([17, [21, 24], 0])
    24
    '''
```

Tracing Recursion, Base Case

- What is the base case?
- What is the recursive structure?

```
def rec_max(lst):
    '''(list of int, can be nested) -> int
    Return max number in possibly nested list of numbers.

    >>> rec_max([17, 21, 0])
    21
    >>> rec_max([17, [21, 24], 0])
    24
    '''
    nums = []
    for element in lst:
        if isinstance(element, int):
            nums.append(element)
    return max(nums)
```

Tracing Recursion...

```
def rec_max(lst):
    '''(list of int, can be nested) -> int
    Return max number in possibly nested list of numbers.

    >>> rec_max([17, 21, 0])
    21
    >>> rec_max([17, [21, 24], 0])
    24
    '''
    nums = []
    for element in lst:
        if isinstance(element, int):
            nums.append(element)
        else:
            nums.append(rec_max(element))
    return max(nums)
```

Tracing Recursion...

- On each recursive call, the current call is suspended and a new copy of the function is executed with its own local variables
- The function executions work as a stack of function calls
- The last function call in is the first one out
- Let's trace the `rec_max` example as Python would run it ...

Tracing Recursion...

```
def rec_max(lst):
    '''(list of int, can be nested) -> int
    Return max number in possibly nested list of numbers.

    >>> rec_max([17, 21, 0])
    21
    >>> rec_max([17, [21, 24], 0])

    24
    '''
    nums = []


    for element in lst:

        if isinstance(element, int):

            nums.append(element)

        else:

            nums.append(rec_max(element))

    return max(nums)
```

Evaluate when lst is [17, [21, 24], 0]

rec_max([17, [21, 24], 0])

[17, [21, 24], 0]

```
def rec_max(lst):
    '''(list of int, can be nested) -> int
    Return max number in possibly nested list of numbers.

    >>> rec_max([17, 21, 0])
    21
    >>> rec_max([17, [21, 24], 0])

    24
    '''
    nums = []
```

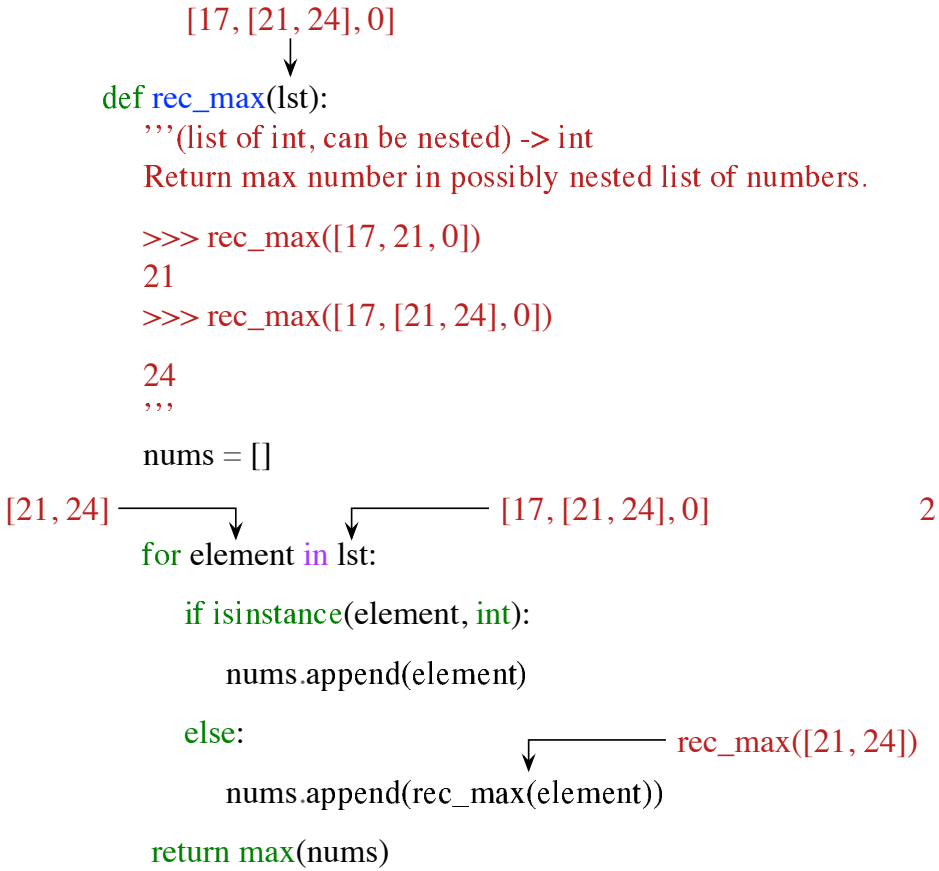
17 ← for element in lst: ← [17, [21, 24], 0]

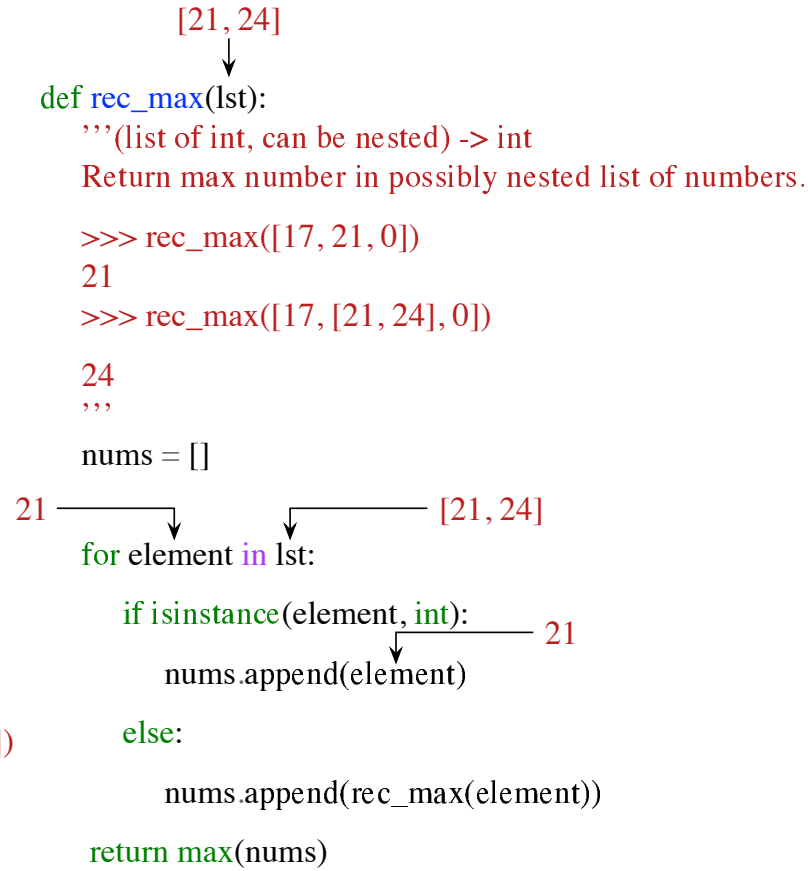
```
        if isinstance(element, int):
```

17 ↓

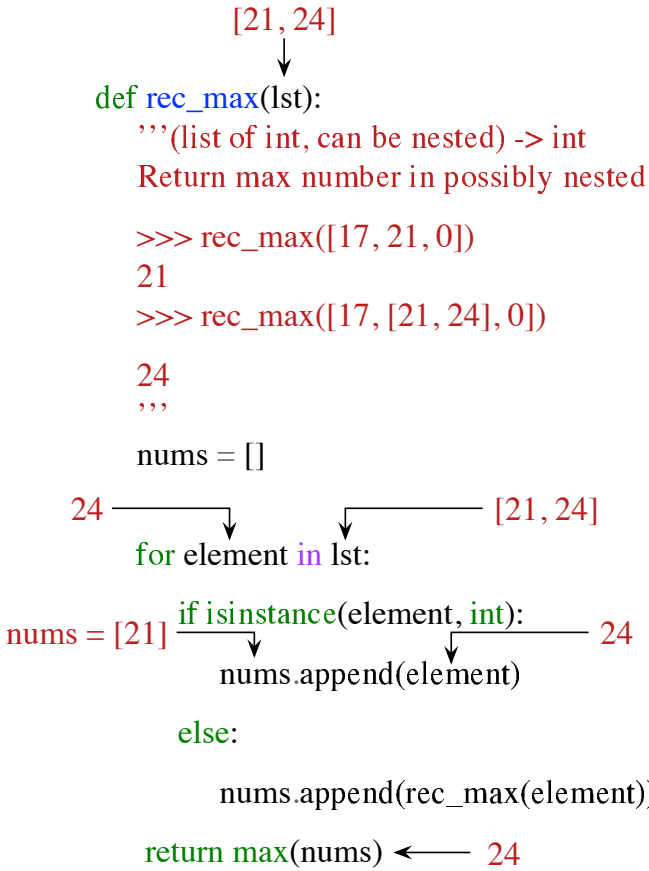
```
            nums.append(element)

        else:

            nums.append(rec_max(element))

    return max(nums)
```

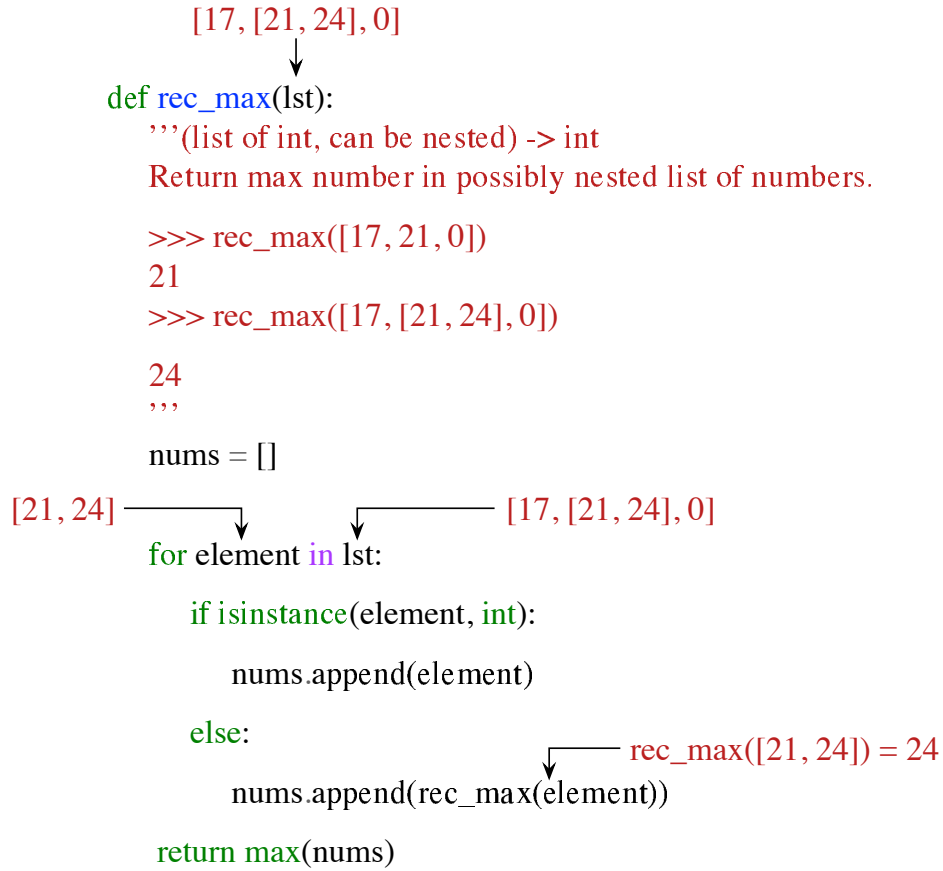
Tracing Recursion...



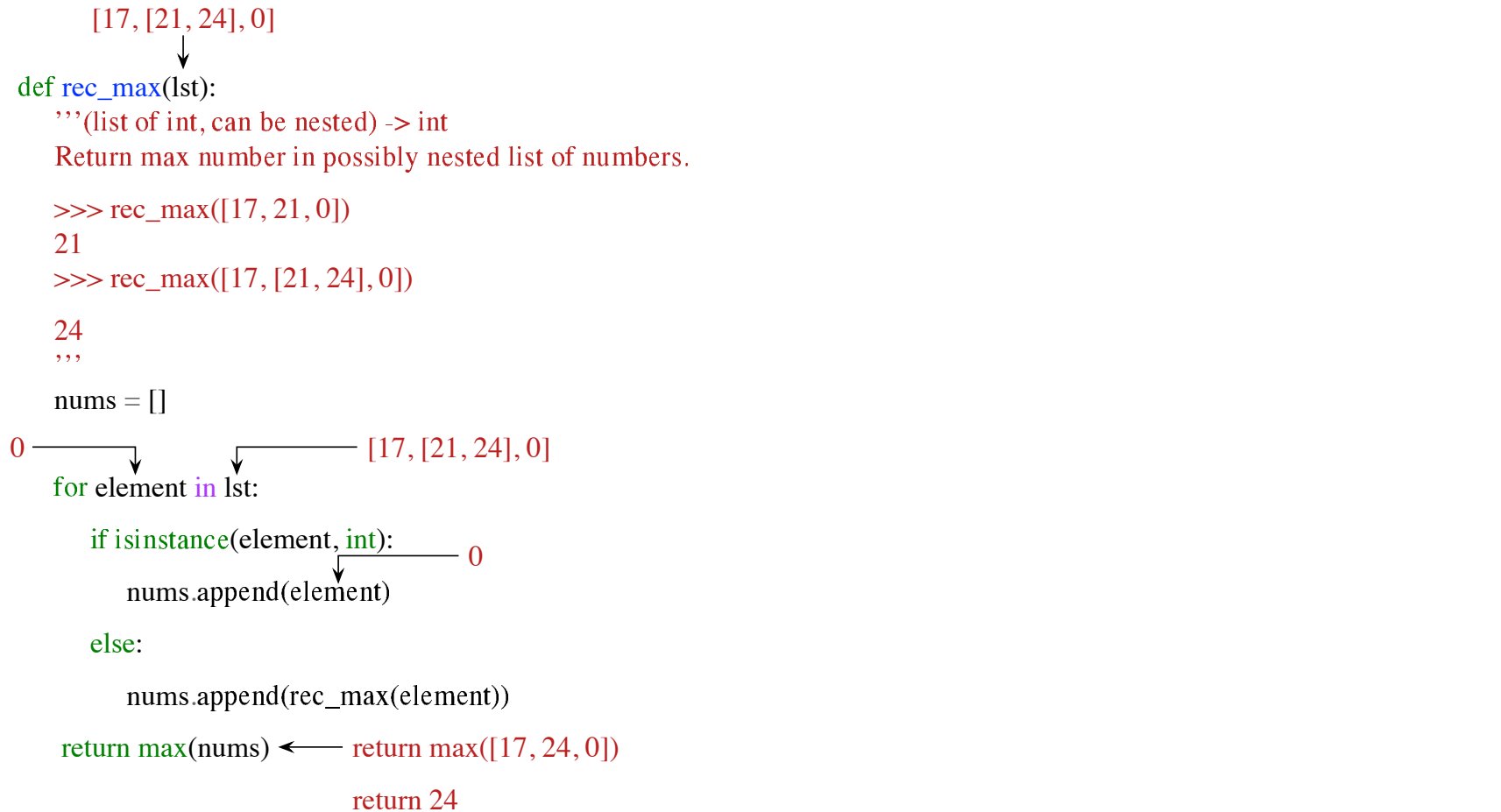


Tracing Recursion...

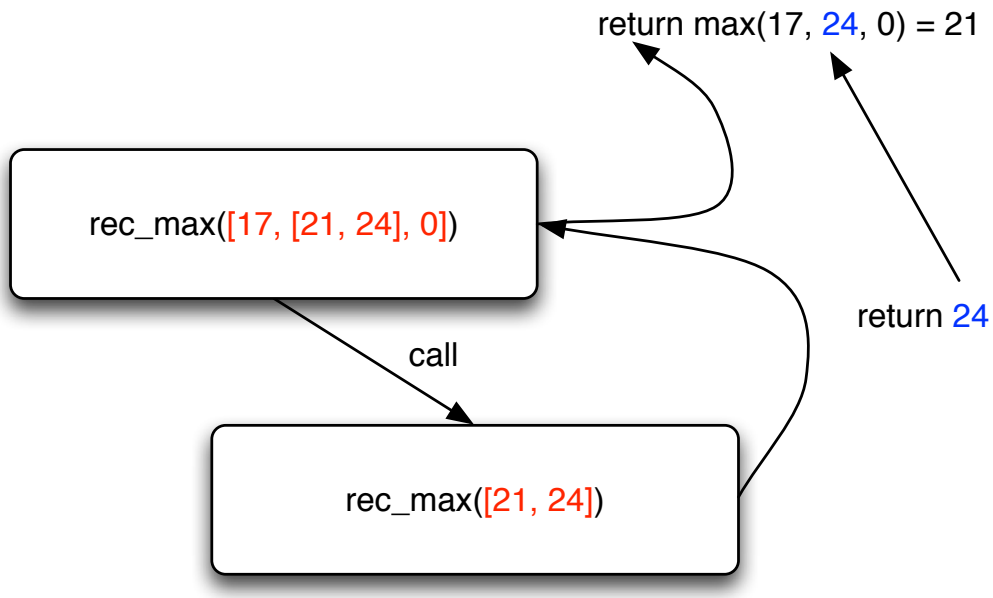




Tracing Recursion...



Tracing Recursion...



Binary Recursion

- Describes a function that makes two recursive calls.
- With two or more elements, we can recursively compute the sum of the first half, and the sum of the second half, and add these sums together.

```
def binary_sum(S,start,stop):
    ''' (list, int, int) -> int

    >>> binary_sum([1, 2, 3, 4, 5, 6], 0, len([1, 2, 3, 4, 5, 6])
    21
    '''
```

Binary Recursion

```
def binary_sum(S, start, stop):
    '''
    Calculate the total sum of a
    given list of integers between
    a start and stop value.
    '''
    if start >= stop:
        raise Exception ("start is greater than stop")
    elif start == stop-1:
        return S[start]
    else:
        # // is floor division, discard the remainder
        mid = (start + stop) // 2
    return binary_sum(S, start, mid) + binary_sum(S, mid, stop)
```

Guessing Numbers

- Let's say I pick a number randomly from 1 to 100
- If you want to determine the number in as few guesses as possible, what strategy should you employ?
- If you want to guess all day, you can use a linear search strategy (guess each number from 1 to 100 in order)
- Improvement: you can always eliminate half the possible numbers by guessing the midpoint in the range of remaining possibilities.
- By eliminating half the remaining numbers on each guess, you can determine the number in no more than 7 steps.
- In general, if I'm thinking of a number from 1 to n, you can determine the number in no more than $\lceil \lg n \rceil$ steps

Guessing Numbers

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.

    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

Tracing Recursion...

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

Evaluate when low is 1, high is 100 and actual is 51.

guess(1, 100, 51)


```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2      <-- mid = (1 + 100) = 50
    if mid == actual:
        return [mid]
    elif mid < actual:                  guess(51, 100, 51)
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

Tracing Recursion...

51 100 51

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

mid = (51 + 100) = 75

guess(51, 74, 51)

51 74 51

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

mid = (51 + 74) = 62

guess(51, 61, 51)

Tracing Recursion...

51 61 51

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

mid = (51 + 61) = 56

guess(51, 55, 51)

51 55 51

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

mid = (51 + 55) = 53

guess(51, 52, 51)

Tracing Recursion...

51 52 51

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

mid = (51 + 52) = 51

return [51]

51 55 51

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

mid = (51 + 55) = 53

[53]

guess(51, 52, 51) = [51]

return [53, 51]

Tracing Recursion...

51 61 51

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

mid = (51 + 61) = 56

[56]

guess(51, 55, 51) = [53, 51]

return [56, 53, 51]

51 74 51

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

mid = (51 + 74) = 62

[62]

guess(51, 61, 51) = [56, 53, 51]

return [62, 56, 53, 51]

Tracing Recursion...

51 100 51

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

mid = (51 + 100) = 75

[75]

guess(51, 74, 51) = [62, 56, 53, 51]

return [75, 62, 56, 53, 51]

1 100 51

```
def guess(low, high, actual):
    '''(int, int, int) -> list of int
    Return list of numbers to guess the actual
    number in low..high.
    >>> guess(1, 100, 51)
    [50, 75, 62, 56, 53, 51]
    '''
    mid = (low + high) // 2
    if mid == actual:
        return [mid]
    elif mid < actual:
        return [mid] + guess(mid + 1, high, actual)
    else:
        return [mid] + guess(low, mid - 1, actual)
```

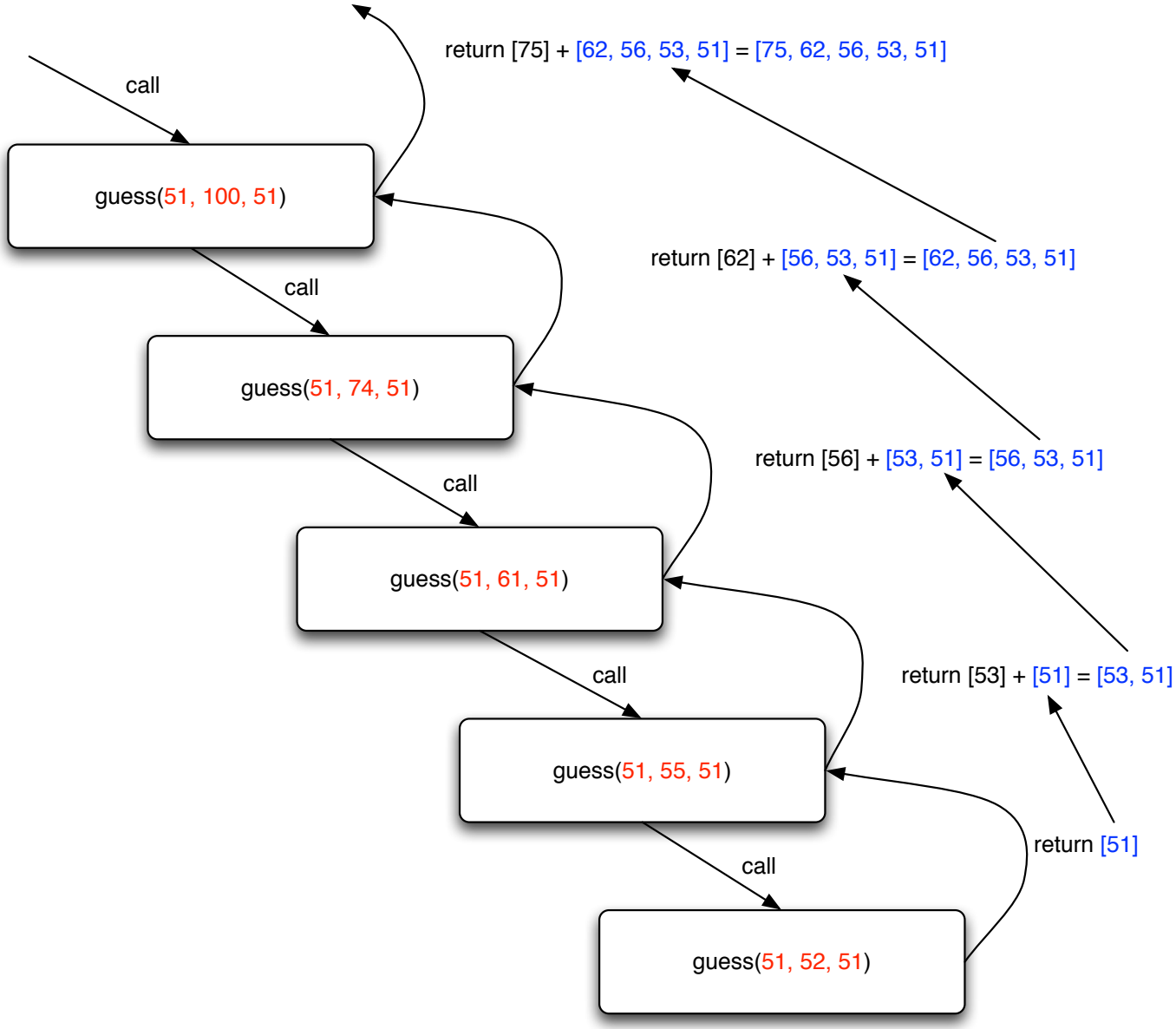
mid = (1 + 100) = 50

[50]

guess(51, 100, 51)

return [50, 75, 62, 56, 53, 51]

Tracing Recursion...



Binary Search

- We can apply this same idea when searching for an item in a sorted list.
- As with the guessing numbers game, we can eliminate about half of the list on each iteration
- The strategy is to find the midpoint of the current list
- We then compare our item with this midpoint item
- Based on this comparison, we know that our item can exist in only one of the two halves, so we recursively search there
- This yields an $O(\lg n)$ function, where n is the length of the list to search
- Running time is proportional to the number of recursive calls performed

Binary Search

- Used to efficiently locate a value in a **sorted** sequence of n elements
- When the sequence of n elements is **sorted** and **indexable**, we can use a much more efficient algorithm (binary search)
- When the sequence is unsorted, the standard approach is to search for the value using a loop (known as a sequential search algorithm)
  - This yields an $O(n)$ function (linear time), where n is the length of the list to search
  - The worst case scenario is that every element is searched

Binary Search, Basic Version

```
def binary_search(lst, s):
    '''(list of float) -> bool
    Return True iff s is found in lst.

    >>> binary_search([1, 2, 3, 4], 2)
    True
    '''
```

Binary Search, Updated Version

```
def binary_search(lst, s):
    '''(list of float) -> int
    Return the index of s in lst, or -1 if s is not in Lst.
    lst must be a sorted list
    '''
    #BASE CASE: If lst is empty, return -1
    if lst == []:
        return -1
```

Binary Search

```
def binary_search(lst, s):
    '''(list of float) -> int
    Return the index of s in lst, or -1 if s is not in Lst.
    lst must be a sorted list
    '''
    #BASE CASE: If lst is empty, return -1
    if lst == []:
        return -1

    #GENERAL CASE
    #get the middle value of lst. If it's larger than s, then s must be in
    #the first half of the list, so call binary_search on the first half.
    #Otherwise, call it on the second half of lst. if it's equal to s, then
    #we've found s and we can stop
```

Binary Search

```
def binary_search(lst, s):
    '''(list of float) -> int
    Return the index of s in lst, or -1 if s is not in Lst.
    lst must be a sorted list
    '''
    #BASE CASE: If lst is empty, return -1
    if lst == []:
        return -1
    #GENERAL CASE
    #get the middle value of lst. If it's larger than s, then s must be in
    #the first half of the list, so call binary_search on the first half.
    #Otherwise, call it on the second half of lst. if it's equal to s, then
    #we've found s and we can stop

    mid_index = len(lst)//2
    mid = lst[mid_index]

    if mid == s:
        #found it, return its index
        return mid_index
```

Binary Search

```
def binary_search(lst, s):
    '''(list of float) -> int
    Return the index of s in lst, or -1 if s is not in Lst.
    lst must be a sorted list
    '''
    #BASE CASE: If lst is empty, return -1
    if lst == []:
        return -1

    mid_index = len(lst)//2
    mid = lst[mid_index]

    if mid == s:
        #found it, return its index
        return mid_index
    elif mid > s:
        #if s is in lst, it must be in the first half of the list
        #so just perform a binary search on the first half of the list
        #and return that search's result
        return binary_search(lst[0:mid_index],s)
    else:
        #if s is in lst, it must be in the latter half of the list
        #so perform a binary search on the latter half of the list,
        #however, this time, if we do get a result, we have to return
        #its offset from our current midpoint
        result = binary_search(lst[mid_index+1:],s)
        if result == -1:
            #didn't find it, just pass on the bad news
            return result
        else:
            #found it, now we must add its index to our midpoint to get
            #the index in the whole list
            return result + mid_index + 1
```

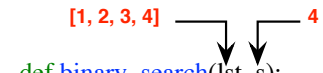
Binary Search

```
def binary_search(lst, s):
    '''(list of float) -> int
    Return the index of s in lst, or -1 if s is not in Lst.
    lst must be a sorted list
    '''
    if lst == []:    #base case
        return -1

    mid_index = len(lst)//2
    mid = lst[mid_index]

    if mid == s:
        return mid_index
    elif mid > s:
        return binary_search(lst[0:mid_index],s)
    else:
        result = binary_search(lst[mid_index+1:],s)
        if result == -1:
            return result
        else:
            return result + mid_index + 1
```

Binary Search

```
def binary_search(lst, s):
    '''(list of float) -> int

    Return the index of s in lst, or -1 if s is not in Lst.

    lst must be a sorted list
    '''


    if lst == []: #base case

        return -1


    mid_index = len(lst) // 2

    mid = lst[mid_index]

    if mid == s:

        return mid_index

    elif mid > s:

        return binary_search(lst[0:mid_index], s)

    else:

        result = binary_search(lst[mid_index+1:], s)

        if result == -1:

            return result
        else:

            return result + mid_index + 1
```

Evaluate when last is [1,2, 3, 4] and s is 2
binary_search([1, 2, 3, 4], 4)




```
def binary_search(lst, s):
    '''(list of float) -> int

    Return the index of s in lst, or -1 if s is not in Lst.

    lst must be a sorted list
    '''


    if lst == []: #base case

        return -1
```

mid_index = len([1, 2, 3, 4]) // 2 = 2

mid = lst[2] = 3

```
    mid_index = len(lst) // 2

    mid = lst[mid_index]

    if mid == s:

        return mid_index

    elif mid > s:

        return binary_search(lst[0:mid_index], s)

    else:
```

binary_search(lst[2+1:], 4)
binary_search(lst[3:], 4)

```
        result = binary_search(lst[mid_index+1:], s)

        if result == -1:

            return result
        else:

            return result + mid_index + 1
```

Binary Search

```
def binary_search(lst, s):
    '''(list of float) -> int

    Return the index of s in lst, or -1 if s is not in Lst.

    lst must be a sorted list
    '''

    if lst == []: #base case
        return -1
    mid_index = len(lst) // 2
    mid = lst[mid_index]
    if mid == s:
        return mid_index
    elif mid > s:
        return binary_search(lst[0:mid_index], s)
    else:
        result = binary_search(lst[mid_index+1:], s)
        if result == -1:
            return result
        else:
            return result + mid_index + 1
```

[4] 4

mid_index = len([4]) // 2 = 0

mid = lst[0] = 4

return 0

Evaluate when last is [1,2, 3, 4] and s is 2
binary_search([1, 2, 3, 4], 4)

```
def binary_search(lst, s):
    '''(list of float) -> int

    Return the index of s in lst, or -1 if s is not in Lst.

    lst must be a sorted list
    '''

    if lst == []: #base case
        return -1
    mid_index = len(lst) // 2
    mid = lst[mid_index]
    if mid == s:
        return mid_index
    elif mid > s:
        return binary_search(lst[0:mid_index], s)
    else:
        result = binary_search(lst[mid_index+1:], s)
        if result == -1:
            return result
        else:
            return result + mid_index + 1
```

[1, 2, 3, 4] 4

mid_index = len([1, 2, 3, 4]) // 2 = 2

mid = lst[2] = 3

binary_search(lst[2+1:], 4)
binary_search(lst[3:], 4) = 0

return 0 + 2 + 1 = 3

Binary Search