Recursive Algorithms

Introduction

Lecture 11

by Marina Barsky

https://www.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/recursion

Algorithms can call other algorithms

```
Algorithm h(integer x)
    return x – 5

Algorithm g(integer x)
    y = h(x - 10)
    return 2*y

Algorithm f(integer x)
    y = g(3*x)
    return y + 1

z = f(5)
print (z)
```

- What is printed?

A. 10
B. 1
C. 4
D. 0
E. None of the above

Stacking code frames

```
Algorithm h(integer x)
    return x – 5


Algorithm g(integer x)
    y = h(x - 10)
    return 2*y


Algorithm f(integer x)
    y = g(3*x)
    return y + 1
```

```
z=f(5)
```

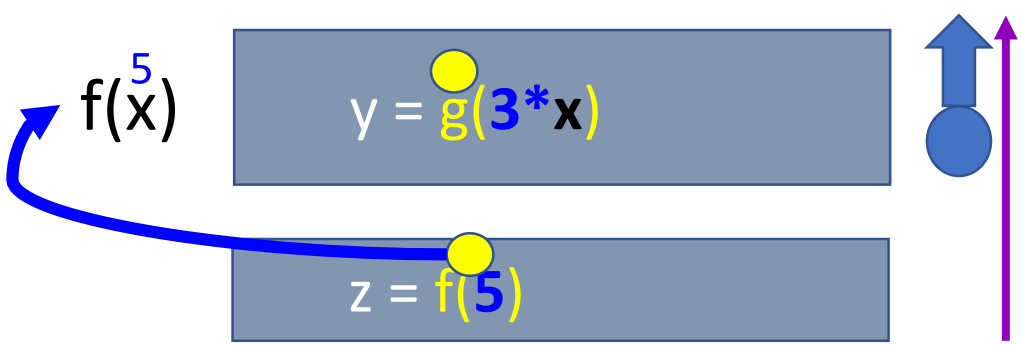
Stack frames

```
z = f(5)
```

Stacking code frames

```
Algorithm h(integer x)
    return x – 5

Algorithm g(integer x)
    y = h(x - 10)
    return 2*y

Algorithm f(integer x)
    y = g(3*x)
    return y + 1

z=f(5)
```

Stack frames



Stacking code frames

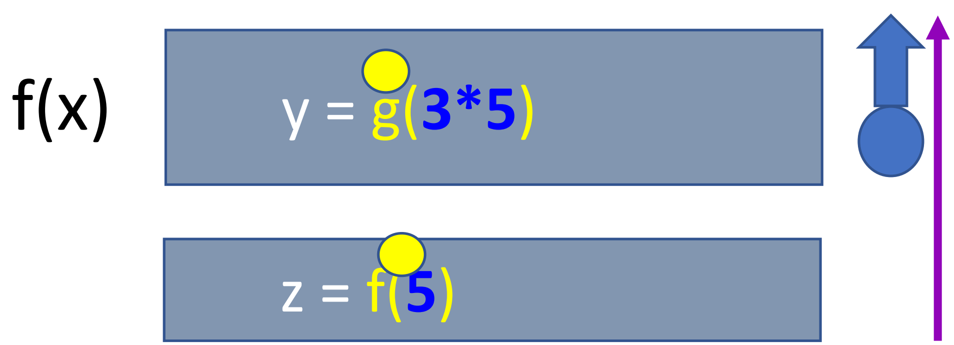
```
Algorithm h(integer x)
    return x – 5
```

```
Algorithm g(integer x)
    y = h(x - 10)
    return 2*y
```

```
Algorithm f(integer x)
    y = g(3*x)
    return y + 1
```

```
z=f(5)
```

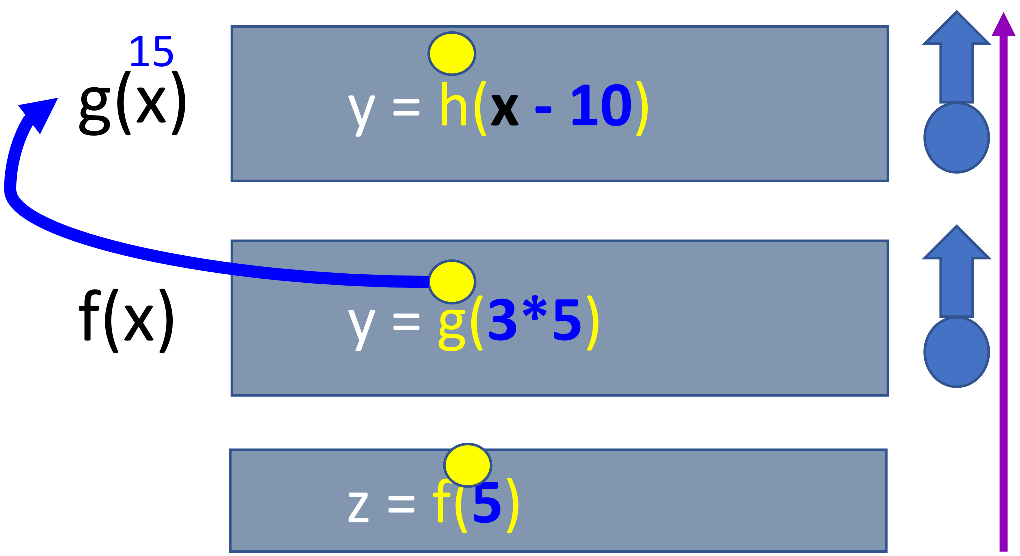
Stack frames



Stacking code frames

```
Algorithm h(integer x)
    return x – 5

Algorithm g(integer x)
    y = h(x - 10)
    return 2*y

Algorithm f(integer x)
    y = g(3*x)
    return y + 1

z=f(5)
```

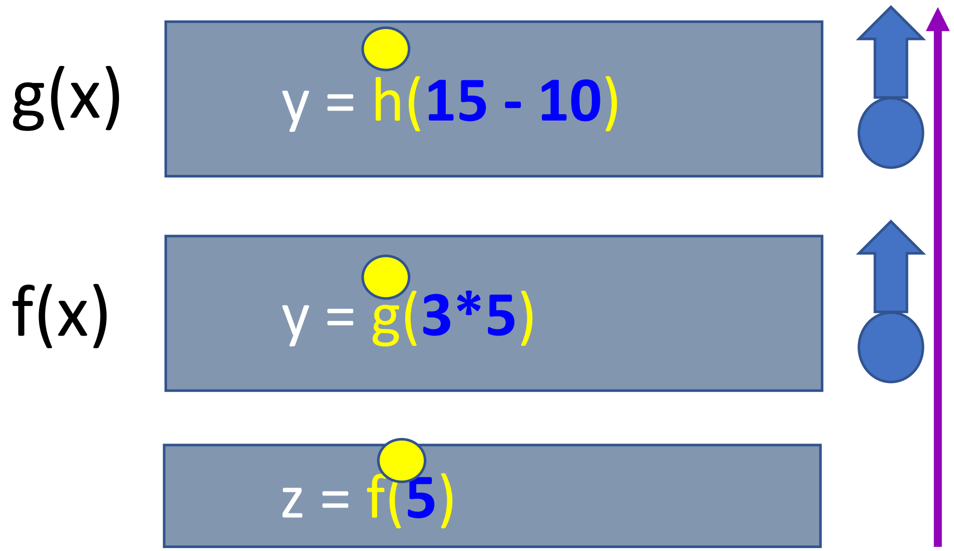
Stack frames



Stacking code frames

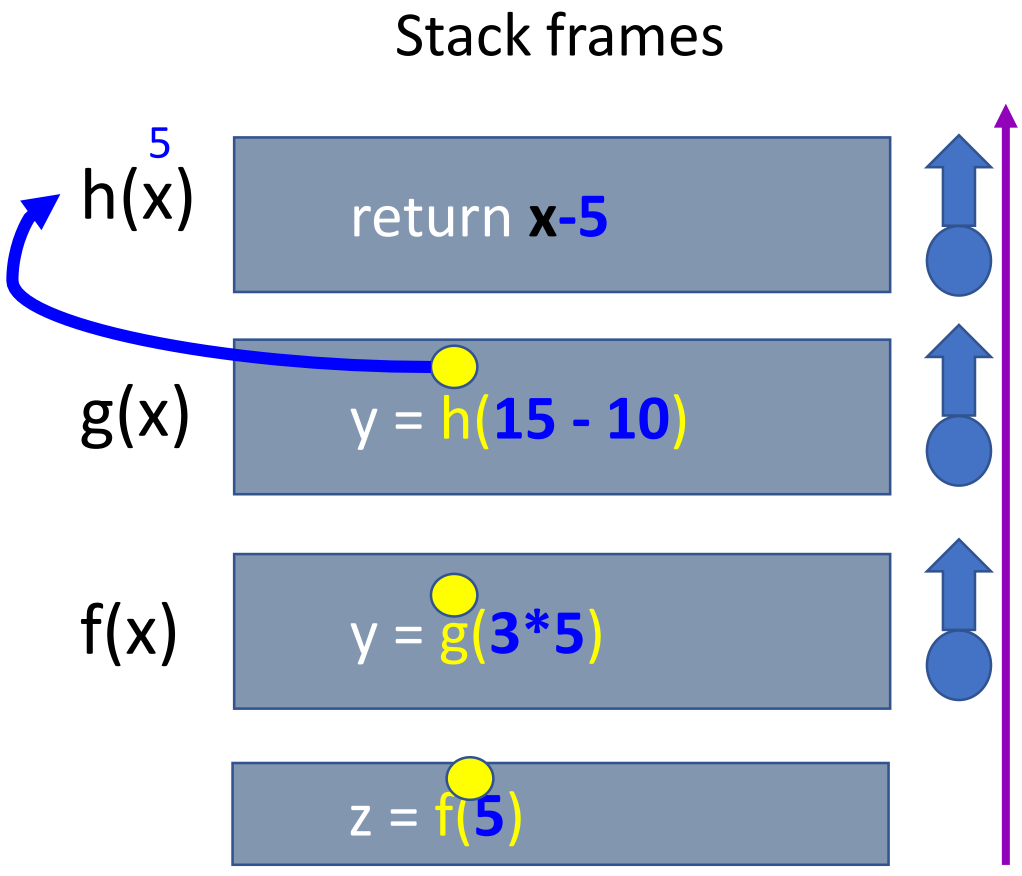
```
Algorithm h(integer x)
    return x – 5

Algorithm g(integer x)
    y = h(x - 10)
    return 2*y

Algorithm f(integer x)
    y = g(3*x)
    return y + 1

z=f(5)
```

Stack frames



Stacking code frames

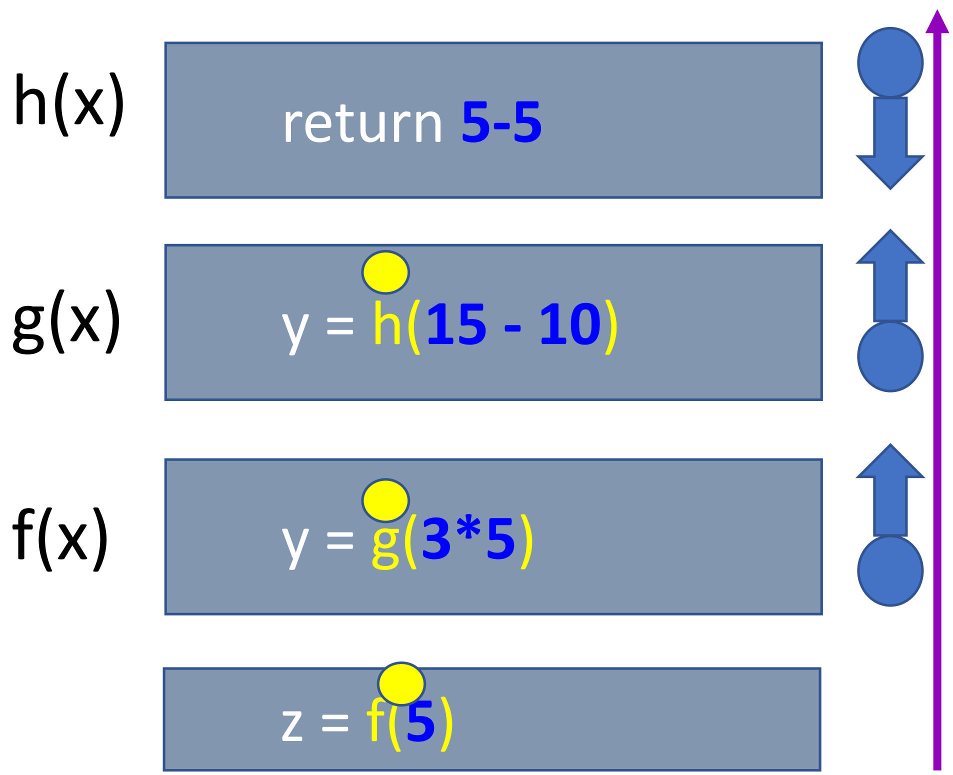
```
Algorithm h(integer x)
    return x – 5

Algorithm g(integer x)
    y = h(x - 10)
    return 2*y

Algorithm f(integer x)
    y = g(3*x)
    return y + 1

z=f(5)
```



Stacking code frames

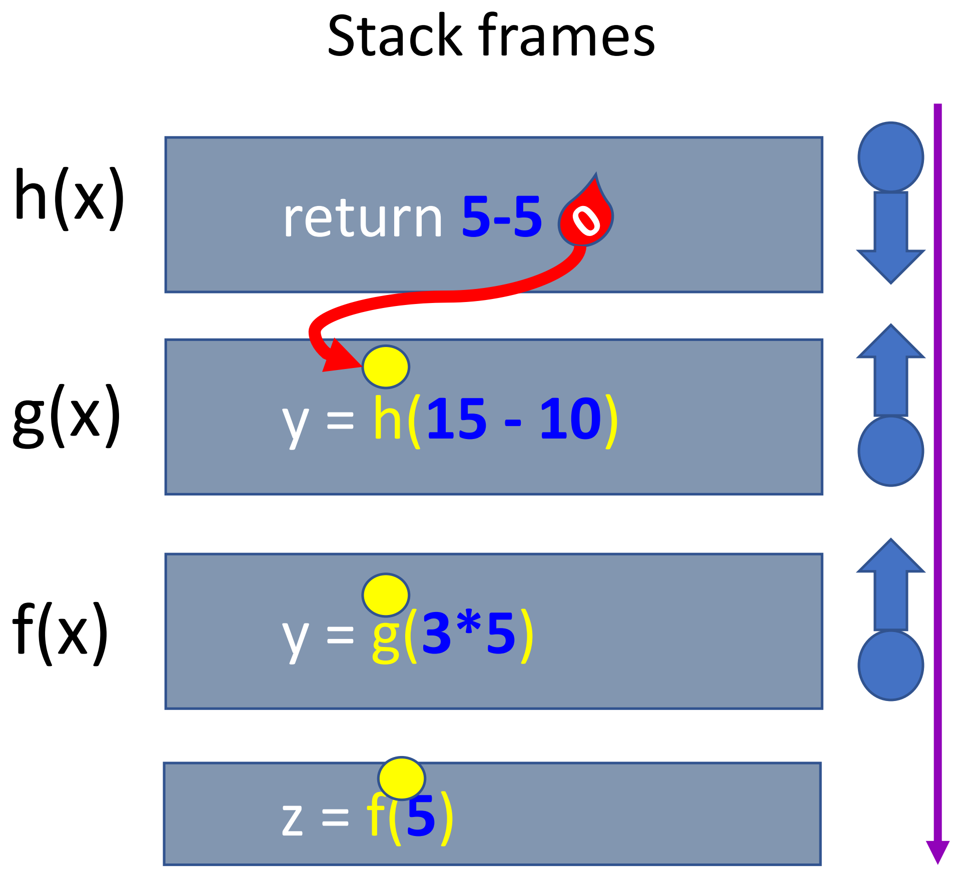
```
Algorithm h(integer x)
    return x – 5

Algorithm g(integer x)
    y = h(x - 10)
    return 2*y

Algorithm f(integer x)
    y = g(3*x)
    return y + 1

z=f(5)
```

Stack frames



Stacking code frames

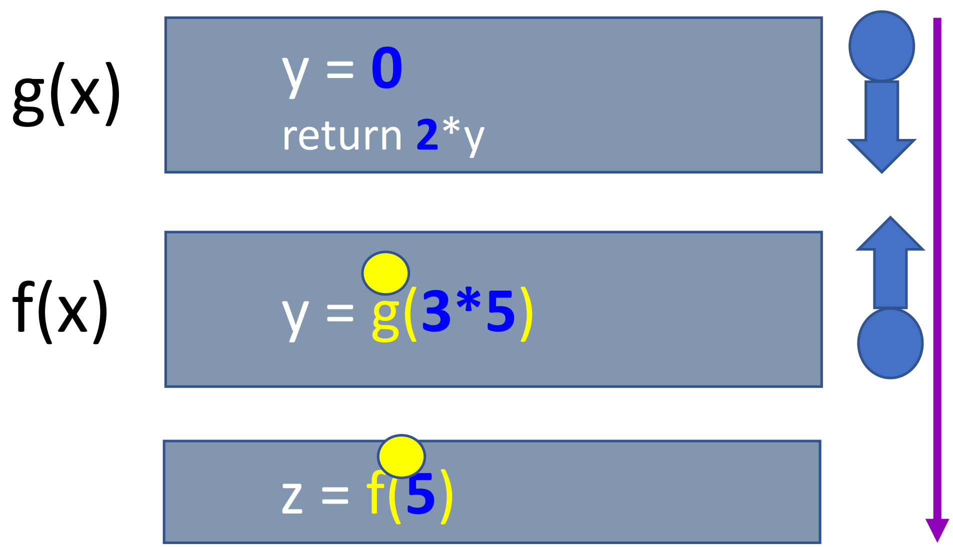
```
Algorithm h(integer x)
    return x – 5

Algorithm g(integer x)
    y = h(x - 10)
    return 2*y

Algorithm f(integer x)
    y = g(3*x)
    return y + 1

z=f(5)
```



Stacking code frames

```
Algorithm h(integer x)
    return x – 5

Algorithm g(integer x)
    y = h(x - 10)
    return 2*y

Algorithm f(integer x)
    y = g(3*x)
    return y + 1

z=f(5)
```

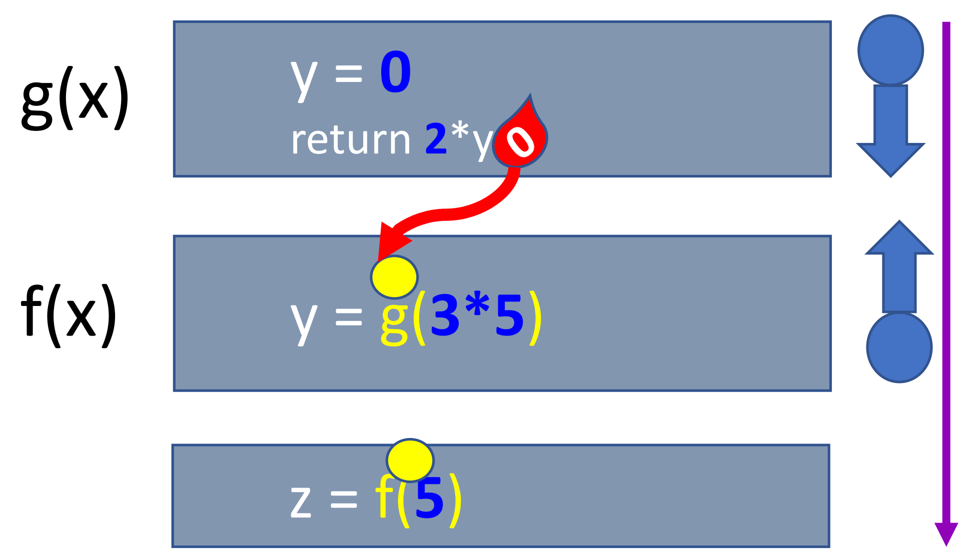
Stack frames



Stacking code frames

```
Algorithm h(integer x)
    return x – 5

Algorithm g(integer x)
    y = h(x - 10)
    return 2*y

Algorithm f(integer x)
    y = g(3*x)
    return y + 1

z=f(5)
```

Stack frames

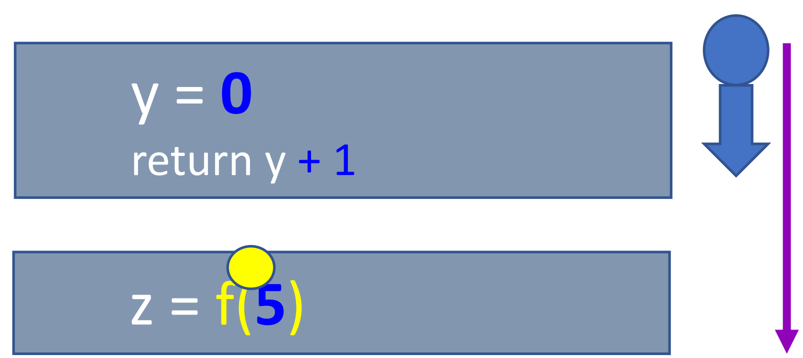


Stacking code frames

```
Algorithm h(integer x)
   return x – 5

Algorithm g(integer x)
   y = h(x - 10)
   return 2*y

Algorithm f(integer x)
   y = g(3*x)
   return y + 1

z=f(5)
```

Stack frames

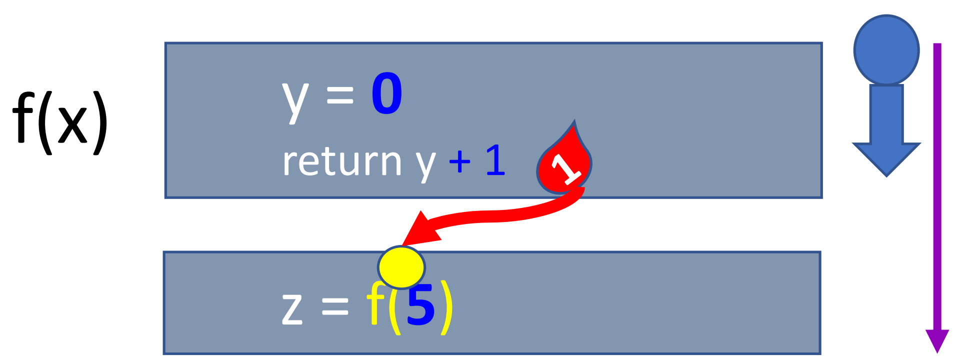
f(x)



Stacking code frames

```
Algorithm h(integer x)
    return x – 5

Algorithm g(integer x)
    y = h(x - 10)
    return 2*y

Algorithm f(integer x)
    y = g(3*x)
    return y + 1
```

Stack frames




```
z=f(5)
```

Stacking code frames

```
Algorithm h(integer x)
    return x – 5

Algorithm g(integer x)
    y = h(x - 10)
    return 2*y

Algorithm f(integer x)
    y = g(3*x)
    return y + 1

z=f(5)
```

Stack frames

z = **1**

An algorithm can call the same algorithm

- What will happen if we place call to algorithm *f*() inside algorithm *f*()?

```
Algorithm f(integer x)
      y = f(3*x)
      return y + 1
```

- The stack frames will pile up until memory permits and then the program will crash (**stack overflow!**)

Recursive algorithms

- We can use algorithms which call the same algorithm inside them if the big problem can be broken into smaller subproblems, which require the same logic to compute

- Such problems are called ***recursive problems***, and the algorithm which contains a call to itself is called a ***recursive algorithm***

Example of recursive problem: factorial

$$5! = 5*\underbrace{4*3*2*1}_{4!}$$

$5! = 5 * (4!)$

$4! = 4 * (3!)$

etc...

Recurrence relation

$F(n) = n * F(n-1)$ for $n > 1$

$F(1) = 1$

Definition of *factorial*

F(n) = n * F(n-1) for n > 0

F(1) = 1

Algorithm factorial (n)

Base case

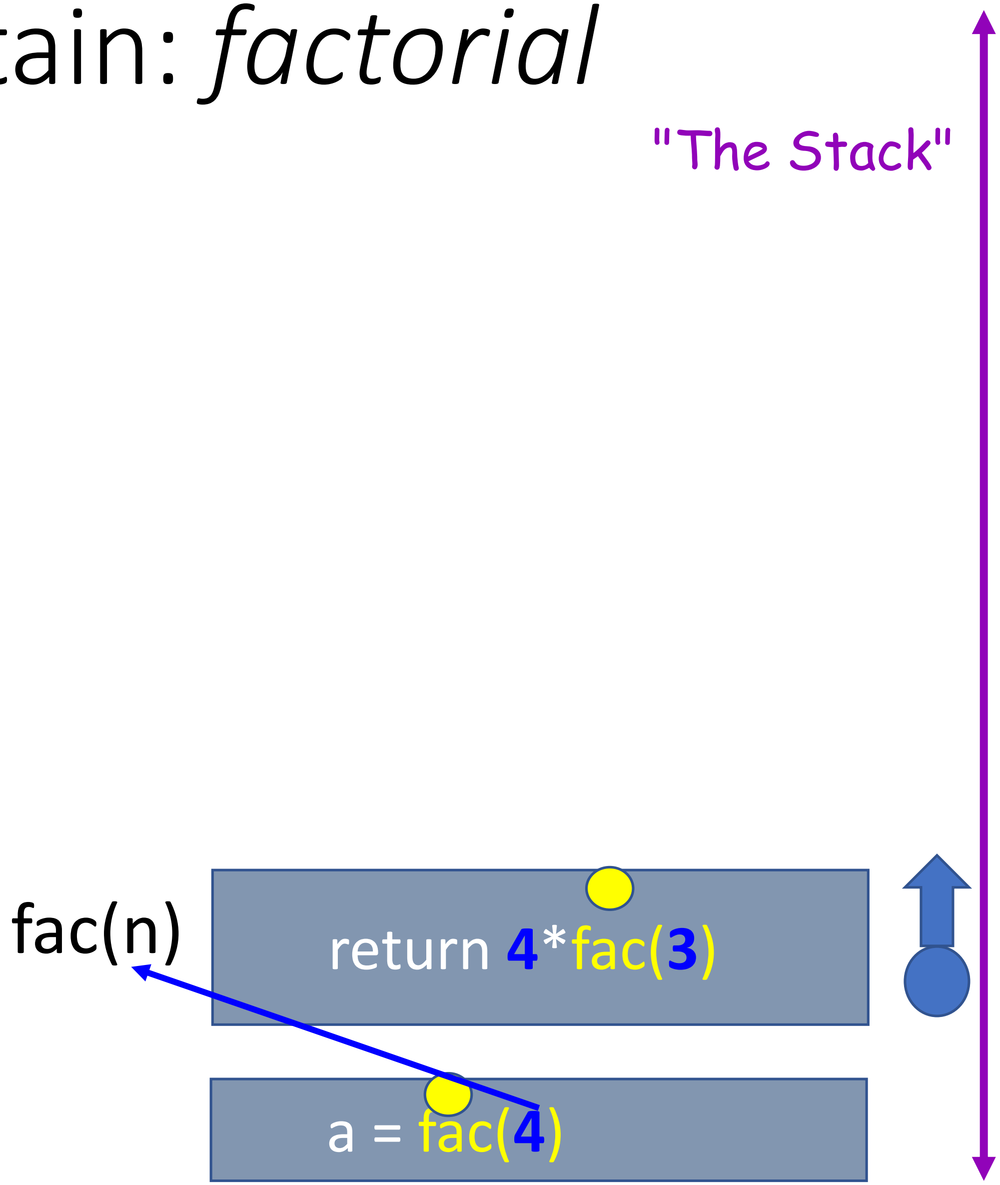
```
    if n <= 1:
        return 1
    return n * factorial (n-1)
```

Recurse with smaller problem

Behind the curtain: *factorial*

```
Algorithm fac (n):
    if n <= 1:
        return 1
    return n * fac (n-1)

a = fac (4)
```

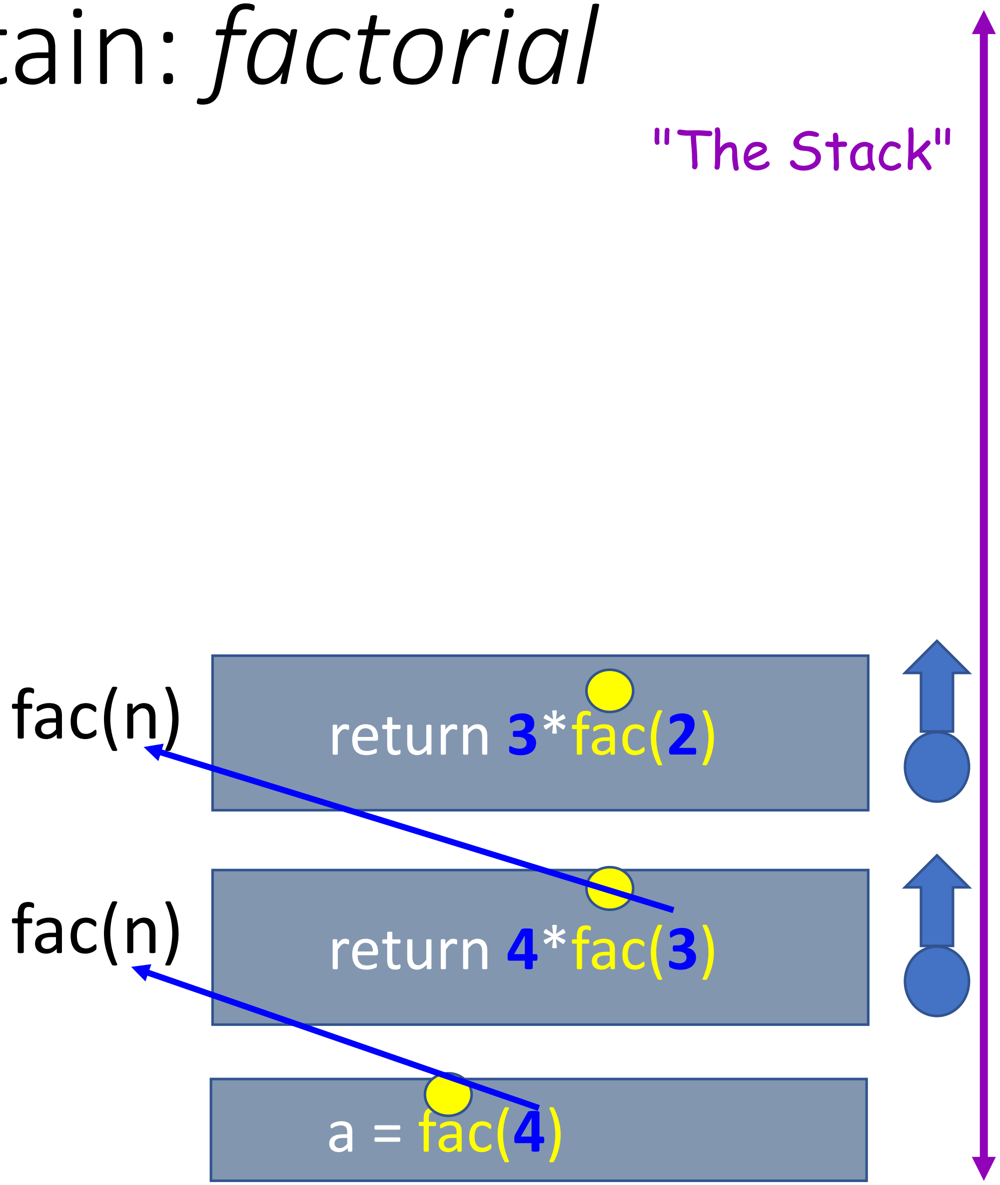
Loaded definition of *fac* to compute *fac(4)*, but cannot compute, needs to compute *fac(3)* first



Behind the curtain: *factorial*

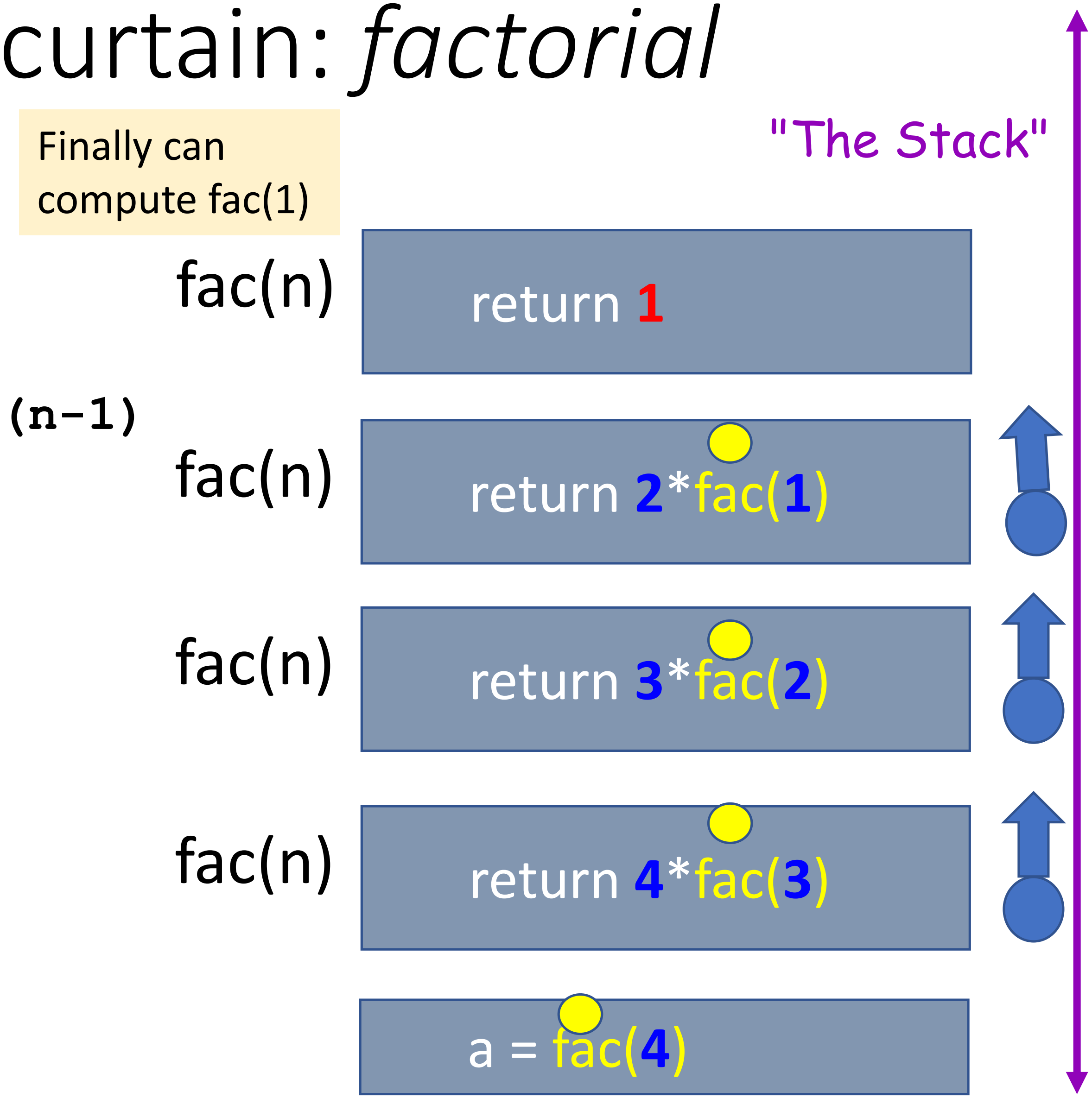
```
Algorithm fac (n):
    if n <= 1:
        return 1
    return n * fac (n-1)

a = fac (4)
```

Loaded a **different copy** of *fac*, to compute *fac(3)*



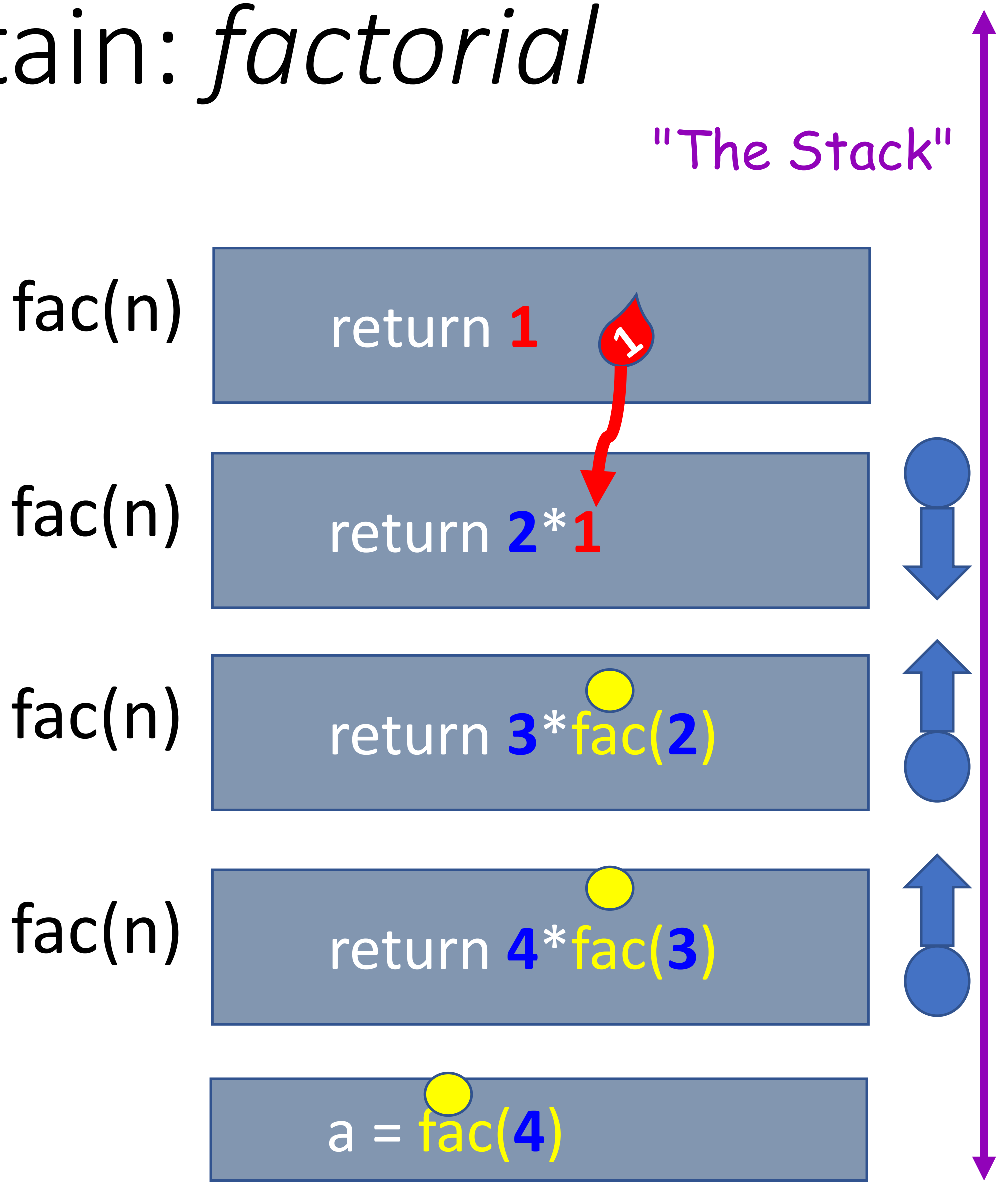
Behind the curtain: *factorial*

```
Algorithm fac (n):
    if n <= 1:
        return 1
    return n * fac (n-1)

a = fac (4)
```



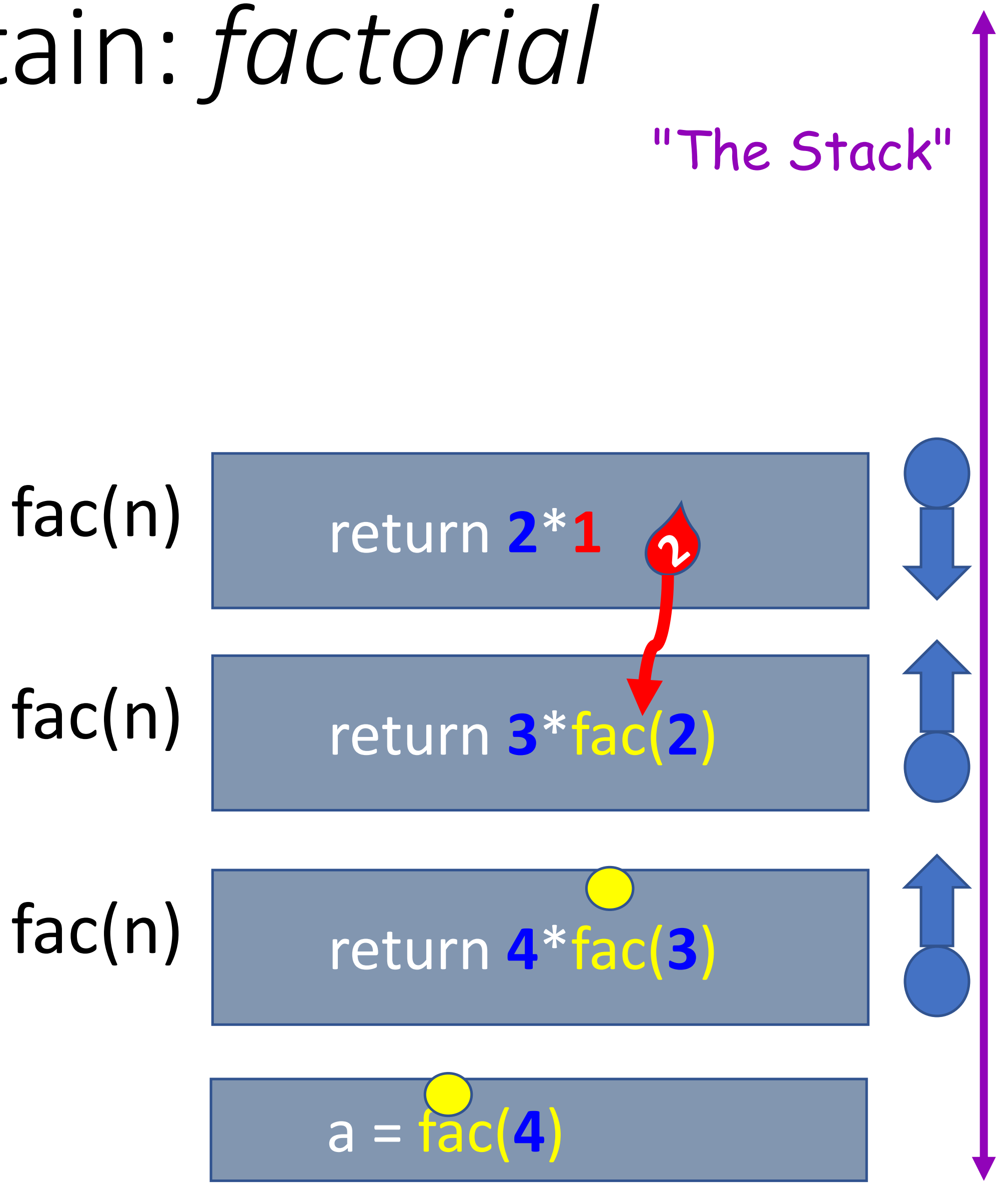
Behind the curtain: *factorial*

```
Algorithm fac (n):
    if n <= 1:
        return 1
    return n * fac (n-1)

a = fac (4)
```



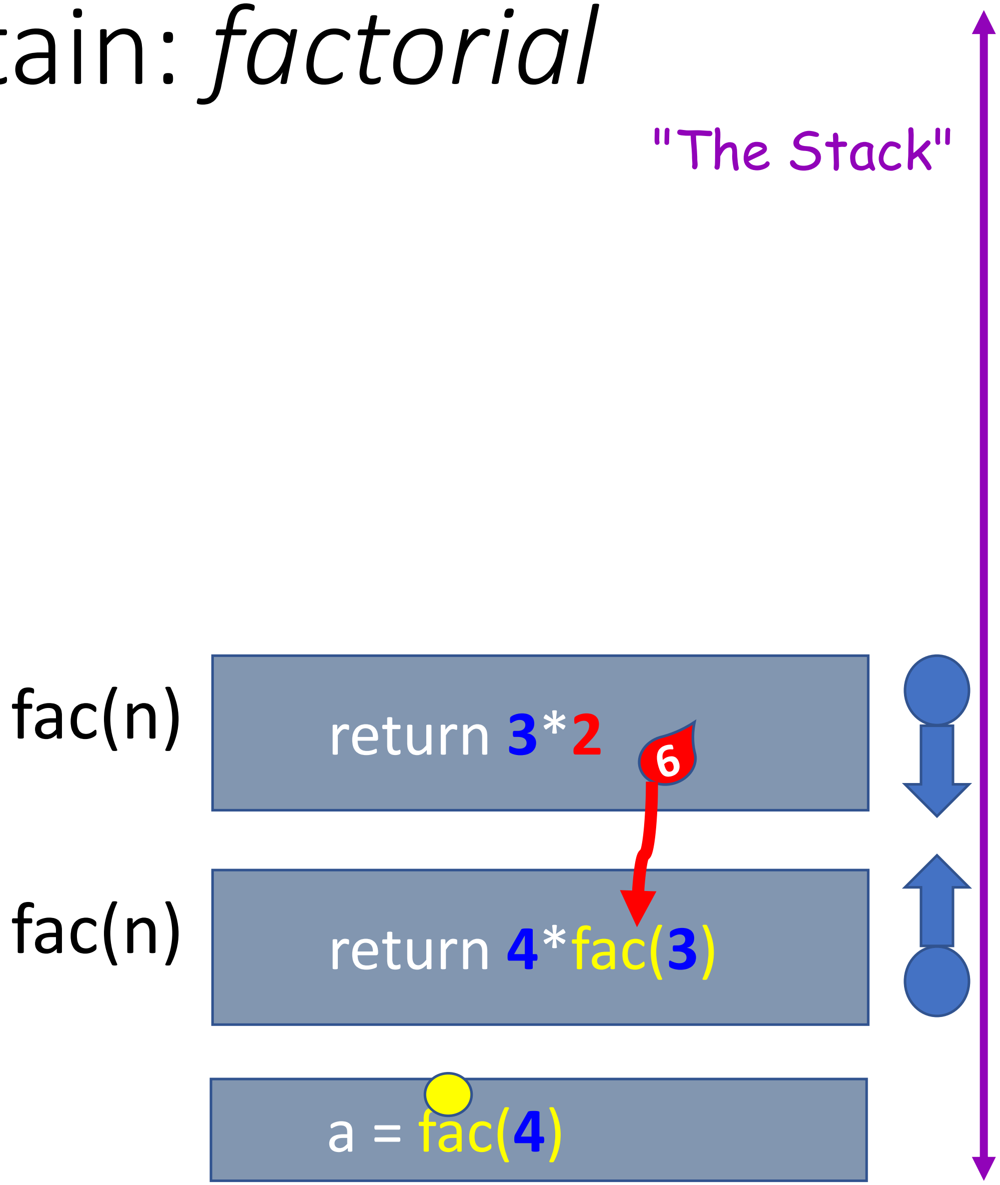
Behind the curtain: *factorial*

```
Algorithm fac (n):
    if n <= 1:
        return 1
    return n * fac (n-1)

a = fac (4)
```



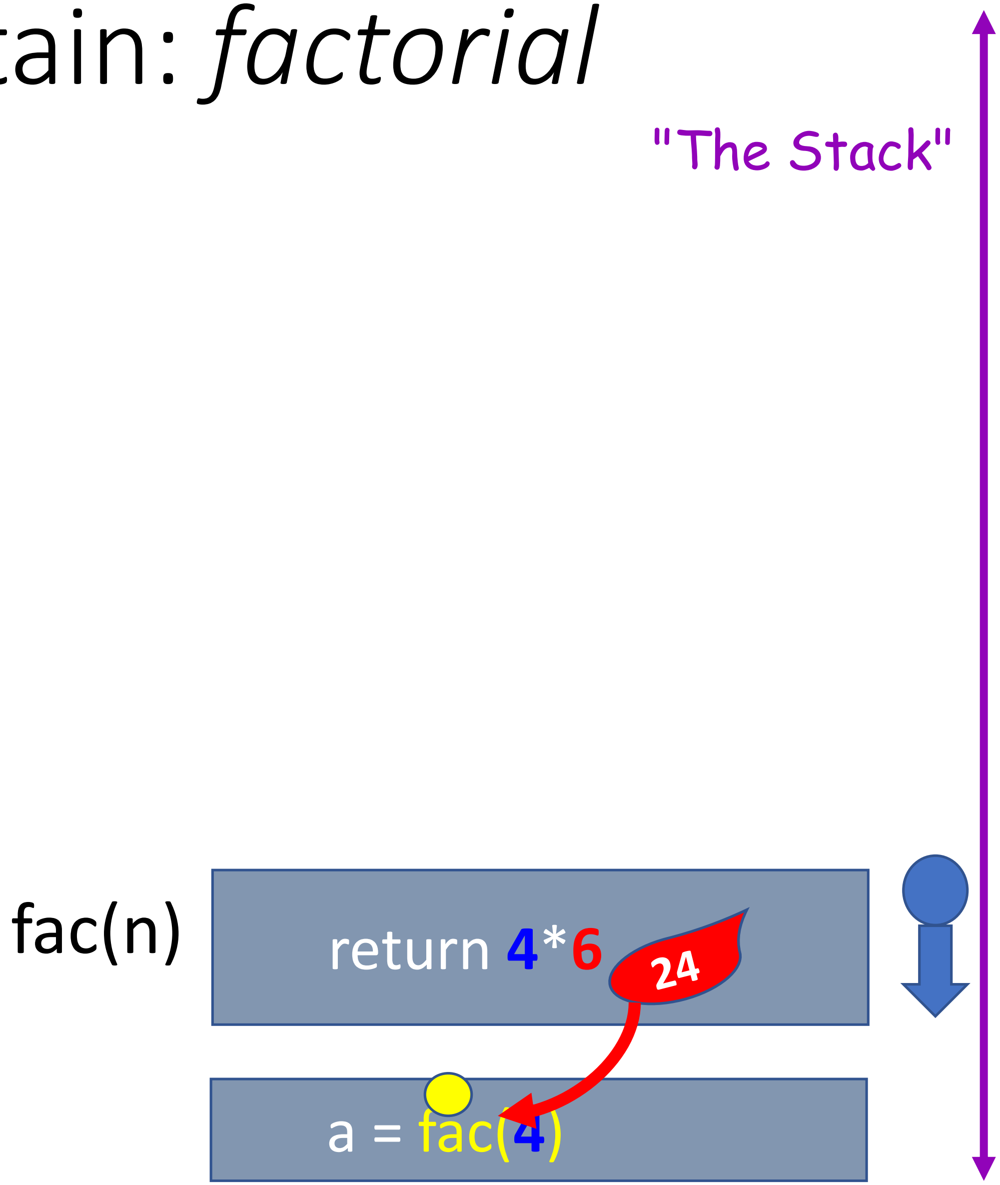
Behind the curtain: *factorial*

```
Algorithm fac (n):
    if n <= 1:
        return 1
    return n * fac (n-1)

a = fac (4)
```



Behind the curtain: *factorial*

```
Algorithm fac (n):
    if n <= 1:
        return 1
    return n * fac (n-1)

a = fac (4)
```



Behind the curtain: *factorial*

```
Algorithm fac (n):
    if n <= 1:
        return 1
    return n * fac (n-1)

a = fac (4)
```

Components of a recursive solution

Base case

- A recursive solution must have **one or more (non-recursive) base cases** (when to stop)

  factorial(1) = 1

Recursive Step

- Recursive calls must **progress towards the base case**: a recursive solution to a problem of certain size should be expressed through the exact same solution **with a smaller problem size**

  factorial(n) = n * factorial(**n-1**)

Recursive step

You are implementing algorithm *power(x,y)* for computing x^y.

What is your recursive step?

```
A. power(x) = power(x,y-1)

B. power(x) = x*power(x,y)

C. power(x) = x*power(x,y-1)

D. power(x) = y*power(x-1,y)

E. None of the above
```

Base case

You are implementing algorithm *power(x,y)* for computing x^y.

What is your base case?

```
A.  If x == 1, return x

B.  If y == 1, return x

C.  If y == 0, return 1

D.  If y == 0, return 0

E.  None of the above
```

Order 1. What is printed?

```
algorithm printNum(count):
   if count < 1:
       return
   print(count)
   printNum(count-1)

printNum(4)
```

A. 4 3 2 1 0

B. 0 1 2 3 4

C. 4 3 2 1

D. 1 2 3 4

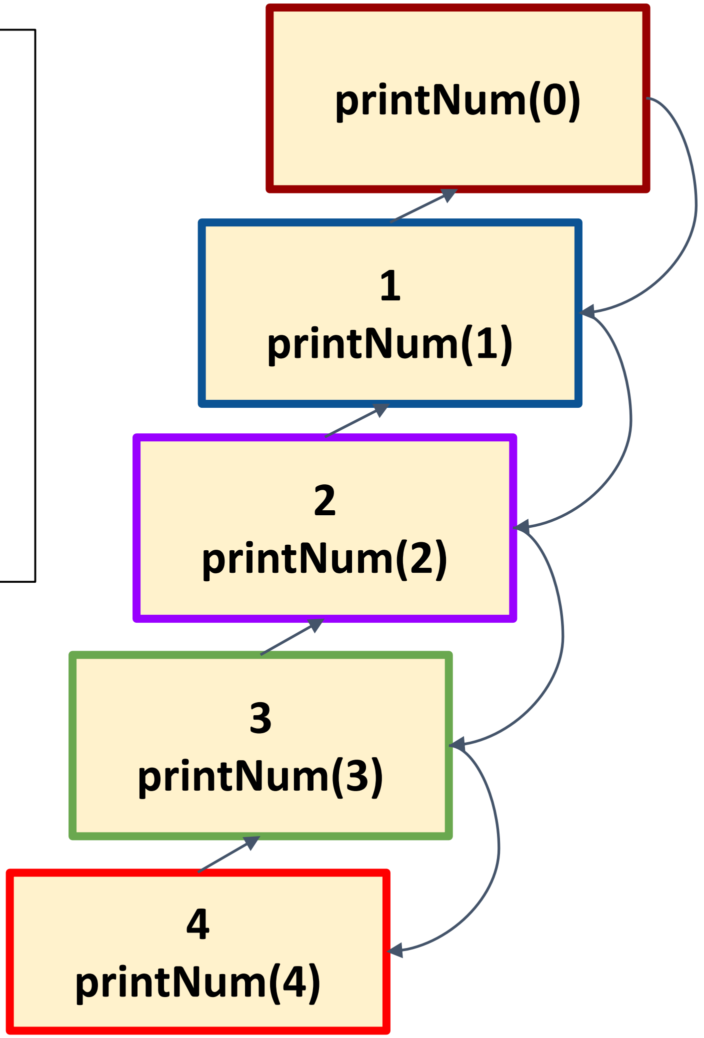
E. Error: stack overflow

What is printed? Solution

```
algorithm printNum(count):
    if count < 1:
        return
    print(count)
    printNum(count-1)

printNum(4)
```

Correct answer: C

4 3 2 1



Order 2. What is printed now?

```
algorithm printNum(count):
   if count < 0:
       return
   printNum(count-1)
   print(count)


printNum(4)
```

A. 4 3 2 1

B. 1 2 3 4

C. 4 3 2 1 0

D. 0 1 2 3 4

Fun. What is fun(5)?

```
Algorithm fun(n):
    if n <= 1:
        return 1
    else if n%2 == 0:
        return fun(n/2)
    else:
        return fun(n/2) + fun(n/2 + 1)
```

A. 3

B. 5

C. 2

D. 1

E. None of the above

Fun. What is fun(5)? Solution

```
Algorithm fun(n):
    if n <= 1:
        return 1
    else if n%2 == 0:
        return fun(n/2)
    else:
        return fun(n/2) + fun(n/2 + 1)
```

A. 3
B. 5
C. 2
D. 1
E. None of the above

fun(5) = fun(2) + fun(3)
    fun(2) = fun(1)
    fun(1) = 1 → fun(2) = 1
    fun(3) = fun(1) + fun(2) = 1 + 1 = 2
fun(5) = 1 + 2 = 3

Reasoning about time complexity

Recursive step:

```
factorial (n) = n * factorial (n-1)
```

Recurrence relation for running time:

Express running time for size n through running time for smaller input: $T(n) = 1 + T(n-1)$

Reasoning about time complexity

$T(n) = 1 + T(n-1)$

$\Downarrow$

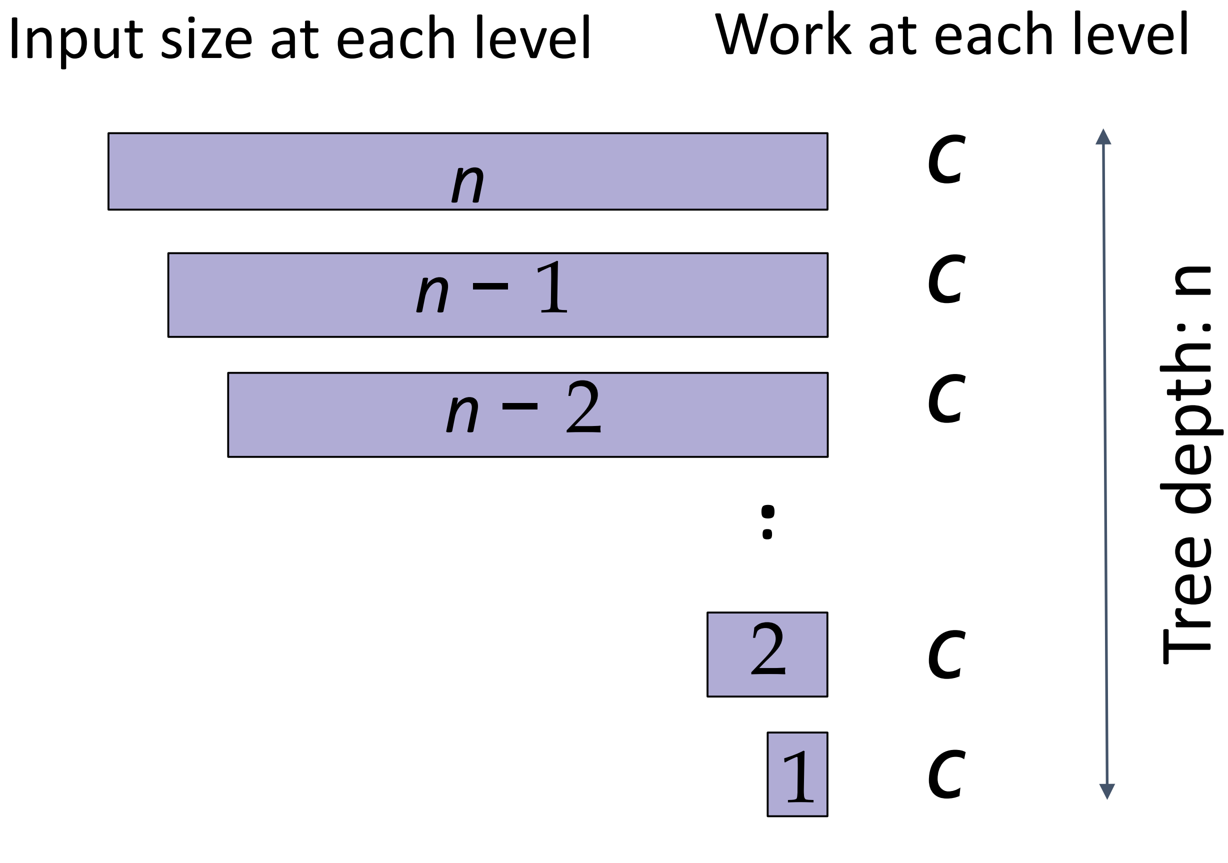
big O

Steps:

- ❑ Draw recursion tree
- ❑ Estimate the depth of the tree
- ❑ Estimate work done at each level of the tree
- ❑ Add all level work together

T(n) = 1 + T(n-1)

Factorial: recursion tree



Total: $cn = O(n)$

Recursive Searching in Sorted Data

Separate and conquer

https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search

Problem: Searching in a sorted array

Input: A sorted array $A[low \ . \ . \ . \ high]$ $(\forall low \leq i < high : A[i] \leq A[i+1])$.

A value *key* to search for.

Output: An index, i , $(low \leq i \leq high)$ where $A[i] = key$.

Otherwise, return -1 (NOT_FOUND).

Searching in a Sorted Array

Example

search(2) $\longrightarrow$ -1
search(3) $\longrightarrow$ 0
search(4) $\longrightarrow$ -1

search(20) $\longrightarrow$ 3
search(20) $\longrightarrow$ 4
search(60) $\longrightarrow$ 6

search(90) $\longrightarrow$ -1

3	5	8	20	20	50	60
0	1	2	3	4	5	6

BinarySearch(A, low, $high$, key)

if $high < low$:
    return -1

$mid = low + \left\lfloor \frac{high - low}{2} \right\rfloor$

if $key == A[mid\,]$:
    return mid
else if $key < A[mid\,]$:
    return BinarySearch(A, low, $mid - 1$, key)
else:
    return BinarySearch(A, $mid + 1$, $high$, key)

Example: Searching for key 50

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

Example: Searching for key 50

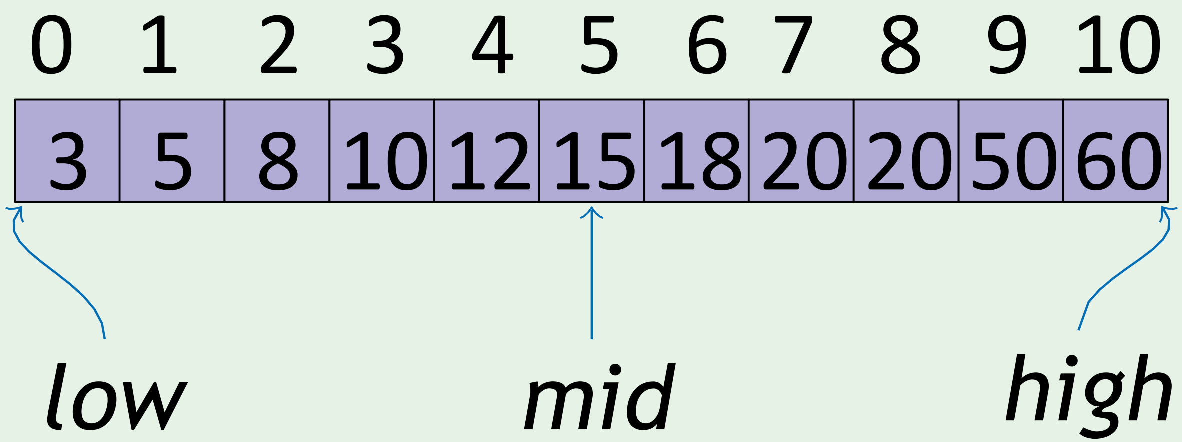
BinarySearch(A, 0, 10, 50)

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *high*

Example: Searching for key 50

BinarySearch(A, 0, 10, 50)



Example: Searching for key 50

BinarySearch(A, 0, 10, 50)

BinarySearch(A, 6, 10, 50)

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

BinarySearch(A, 0, 10, 50)

BinarySearch(A, 6, 10, 50)

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

BinarySearch(A, 0, 10, 50)

BinarySearch(A, 6, 10, 50)

BinarySearch(A, 9, 10, 50)

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

BinarySearch(A, 0, 10, 50)

BinarySearch(A, 6, 10, 50)

BinarySearch(A, 9, 10, 50)

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

$\text{BinarySearch}(A,\ 0,\ 10,\ 50)$

$\text{BinarySearch}(A,\ 6,\ 10,\ 50)$

$\text{BinarySearch}(A,\ 9,\ 10,\ 50) \rightarrow 9$

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

Puzzle challenge: find the fake coin

- There are 8 identical-looking coins
- One of these coins is counterfeit and is known to be lighter than the genuine coins

- What is the minimum number of weighings needed to identify the fake coin with a two-pan balance scale without weights?

Iterative Version

BinarySearchIt(A, low, $high$, key)

while $low \le high$:

$$mid = low + \left\lfloor \frac{high - low}{2} \right\rfloor$$

if $key == A[mid]$:

return mid

else if $key < A[mid]$:

$high = mid - 1$

else:

$low = mid + 1$

return -1

Iterative Version

BinarySearchIt(A, low, $high$, key)

while $low \le high$:

$$mid = low + \left\lfloor \frac{high - low}{2} \right\rfloor$$

if $key == A[mid]$:

return mid

else if $key < A[mid]$:

$high = mid - 1$

else:

$low = mid + 1$

return -1

Running time:

- The size of the input halves at each iteration
- Such loops terminate in *log n* steps
- At each loop iteration we do a constant number of operations
- So the running time is O(*log n)*

BinarySearch(A, low, $high$, key)

if $high < low$:
    return -1
$mid = low + \left\lfloor \frac{high - low}{2} \right\rfloor$
if $key == A[mid]$:
    return mid
else if $key < A[mid]$:
    return BinarySearch(A, low, $mid - 1$, key)
else:
    return BinarySearch(A, $mid + 1$, $high$, key)

Recurrence relation for running time:
$T(n) = 1 + T(n/2)$

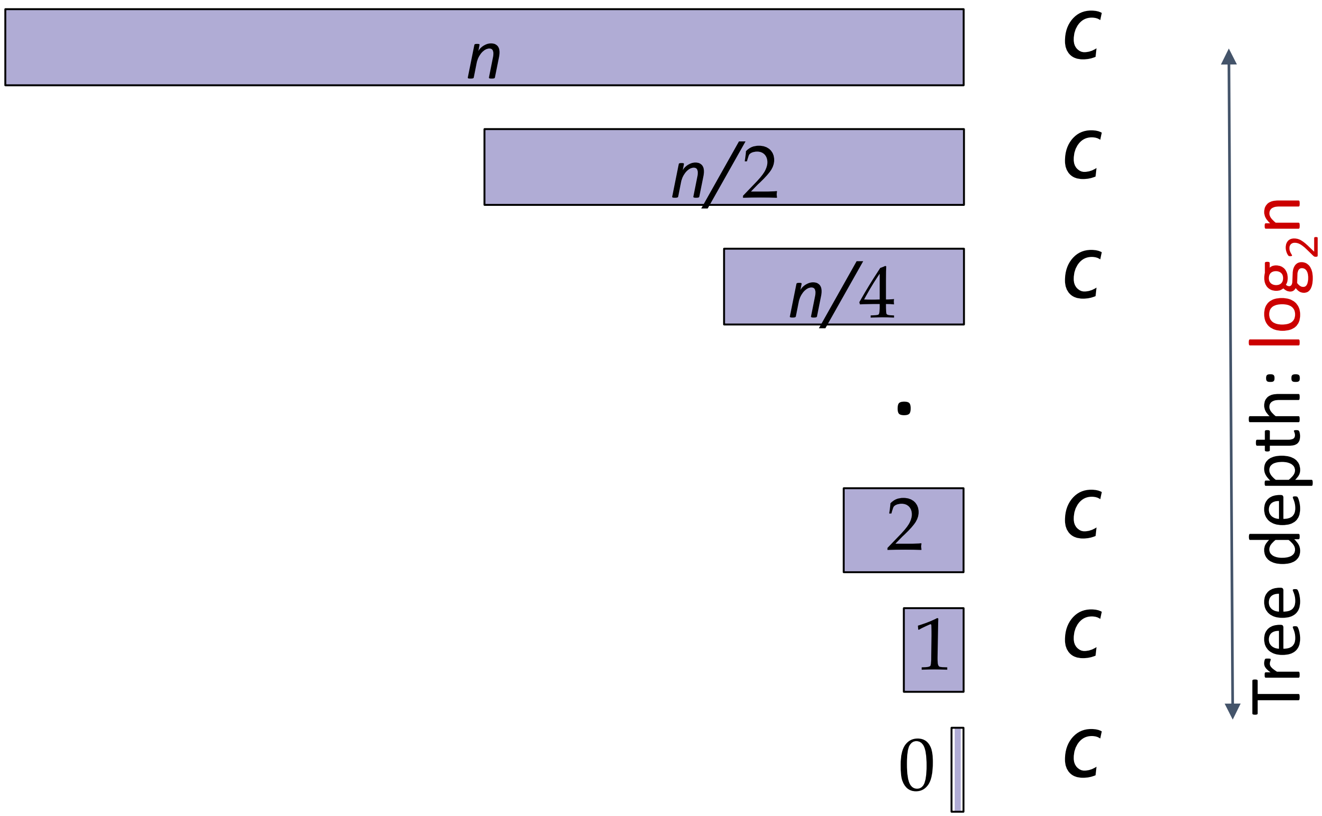
Recursion Tree

T(n) = 1 + T(n/2)

Binary Search:

Input size at each level

Work at each level



Total: $\sum_{i=0}^{\log_2 n} c = O(\log n)$

Linear search

$O(n)$

Binary search

$O(\log n)$

Calculating runtime of recursive algorithms
is not always that easy

Recursive
Data Structures

When recursion feels natural

1. The *problem* is defined recursively:

   factorial (n) = n * factorial(n-1)

   binarySearch(A, 0, n) = … and binarySeach(A, 0, n/2)

2. The ***structure*** is defined recursively:

```
class Node {
    int data;
    Node next;
}
```

Linked List

```
class Node {
    int data;
    Node leftChild;
    Node rightChild;
}
```

Binary tree

Example: recursive search in Linked Lists 1/3

```
Algorithm recurFind (Node current, int target, int position)
    if (current == null)
        return -1;
    if (current.data == target)
        return position;
    return recurFind (current.next, target, position+1);


pos = recurFind(head, target, 0);
```

Base case: current node is null, we reached the end of the Linked List, return -1

Example: recursive search in Linked Lists 2/3

```
Algorithm recurFind (Node current, int target, int position)
    if (current == null)
        return -1;
    if (current.data == target)
        return position;
    return recurFind (current.next, target, position+1);

pos = recurFind(head, target, 0);
```

Check current data

Example: recursive search in Linked Lists 3/3

```
Algorithm recurFind (Node current, int target, int position)
    if (current == null)
        return -1;
    if (current.data == target)
        return position;
    return recurFind (current.next, target, position+1);

pos = recurFind(head, target, 0);
```

Recur with the next node and also increment position counter by 1

Which correctly recursively finds the size of a Linked List?

```
public int size(Node n){
    if (n.next == null){
        return 1;
    } else {
        return 1 + size(n.next);
    }
}
```

A

```
public int size(Node n){
    int s = 1;
    if (n.next == null){
        return s;
    } else {
        s++;
        retun size(n.next);
    }
}
```

B

```
public int size(Node n){
    if (n == null){
        return 0;
    } else {
        return 1 + size(n.next);
    }
}
```

C

D. More than one of the above

E. None of the above

Recursion: summary

- Recursive algorithms are particularly appropriate when the underlying ***problem*** or the ***data*** to be treated are defined in recursive terms

- To design a recursive algorithm:
  1. Think of the simplest possible input: that becomes the **base case**
  2. Imagine that we know a solution to the problem of size n-1. Think of the steps needed to convert this solution to the solution to a larger problem. This is your **recursive step**

Recursion vs. iteration

➔**Recursion**

- Each recursive call **requires extra space** on the stack
- If we get **infinite recursion**, the program will eventually run out of memory, cause stack overflow, and **the program will terminate**
- Solutions to some problems are **easier to formulate** recursively

➔**Iteration**

- Each iteration **does not require extra space**
- An **infinite loop could loop forever** since there is no extra memory being created
- Iterative **solutions** to a problem may **not** always be as **obvious** as a recursive solution

Generally, recursive solutions are slower than iterative solutions due to the overhead of function calls