

CPS 305

Data Structures

Prof. Alex Ufkes

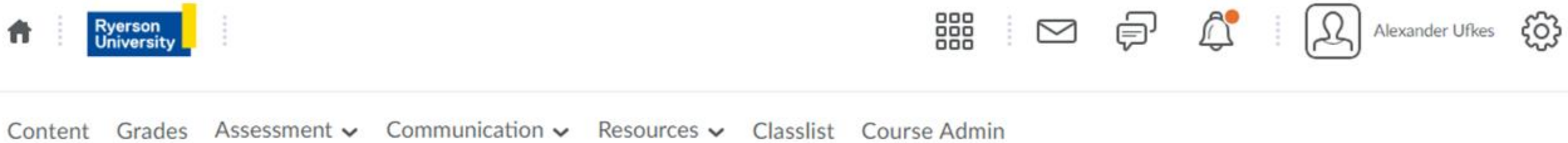
Topic 7: Tail Recursion, Hashing intro

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

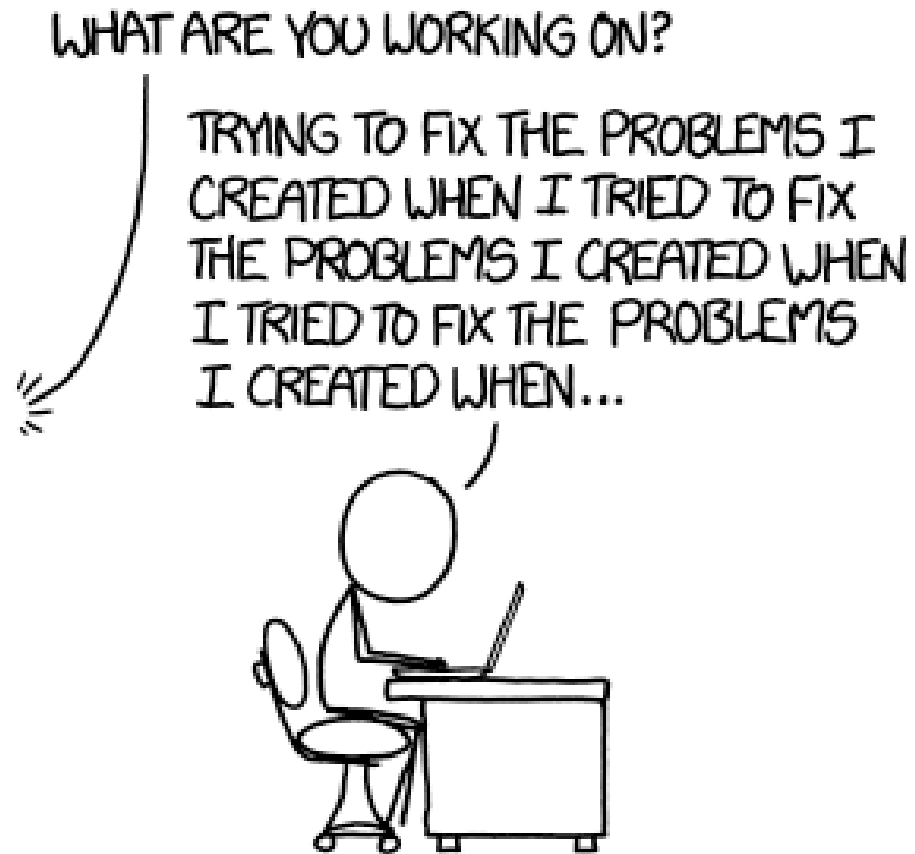
The copyright to this original work is held by Alex Ufkes. Students registered in course CPS 305 can use this material for the purposes of this course but no other use is permitted, and there can be no sale or transfer or use of the work for any other purpose without explicit permission of Alex Ufkes.

Course Administration



- Practical test on Wednesday, July 16, 9-11am
- Rooms TBA, seating TBA
- Make sure you are comfortable with the lab computer environment! Test is in Linux.

Recursion Review

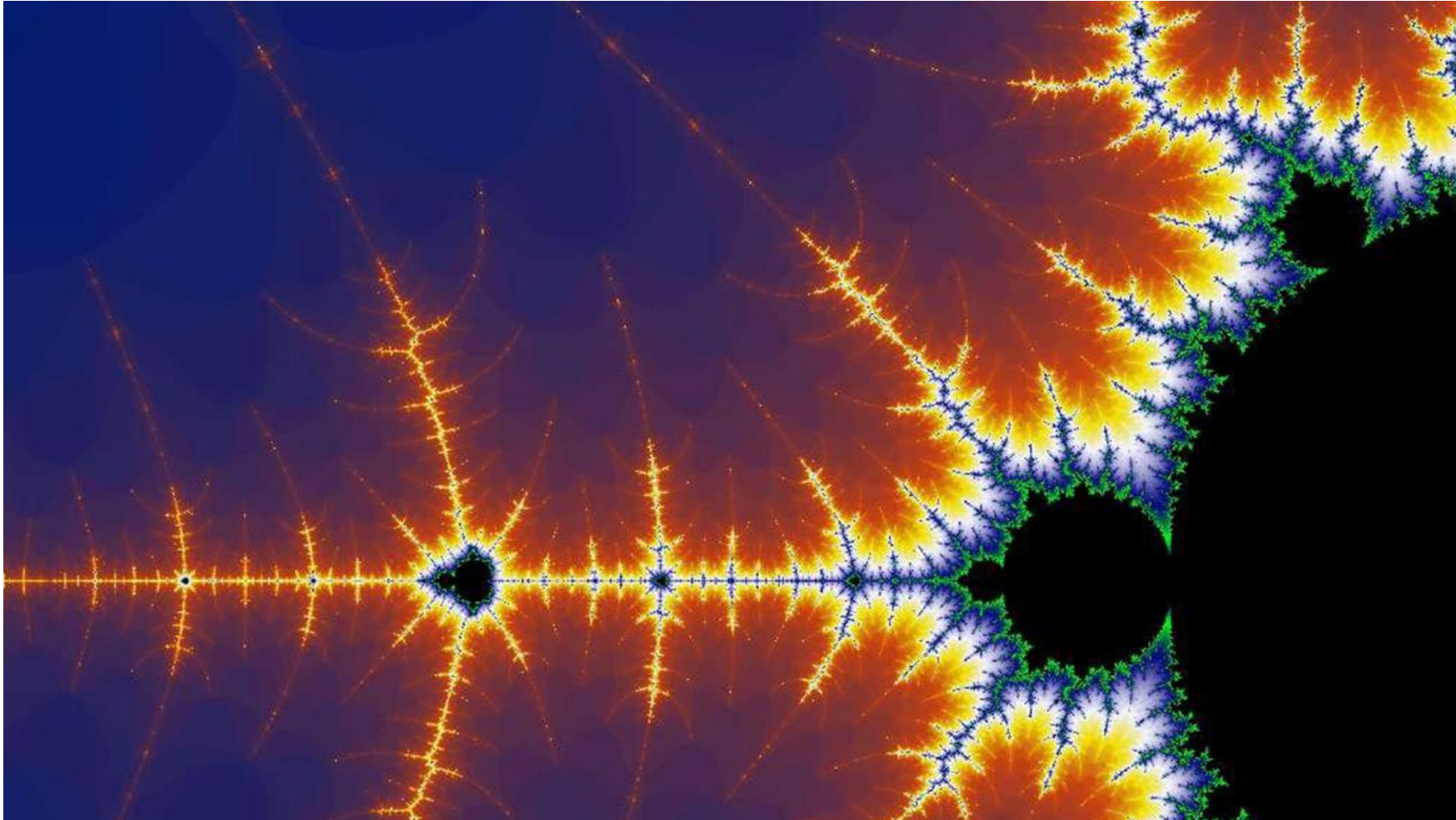


Functions Calling Functions

Breaking a large problem into smaller sub-problems

- We've written many functions to solve many problems.
- Many of these have themselves called other functions!
- It is of course perfectly legal for one function to call another.
- Two smaller problems are often easier to solve than one larger problem.

Self-Similar Problems



Self-Similar Problems

- A thing is said to be self-similar if it contains a strictly smaller version of itself inside it as a proper part.
- Each Fibonacci number is the sum of the previous two
- Every factorial is created from a smaller factorial ($5! = 5 \cdot 4!$).
- Recursion, a function calling itself with ever-smaller argument values, is often a natural way to solve self-similar problems.
- **Assuming** that we can spot the underlying self-similarity.
- This is often the most difficult part!

Recursion

Defining a function (or method) in terms of itself

In other words, a function that calls itself

Factorial:

The product of all positive integers less than or equal to a given non-negative number.

$$5! = 5 * 4 * 3 * 2 * 1$$

Factorial: Iterative definition

$$n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1$$

Factorial: Recursive definition

$$n! = n * (n-1)!$$

We've defined $n!$ in terms of another (*smaller*) factorial

Recursion Properties

Recursive solutions **must** satisfy these three rules:

1. Contain a **base case**

2. Contain a **recursive case**

3. Make **progress** towards the base case

- A subproblems so small it can be solved directly
- Simply check for it, returns its solution

Factorial: Base Case

$$5! = 5 * 4 * 3 * 2 * 1 = 4!$$

$$5! = 5 * 4!$$

$$5! = 5 * 4 * 3!$$

$$5! = 5 * 4 * 3 * 2!$$

$$5! = 5 * 4 * 3 * 2 * 1!$$

$$5! = 5 * 4 * 3 * 2 * 1 * 0!$$

- Factorial is not defined for negative integers.
- Our definition ends here!
- $0!$ is called our **base case**.
- **$0! = 1$**
- The value of the base case must **NOT** contain a factorial!

Recursion Properties

Recursive solutions **must** satisfy these three rules:

$$0! = 1$$

1. Contain a **base case**

2. Contain a **recursive case**

- A subproblems too large to be solved directly.
- We must make it smaller

3. Make **progress** towards the base case

Factorial: Recursive Case

$$5! = 5 * 4 * 3 * 2 * 1$$

$$5! = 5 * 4!$$

Can we generalize for any factorial?

$$n! = n * (n-1)!$$

- We call this the ***recursive case***.
- Factorial is defined in terms of another factorial.

Recursion Properties

Recursive solutions **must** satisfy these three rules:

1. Contain a **base case**

$$0! = 1$$

2. Contain a **recursive case**

$$n! = n * (n-1)!$$

3. Make **progress** towards the base case

Factorial: Progress Towards Base Case?

$$n! = n * (n-1)!$$

$$n! = n * \overbrace{(n-1) * (n-2)!}$$

$$n! = n * (n-1) * \overbrace{(n-2) * \underline{(n-3)!}}$$

- If we continue unwrapping this, will we eventually hit the base case?
- In other words, will the above term eventually be **0!** for any **n**?

Recursion Properties

Recursive solutions **must** satisfy these three rules:

1. Contain a **base case**

$$0! = 1$$

2. Contain a **recursive case**

$$n! = n * (n-1)!$$

3. Make **progress** towards the base case

Yes.

Iteration VS Recursion

Every **iterative** solution has a semantically equivalent **recursive** solution.

Every **recursive** solution has a semantically equivalent **iterative** solution.

Some problems are easier to solve iteratively, others are
(much, *much*) easier to solve recursively.

Factorial can be solved easily either way, so let's try!

Review: In Python

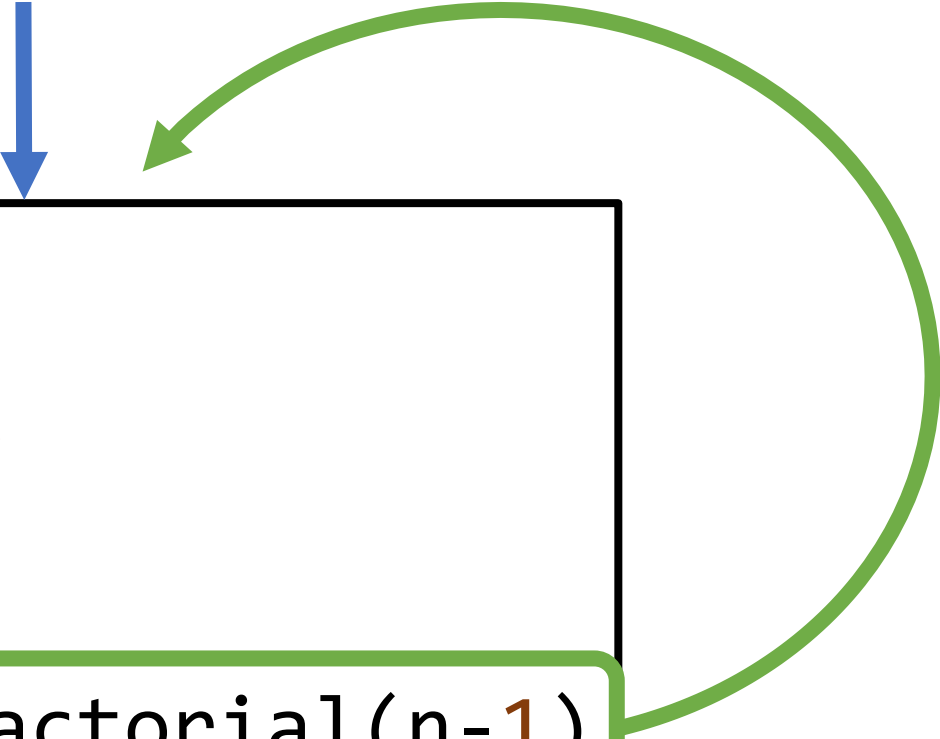
Iterative Solution:

```
def factorial(n):  
    a = 1  
    for x in range(n, 1, -1):  
        a = a * x  
    return a
```

Recursive Solution:

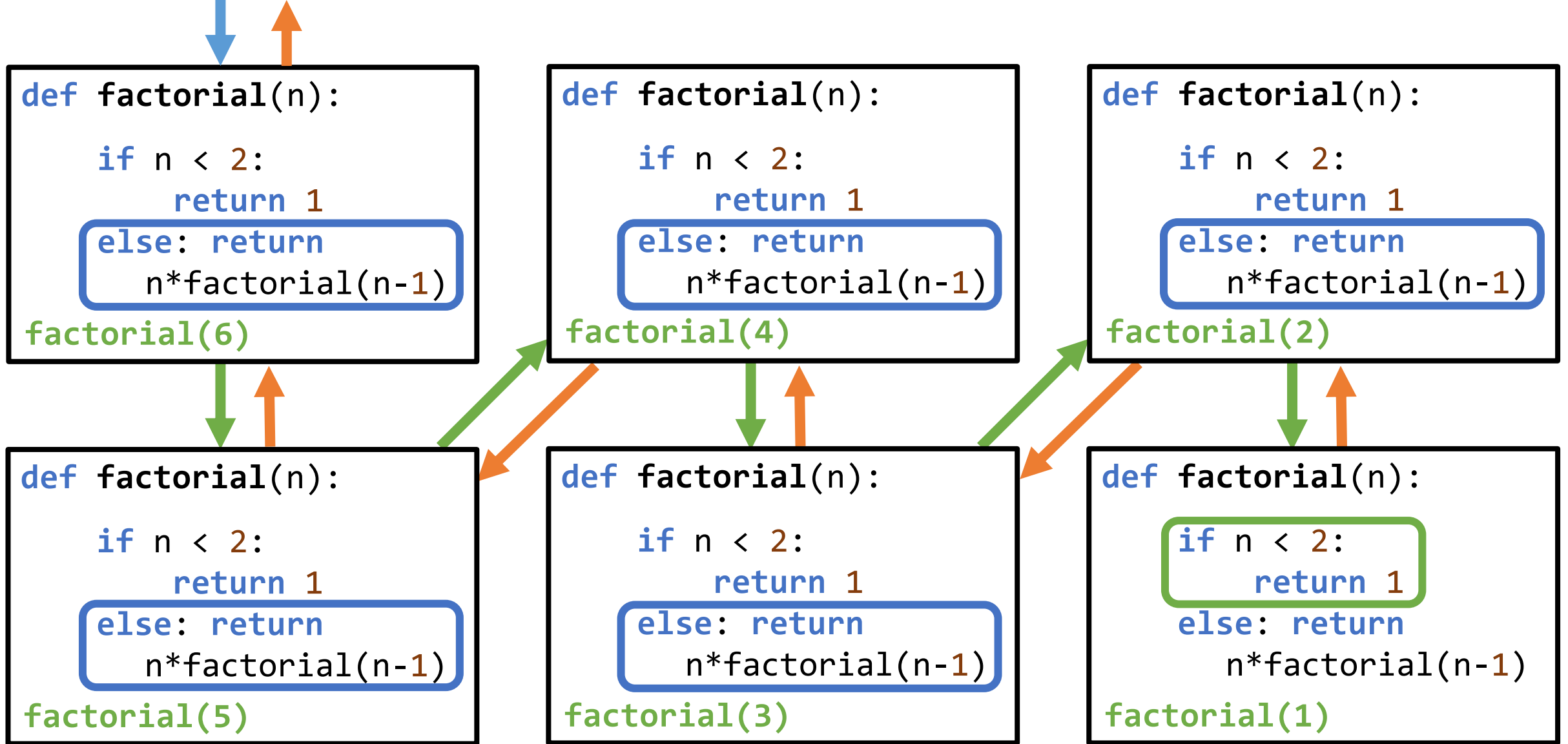
```
def factorial(n):  
    if n < 2:  
        return 1  
    else:  
        return n * factorial(n-1)
```

First call is to **factorial(6)**



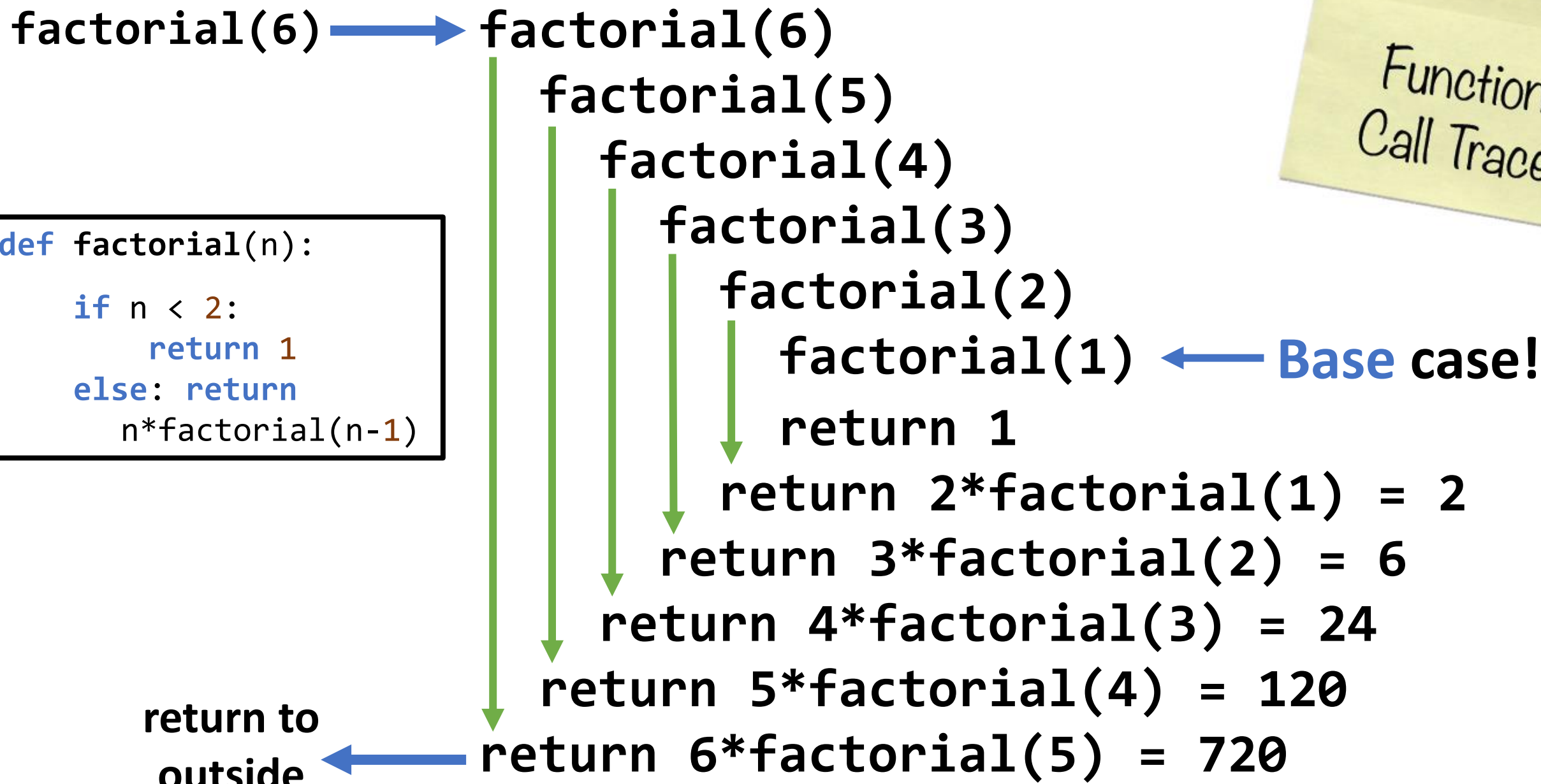
```
def factorial(n):  
    if n < 2: False!  
        return 1  
    else:  
        return n * factorial(n-1)
```

- **factorial(6)** calls **factorial(5)**
- We don't actually return yet!
- We *can't* return until **factorial(5)** is computed



- We have six *unique* function instances! Each has its own local scope.
- They are all **different** *instances* of the **same** *function*.

Function
Call Trace



Factorial: Iterative

```
(defun fact-do (n)
  (do ((fact n (* fact n))) ((= n 1) fact)
    (setf n (1- n))
  ))

(defun fact-dotimes (n)
  (let ((fact 1))
    (dotimes (i n fact)
      (setf fact (* fact (1+ i))))
  )))
```

Factorial: Recursive

```
(defun fact-rec (n)
  (if (< n 2) 1
      (* n (fact-rec (- n 1)))
  ) )
```

Base Case



Recursive Case




```
(defun fact-rec (n)
  (if (< n 2) 1
      (* n (fact-rec (- n 1)))
  ) )
```

```
(defun fact-do (n)
  (do ((fact n (* fact n))) ((= n 1) fact)
      (setf n (1- n))
  ))
```

```
(defun fact-dotimes (n)
  (let ((fact 1))
    (dotimes (i n fact)
      (setf fact (* fact (1+ i))))
  )))
```

```
* (fact-do 6)
720
* (fact-dotimes 6)
720
* (fact-rec 6)
720
```

Another! List Sum

Recurrence relation: Common way of describing recursive functions

$$\text{sum}(\text{items}) = \begin{cases} 0 & \text{if items is empty} \\ \text{first}(\text{items}) + \text{sum}(\text{rest}(\text{items})) & \text{otherwise} \end{cases}$$

**Base
Case**

```
(defun sum-rec (x)
  (if (null x) 0
      (+ (first x) (sum-rec (rest x)))))
)
```

Smaller problem

Recursive Case

```
(defun sum-rec (x)
  (if (null x) 0
      (+ (first x)
          (sum-rec (rest x)))))
)
```

```
* (trace sum-rec)
(SUM-REC)
```

```
* (sum-rec '(6 4 2 1 3 5))
0: (SUM-REC (6 4 2 1 3 5))
  1: (SUM-REC (4 2 1 3 5))
    2: (SUM-REC (2 1 3 5))
      3: (SUM-REC (1 3 5))
        4: (SUM-REC (3 5))
          5: (SUM-REC (5))
            6: (SUM-REC NIL)
              6: SUM-REC returned 0
            5: SUM-REC returned 5
          4: SUM-REC returned 8
        3: SUM-REC returned 9
      2: SUM-REC returned 11
    1: SUM-REC returned 15
  0: SUM-REC returned 21
```

21

*

Every recursive solution has a semantically equivalent iterative solution, and vice versa.

Most people find iteration more intuitive.
Why would we use recursion?

Linear Recursion

- In each of the previous examples, we made a single recursive call at each step.
- We call this linear recursion. Execution happens in a straight line, no *branching*.
- Any linear recursion can straightforwardly be converted to iteration.
- Aside from slightly smaller or simpler code, we didn't ***gain*** anything by using recursion.

Recursive Branching

The true power of recursion:

- The true power of recursion lies in the ability to branch in two or more directions.
- Contrast this with iteration - loops simply execute in a linear fashion until they are finished.
- Nested loops still execute in a single direction; In 2D, we move across the “rows” one at a time.
- To visualize, just imagine concatenating all the rows of a matrix end to end.

Recursive Divide and Conquer

Divide and Conquer is a recursive approach to algorithm design.

1. **Divide:** We break the problem into one or more smaller, self-similar subproblems.
2. **Conquer:** Solve one or more subproblems
3. **Combine:** Join the subproblem solutions into a solution to the original problem

Divide and Conquer

Mergesort and **Quicksort** are divide-and-conquer approaches to sorting:

Mergesort: Divide the list in half, recursively sort each half, merge sorted lists

Quicksort: Partition list into two sublists, recursively sort each half.

Both of these exhibit the *divide-in-half* approach

Divide and Conquer

Binary Search also takes a divide-in-half approach:

Binary search:

- Find midpoint (**divide**), thus creating two sub-problems (left and right lists)
- Test midpoint value
- Search **ONE** of the lists (**conquer**)

Same divide-in-half approach, but we solve one subproblem instead of two!

Divide and Conquer

Quickselect also takes a divide-in-half approach:

Quickselect (kth smallest):

- Partition list into two sublists.
- Compare pivot value with k.
- Recursively partition **ONE** sublist.

Same divide-in-half approach, but we solve one subproblem instead of two!

Divide and Conquer

Mergesort:	Split in half, solve both subproblems
Quicksort:	Split in half (we hope), solve both subproblems
Binary Search:	Split in half, solve ONE subproblem
Quickselect:	Split in half, solve ONE subproblem

In a future course (CPS616) you'll see many other divide and conquer forms.

- Split into 4, solve 3
- Split into 8, solve 7

And so on.

Divide and Conquer


What about our recursive sum?

- This is another common form of divide and conquer, called down-by-one D&C
- Create two problems, one of size **1** and another of size **$n-1$**
- Commonly, the problem of size 1 is its own solution
- What is the sum of a single value? It is simply that value.
- Factorial is another example of down-by-one D&C

```

* (sum-rec '(6 4 2 1 3 5))
  0: (SUM-REC (6 4 2 1 3 5))
    1: (SUM-REC (4 2 1 3 5))
      2: (SUM-REC (2 1 3 5))
        3: (SUM-REC (1 3 5))
          4: (SUM-REC (3 5))
            5: (SUM-REC (5))
              6: (SUM-REC NIL)
                6: SUM-REC returned 0
              5: SUM-REC returned 5
            4: SUM-REC returned 8
          3: SUM-REC returned 9
        2: SUM-REC returned 11
      1: SUM-REC returned 15
    0: SUM-REC returned 21
21
*

```



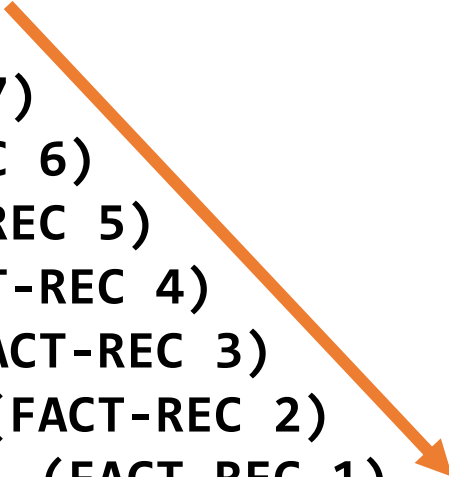
```

(defun sum-rec (x)
  (if (null x) 0
      (+ (first x)
          (sum-rec (rest x)))))
)

```

Problem size goes down by
one each recursive call

* (fact-rec 7)
0: (FACT-REC 7)
1: (FACT-REC 6)
2: (FACT-REC 5)
3: (FACT-REC 4)
4: (FACT-REC 3)
5: (FACT-REC 2)
6: (FACT-REC 1)
6: FACT-REC returned 1
5: FACT-REC returned 2
4: FACT-REC returned 6
3: FACT-REC returned 24
2: FACT-REC returned 120
1: FACT-REC returned 720
0: FACT-REC returned 5040
5040
*



```
(defun fact-rec (n)
  (if (< n 2) 1
      (* n (fact-rec (- n 1)))
  ) )
```

A bit trickier:

- Problem “size” is the *value* of an integer.
- Integer *value* goes down by one each recursive call.

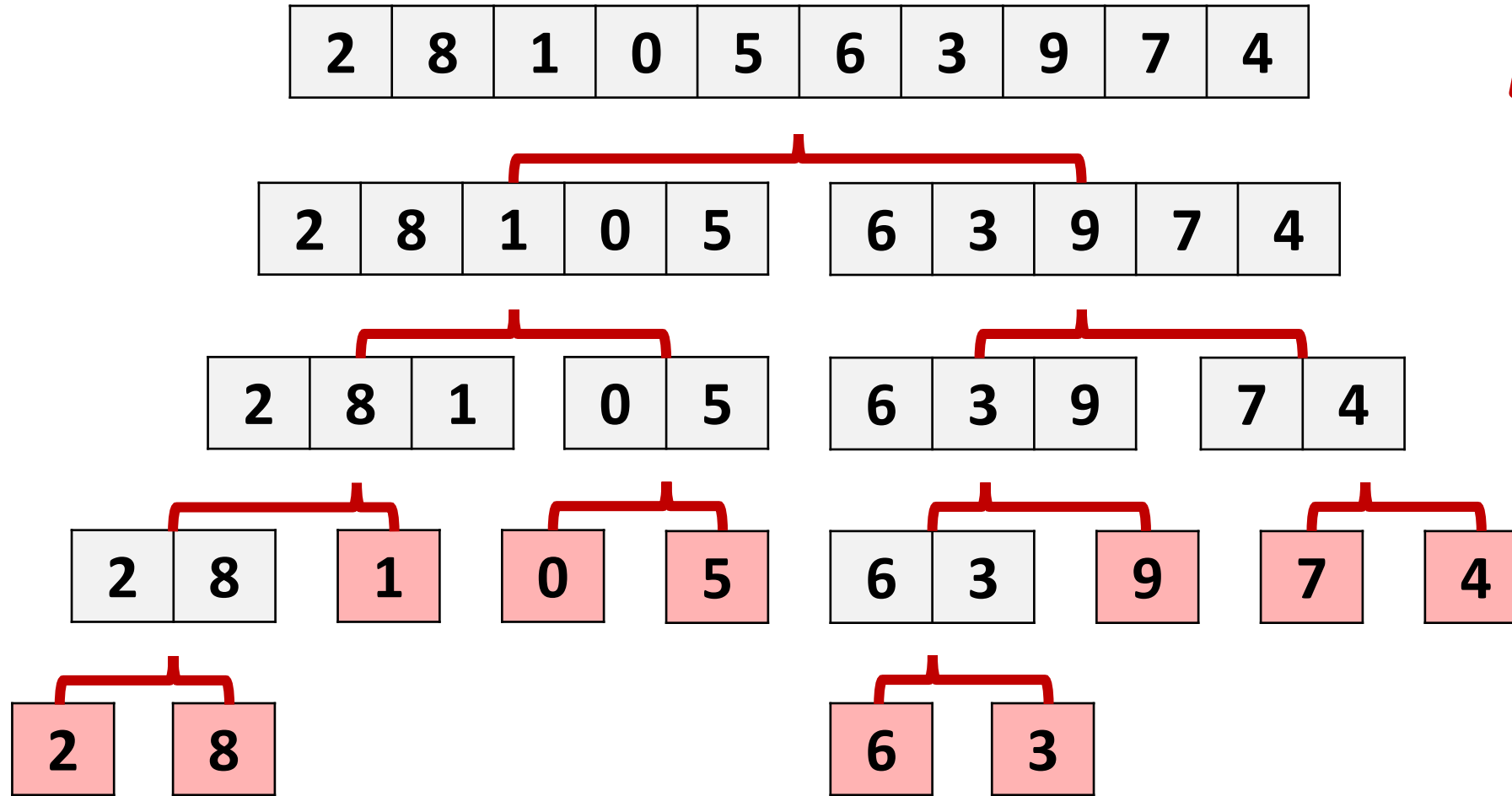
Sum: Division-in-Half?

What is the sum of a list?

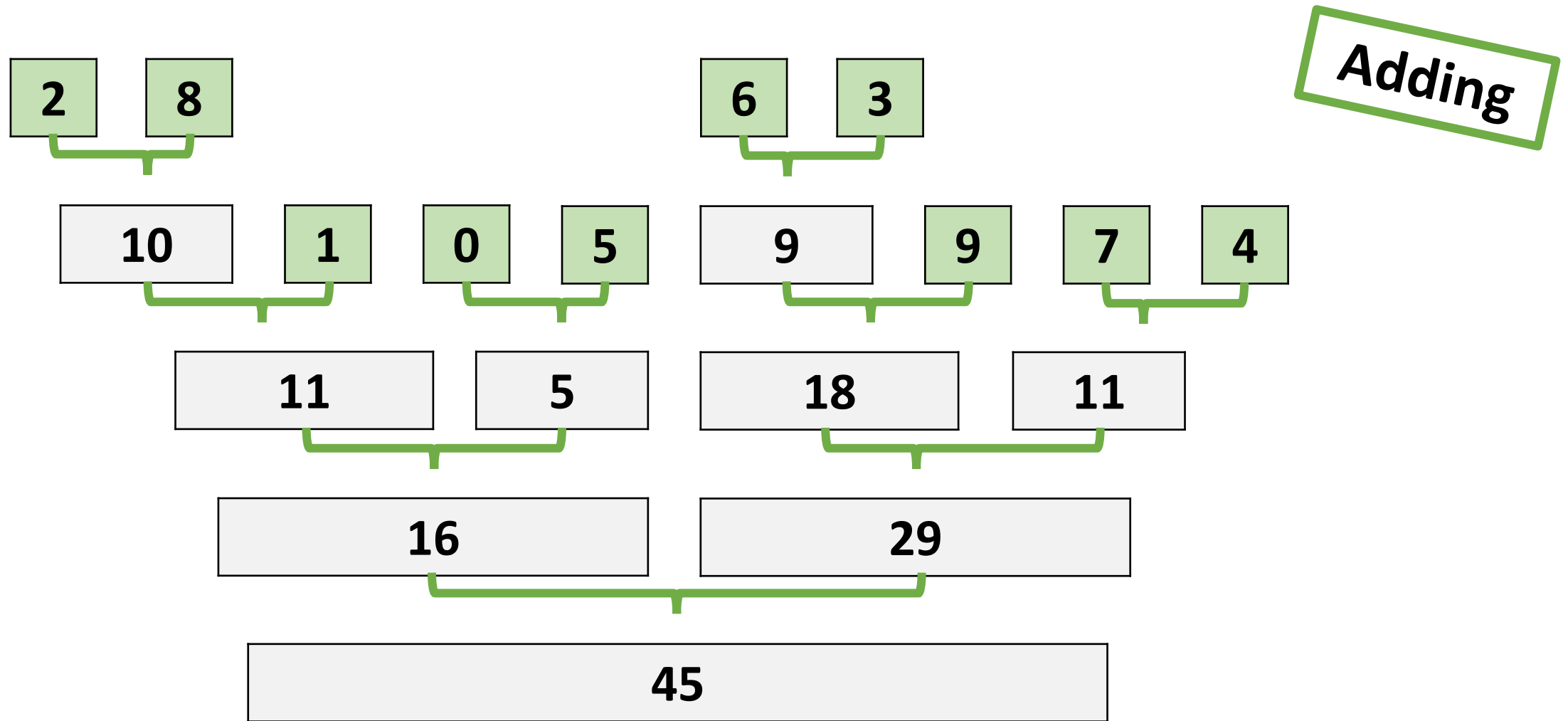
- The first element + the sum of the rest, Or...
- The sum of the left half + the sum of the right half

Just like Mergesort, except instead of the merge operation on the way back up, we just add elements

Recursive Splitting



Stop splitting when each subarray contains only a single element



When recursive split hits base case, we can begin adding.

```

(defun sum-half (x &optional (size (length x)))
  (cond ((null x) 0)
        ((= size 1) (first x))
        (t (let* ((half (floor size 2))
                  (rm (mod size 2))
                  (p (+ half rm)))
              (+ (sum-half (subseq x 0 p) p)
                 (sum-half (subseq x p) half))))
    )
)
)

```

Caveat: This is not “better”, still $O(n)$

- All we’ve done is add a ton of overhead
- We’re just demonstrating the approach

```

(defun sum-half (x &optional (size (length x)))
  (cond ((null x) 0)
        ((= size 1) (first x))
        (t (let* ((half (floor size 2))
                  (rm (mod size 2))
                  (p (+ half rm)))
              (+ (sum-half (subseq x 0 p) p)
                 (sum-half (subseq x p) half))))
        )
  )
)

```

*** (trace sum-half)**
(SUM-HALF)

```

* (sum-half '(1 3 5 7 9 8 6 4))
0: (SUM-HALF (1 3 5 7 9 8 6 4))
1: (SUM-HALF (1 3 5 7) 4)
2: (SUM-HALF (1 3) 2)
3: (SUM-HALF (1) 1)
3: SUM-HALF returned 1
3: (SUM-HALF (3) 1)
3: SUM-HALF returned 3
2: SUM-HALF returned 4
2: (SUM-HALF (5 7) 2)
3: (SUM-HALF (5) 1)
3: SUM-HALF returned 5
3: (SUM-HALF (7) 1)
3: SUM-HALF returned 7
2: SUM-HALF returned 12
1: SUM-HALF returned 16
1: (SUM-HALF (9 8 6 4) 4)
2: (SUM-HALF (9 8) 2)
3: (SUM-HALF (9) 1)
3: SUM-HALF returned 9
3: (SUM-HALF (8) 1)
3: SUM-HALF returned 8
2: SUM-HALF returned 17
2: (SUM-HALF (6 4) 2)
3: (SUM-HALF (6) 1)
3: SUM-HALF returned 6
3: (SUM-HALF (4) 1)
3: SUM-HALF returned 4
2: SUM-HALF returned 10
1: SUM-HALF returned 27
0: SUM-HALF returned 43

```

Length: Nested Lists?

`(count-nums '(11 ((12 (23 46)) 5 61) (3)))`

How many numbers are in this nested list?

- This is a very tricky problem to solve iteratively.
- Branching recursion to the rescue!

Length: Nested Lists?

`(count-nums '(11 ((12 (23 46)) 5 61) (3)))`

How many numbers are in this nested list?

```
(defun count-nums (a)
  (cond ((null a) 0)
        ((listp (first a)) (+ (count-nums (first a))
                               (count-nums (rest a))))
        (t (+ 1 (count-nums (rest a)))))
)
```

```
* (count-nums '(11 ((12 (23 46)) 5 61) (3)))
7
```

Sum: Nested Lists?


```
(sum-nums '(11 ((12 (23 46)) 5 61) (3)))
```

What is the *sum* of all numbers in the nested list?


```
(defun sum-nums (a)
  (cond ((null a) 0)
        ((listp (first a)) (+ (sum-nums (first a))
                               (sum-nums (rest a))))
        (t (+ (first a) (sum-nums (rest a)))))
)
```

```
* (sum-nums '(11 ((12 (23 46)) 5 61) (3)))
161
```

```
(defun count-nums (a)
  (cond ((null a) 0)
        ((listp (first a)) (+ (count-nums (first a))
                               (count-nums (rest a))))
        (t (+ 1 (count-nums (rest a)))))
  )
)
```



```
(defun sum-nums (a)
  (cond ((null a) 0)
        ((listp (first a)) (+ (sum-nums (first a))
                               (sum-nums (rest a))))
        (t (+ (first a) (sum-nums (rest a)))))
  )
)
```



Flatten a Nested List?

`(flatten '(11 ((12 (23 46)) 5 61) (3)))`
`-> (11 12 23 46 5 61 3)`

```
(defun flatten (a)
  (cond ((null a) a)
        ((listp (first a)) (append (flatten (first a))
                                     (flatten (rest a))))
        (t (cons (first a) (flatten (rest a)))))
)
```

```
* (flatten '(11 ((12 (23 46)) 5 61) (3)))
(11 12 23 46 5 61 3)
```


Dangers of Recursion

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

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

```
def factorial(3):  
    if n < 2:  
        return 1  
    else: return  
        n*factorial(n-1)
```

```
def factorial(4):  
    if n < 2:  
        return 1  
    else: return  
        n*factorial(n-1)
```

```
def factorial(5):  
    if n < 2:  
        return 1  
    else: return  
        n*factorial(n-1)
```

- Function calls create stack frames.
- Stack frame? A data structure containing a function's local data, parameters, variables, etc.
- Thus, function calls cause the stack to grow.

Dangers of Recursion

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

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

```
def factorial(3):  
    if n < 2:  
        return 1  
    else: return  
        n*factorial(n-1)
```

```
def factorial(4):  
    if n < 2:  
        return 1  
    else: return  
        n*factorial(n-1)
```

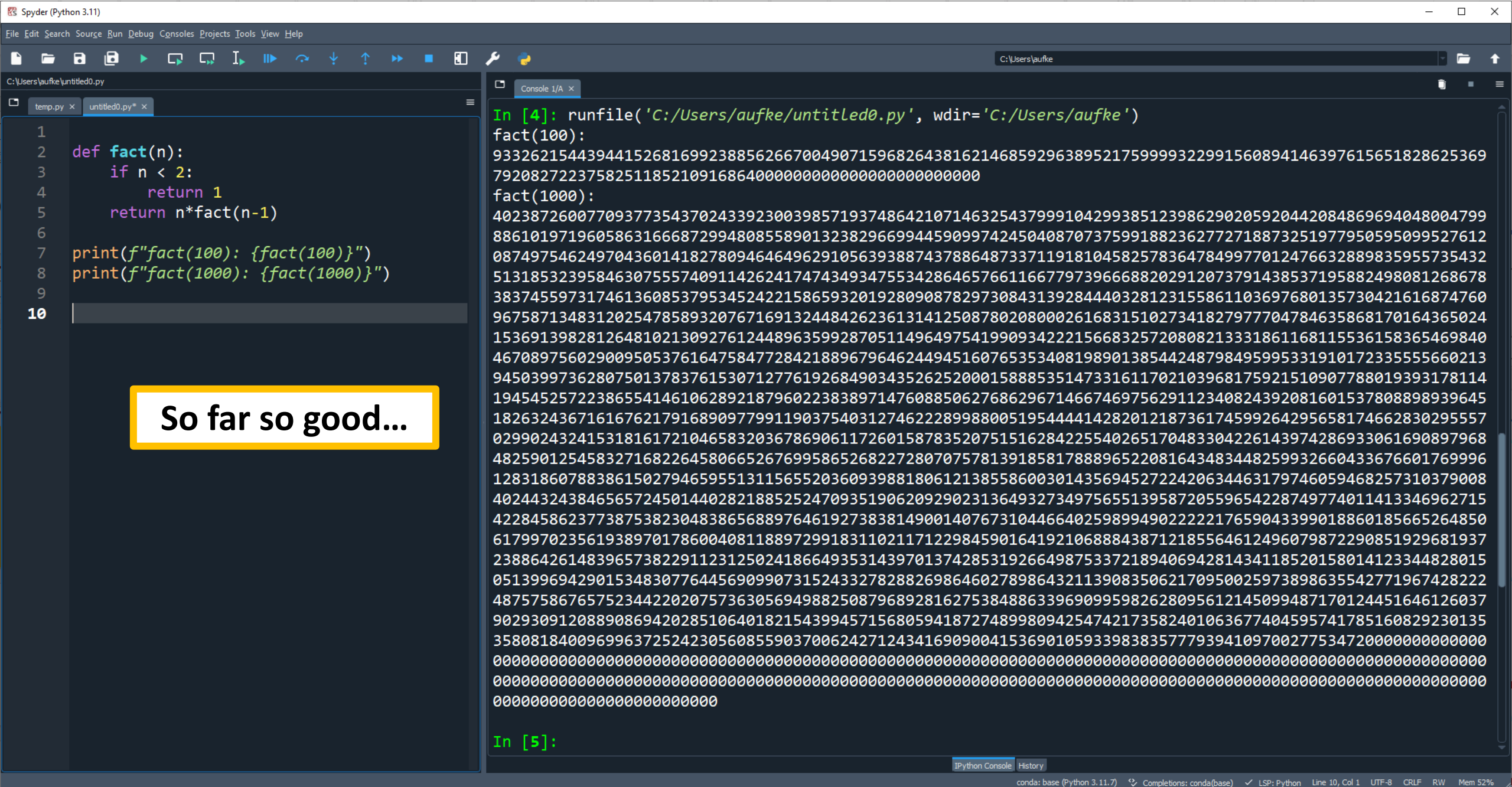
```
def factorial(5):  
    if n < 2:  
        return 1  
    else: return  
        n*factorial(n-1)
```

- Function calls create stack frames.
- Stack frame? A data structure containing a function's local data, parameters, variables, etc.
- Thus, function calls cause the stack to grow.
- When functions return, their frames are popped, and the stack shrinks back down.

Dangers of Recursion

- Too many nested function calls? Stack overflow!
- Stack size varies by language and program, but around 8Mb is common
- A common problem for deep recursion.





So far so good...

Dangers of Recursion

Python save us, forbids recursion past depth of 3000:

The screenshot shows the Spyder Python IDE interface. The left pane displays a Python script named `untitled0.py` with the following code:

```
1
2 def fact(n):
3     if n < 2:
4         return 1
5     return n*fact(n-1)
6
7 print(f"fact(100): {fact(100)}")
8 print(f"fact(1000): {fact(1000)}")
9 print(f"fact(10000): {fact(10000)}")
10
```

The right pane shows the console output, which includes the recursive calls and a `RecursionError`:

```
return n*fact(n-1)
File c:\users\aufke\untitled0.py:5 in fact
return n*fact(n-1)
File c:\users\aufke\untitled0.py:5 in fact
return n*fact(n-1)
RecursionError: maximum recursion depth exceeded
In [6]:
```

The error occurs because the recursive function `fact` is called with `n=10000`, which exceeds the maximum recursion depth of 3000 allowed by Python.

Dangers of Recursion

Isn't this limiting? Does it mean we can't use Mergesort on very large lists?

- How big would a list have to be to require 3000+ splits to get to the base case...?
- 2^{3000} elements. Such a list cannot exist in practice, according to the known laws of physics.
- *(There are between 10^{78} and 10^{82} atoms in the known universe)*

We're in no danger for the branching recursive algorithms we've seen so far.

Dangers of Recursion

```
(defun fact-rec (n)
  (if (< n 2) 1
      (* n (fact-rec (- n 1)))
  ) )
```

Long linear recursion in LISP?
* (fact-rec 1000000)

INFO: Caught stack overflow exception (sp=0x005c1ffc); proceed with caution.

debugger invoked on a SB-KERNEL::CONTROL-STACK-EXHAUSTED in thread
#<THREAD "main thread" RUNNING {23550049}>:

Control stack exhausted (no more space for function call frames).

This is probably due to heavily nested or infinitely recursive function calls, or a tail call that SBCL cannot or has not optimized away.

PROCEED WITH CAUTION.

Tail call?

Tail Recursion

Tail Recursion

- A function is said to be tail recursive if the recursive call is the last statement that executes.
- Put another way, nothing else is computed after the recursive call returns.

```
(defun fact-rec (n)
  (if (< n 2) 1
      (* n (fact-rec (- n 1)))))
)
```

This is NOT tail recursive!

- After (fact-rec (- n 1)) returns...
- We still must evaluate the product (* n fact-rec)

```
(defun fact-rec (n)
  (if (< n 2) 1
      (* n (fact-rec (- n 1)))))
)
```

Tail Recursion

How do we make it tail recursive? Typically, this involves passing the running result through the recursion as an argument:

- 2nd parameter, **f**, will hold the running result.
- Initialize to 1 in this case. Depends on problem.

```
(defun fact-tail (n &optional (f 1))
  (if (< n 2) f
      (fact-tail (- n 1) (* n f))))
)
```

Base case: Return **f**

Recursive case: Pass **n-1, **n*f****

Tail Recursion

```
(defun fact-tail (n &optional (f 1))  
  (if (< n 2) f  
      (fact-tail (- n 1) (* n f))  
  ) )
```

Another question you could ask:

- Can I print the final result in the base case?
- If so, you've most likely got tail recursion

So... why is tail recursion useful?

Tail Call Optimization

- Many languages, when they compile tail recursive functions, will convert them into ***iterative*** machine/byte code.
- We can write deep recursive functions in our source code...
- The compiler will optimize it into iteration behind the scenes
- New stack frames are no longer created for each call.

debugger invoked on a SB-KERNEL::CONTROL-STACK-EXHAUSTED in thread #<THREAD "main thread" RUNNING {23550049}>:

Control stack exhausted (no more space for function call frames).

This is probably due to **heavily nested or infinitely recursive function calls**, or a **tail call** that SBCL cannot or has not optimized away.

Tail Call Optimization

- Many languages support *Tail Call Optimization*, or TCO.
- LISP does, Elixir does (CPS506), Python famously does ***not***.
- It allows us to think recursively (well, *tail* recursively) without worrying about stack overflow!

Tail Recursion: Another Example

- 2nd parameter, **acc**, will hold the running sum.
- Initialize to 0

```
(defun sum-tail (x &optional (acc 0))  
  (if (null x) acc  
      (sum-tail (rest x) (+ (first x) acc))))  
)
```

Base case:
Return **acc**

Recursive case: Pass **rest** of
list, **acc+first of list**)

Tail Recursion: Another Example

Is this function tail recursive? Why or why not?

```
(defun mystery (x)
  (if (null x) x
      (append (mystery (rest x)) (list (first x)))))
)
```

No! The append form evaluates after the recursive mystery call.

- What is this function doing, and can we implement it tail recursively?
- It's reversing the list, and yes we can!

Tail Recursion: Another Example

```
(defun tr-reverse (x &optional (acc nil))  
  (if (null x) acc  
      (tr-reverse (rest x) (cons (first x) acc))))  
)
```

Same pattern as the others:

- Optional parameter, acc, holds running result (a list in this case)
- Pass in rest of list, cons first with acc (update running result)

Recursion VS Iteration

Entirely problem dependent:

- Some tasks are far easier to solve recursively, others are far easier to solve iteratively.
- One (or few) base cases? Obvious recursive case? Try recursion.
- Recursive functions can be very elegant, but don't force it

Branching Recursion

We've seen a few algorithms already:

- Mergesort, Quicksort, Quickselect
- These can be implemented iteratively, but it's quite a bit trickier.
- Branching recursion lets us offload a lot of bookkeeping to the call stack.
- The result? The most efficient sorting algorithms possible – $O(n \log n)$

Branching Recursion + Data Structures

Many data structures are recursively defined:

- Nested lists, for example:

```
(defun flatten (a)
  (cond ((null a) a)
        ((listp (first a)) (append (flatten (first a))
                                     (flatten (rest a))))
        (t (cons (first a) (flatten (rest a)))))
  )
)
```

Branching Recursion + Data Structures

Many data structures are recursively defined:

- Nested lists, for example.
- Tree and graph traversal are also natural applications of recursion.
- We'll see these data structures later in the course.

Branching Recursion: Dangers

- Tail Call optimization can combat the danger of stack overflow during linear recursion...
- Branching recursion comes with its own set of dangers.

Consider...

Fibonacci Numbers



- Each Fibonacci number is the sum of the **previous two**.
- We cannot compute the **20th** Fibonacci number without first computing the **18th** and **19th**
- Solving the larger sub-problem requires solving **TWO** self-similar sub-problems!
- Branching recursion to the rescue...?

```
(defun fib-rec (n)
  (if (<= n 2) 1
      (+ (fib-rec (- n 1))
          (fib-rec (- n 2)))
  ) )
```

Looks good...? Right...?

```
* (fib-rec 1)
1
* (fib-rec 2)
1
* (fib-rec 3)
2
* (fib-rec 4)
3
* (fib-rec 5)
5
* (fib-rec 6)
8
* (fib-rec 7)
13
* (fib-rec 8)
21
* (fib-rec 9)
34
```

```
(defun fib-rec (n)
  (if (<= n 2) 1
      (+ (fib-rec (- n 1))
          (fib-rec (- n 2)))
  ) )
```

Try this:

*** (fib-rec 100)**

This call will not return in your lifetime, and that's a vast underestimate.

Recursive Fibonacci

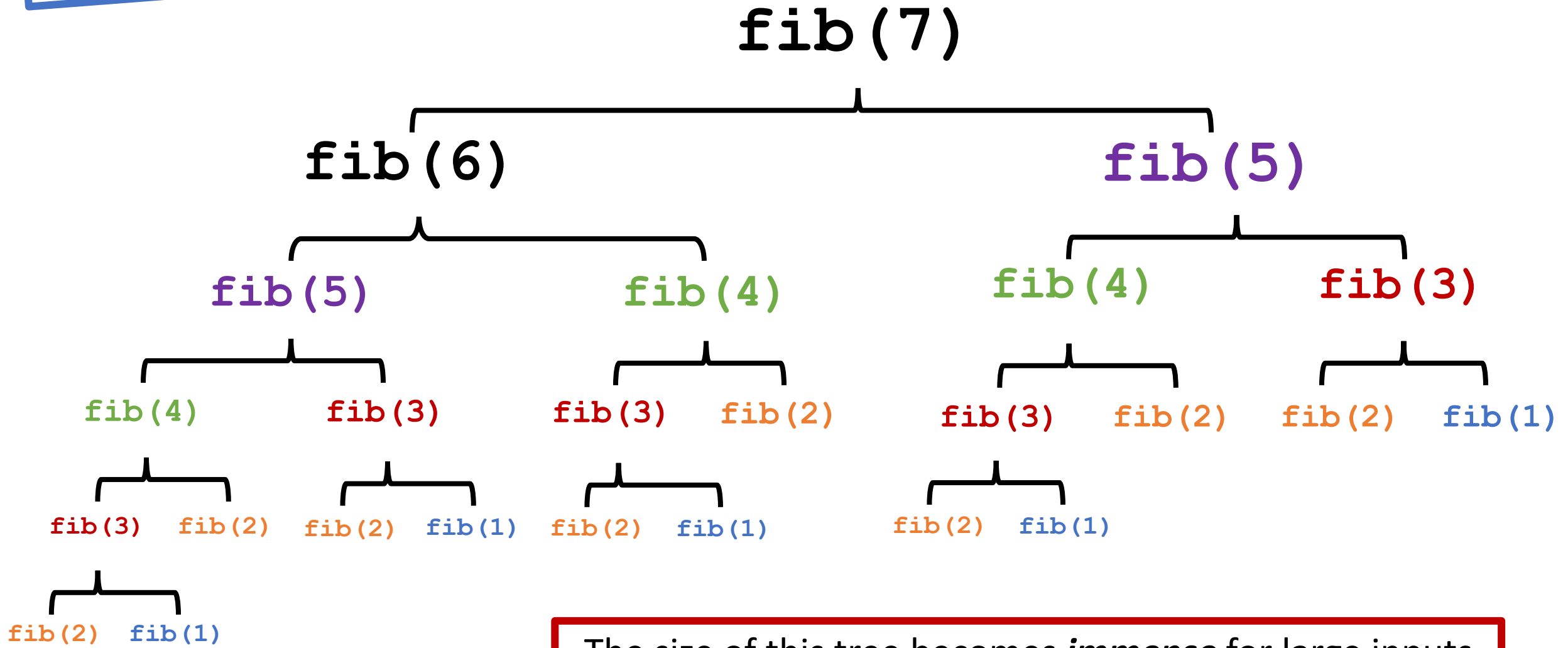
`(fib n) = (+ (fib (- n 1)) (fib (- n 2)))`

```
(fib 10) = (+ (fib 9) (fib 8))
(fib 9)  = (+ (fib 8) (fib 7))
(fib 8)  = (+ (fib 7) (fib 6))
(fib 7)  = (+ (fib 6) (fib 5))
(fib 6)  = (+ (fib 5) (fib 4))
(fib 5)  = (+ (fib 4) (fib 3))
(fib 4)  = (+ (fib 3) (fib 2))
(fib 3)  = (+ (fib 2) (fib 1))
(fib 2)  = 1
(fib 1)  = 1
```

Who sees the issue?

- Finding the 10th makes two recursive calls, to find 9th and 8th.
- Finding 9th also makes two recursive calls, to find 8th and 7th
- We're finding the 8th number twice here!
- 10th needs 8th, as does 9th
- These are called ***overlapping subproblems***

Overlapping Subproblems?



The size of this tree becomes *immense* for large inputs

But computers are super fast...?

	$\log(n)$	n	n^2	2^n	Branching recursive Fibonacci
$n = 8$	3	8	64	256	
16	4	16	256	65536	
32	5	32	1024	4294967296	
64	6	64	4096	1.845^{19}	
128	7	128	16384	3.403^{38}	

- In nanoseconds, this is around 10,000,000,000,000,000,000,000 years.
- Stars and galaxies won't exist anymore.

Recursive Fibonacci: Overlapping Subproblems

- Avoiding overlapping subproblems for Fibonacci is easy enough, just iterate instead.
- If we insist on recursion, we can use tail recursion.
- Pass the previous Fibonacci terms through as arguments, rather than recomputing them over and over again.
- Not so simple for other more complex problems.
- Additionally, branching recursion ***in no way*** promises there will be overlapping subproblems (mergesort).
- This is a large topic in a future course, CPS616

A detailed close-up photograph of several interlocking mechanical gears. The gears are made of metal and are arranged in a complex, overlapping pattern. The lighting is dramatic, with strong highlights and deep shadows, emphasizing the metallic texture and the precision of the gear teeth. A semi-transparent white horizontal band is overlaid across the center of the image, serving as a background for the title text.

Shifting Gears

New ADT: Associative List

- Based on the idea of ***Key-Value*** pairs
- Different names in different languages: Map, Dictionary, HashMap, Associative Array, etc.
- Given a key, return the value associated with that key
- Basic arrays can be thought of as a special form of this, where the keys are integer indexes, and the values are the elements.

Association Lists in LISP

An **a-list** is a list of **cons pairs**:

```
* (cons "foo" 'bar)  
("foo" . BAR) ; One pair
```



Association Lists in LISP

An **a-list** is a list of **cons pairs**:

```
* (cons "foo" 'bar)
("foo" . BAR) ; One pair
* (list (cons "foo" 'bar))
(("foo" . BAR)) ; a-list with one pair
* (list (cons "foo" 'bar) (cons "baz" 'bin))
(("foo" . BAR) ("baz" . BIN)) ; a-list with two pairs
*
```


A-LIST: Lookup

Use **ASSOC** form. Arguments are key, and a-list:

```
* (defvar *words* '((one . un) (two . deux)
(three . trois) (four . quatre) (five . cinq)))
* (assoc 'four *words*)
(FOUR . QUATRE)
```

- By default, ASSOC uses EQ to compare keys
- Use something different? EQUAL for string keys:

```
* (assoc "a" '(("b" . 2) ("a" . -1) ("c" . 5)) :test 'equal)
("a" . -1)
```

A-LIST: Lookup

Use **ASSOC** form. Arguments are key, and a-list:

```
* (defvar *words* '((one . un) (two . deux)
(three . trois) (four . quatre) (five . cinq)))
* (assoc 'four *words*)
(FOUR . QUATRE)
* (car (assoc 'four *words*))
FOUR
* (cdr (assoc 'four *words*))
QUATRE
```

Use CAR/CDR to access key/value from association

A-LIST: Add Entry

CONS or PUSH

```
* (defvar *words* '((one . un) (two . deux) (three . trois)))  
* (push '(six . six) *words*)  
((SIX . SIX) (ONE . UN) (TWO . DEUX) (THREE . TROIS))  
* (setf *words* (cons (cons 'seven 'sept) *words*))  
((SEVEN . SEPT) (SIX . SIX) (ONE . UN) (TWO . DEUX) (THREE .  
TROIS))
```

Notice: PUSH mutates, CONS does not.

- Must reassign when using CONS

A-LIST: Remove Entry

*** *words***

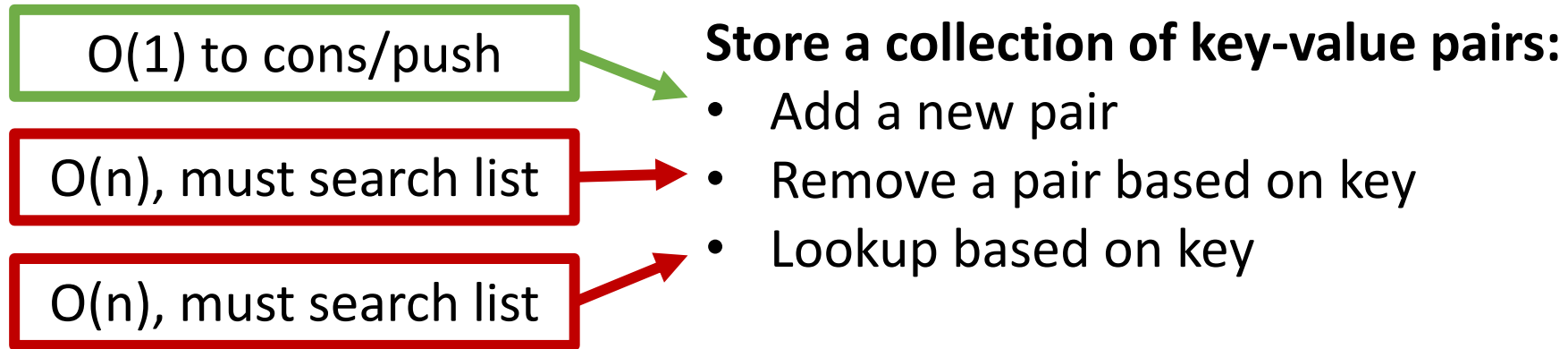
((SEVEN . SEPT) (SIX . SIX) (ONE . UN) (TWO . DEUX) (THREE . TROIS))

*** (remove 'three *words* :key 'car)**

((SEVEN . SEPT) (SIX . SIX) (ONE . UN) (TWO . DEUX))

- Must tell the remove form to perform the removal based on CAR (first element)
- (Rather than 2nd element, CDR)

ADT: Associative List



- The A-LIST in LISP supports these operations, and is implemented as a **linked list** of **cons pairs**
- **Given this implementation, what is the cost of each operation?**
- We can do better. All three of these can be done in **$O(1)$** , with the right base (*not* a linked list!)

Hash Tables

New ADT: Hash Table

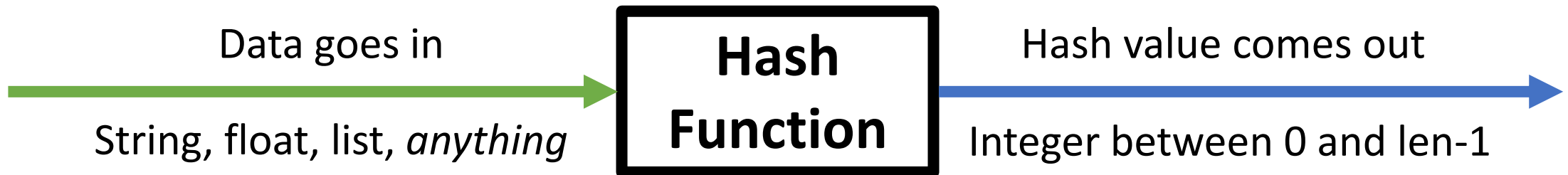
The properties of a Hash Table are like those of a Set:

- No duplicates, order is not preserved
- In fact, sets and dictionaries (associative arrays) are typically built on hash tables.
- A hash table provides $O(1)$ lookup, insertion, and removal.
- The BASE of a hash table is an array, rather than a linked list.
- Before learning about Hash Tables, we must understand *hash functions* that produce *hash values*.

Hash Functions, Hash Values

Fundamentally:

- A hash **function** transforms some arbitrary data into an integer value to be used as an array index.
- If our array has 1000 elements, the hash value must be between 0 and 999
- Different data will benefit from different hash functions.



Hash Function: Required Properties

Hash Function

Determinism:

- A hash a function, when given the same input, must produce the same output.
- Hash function should ***not*** depend on mutable global variables, random values, etc.

Caveat:

- The same input always produces the same output, **HOWEVER...**
- Different input can also produce the same hash code!

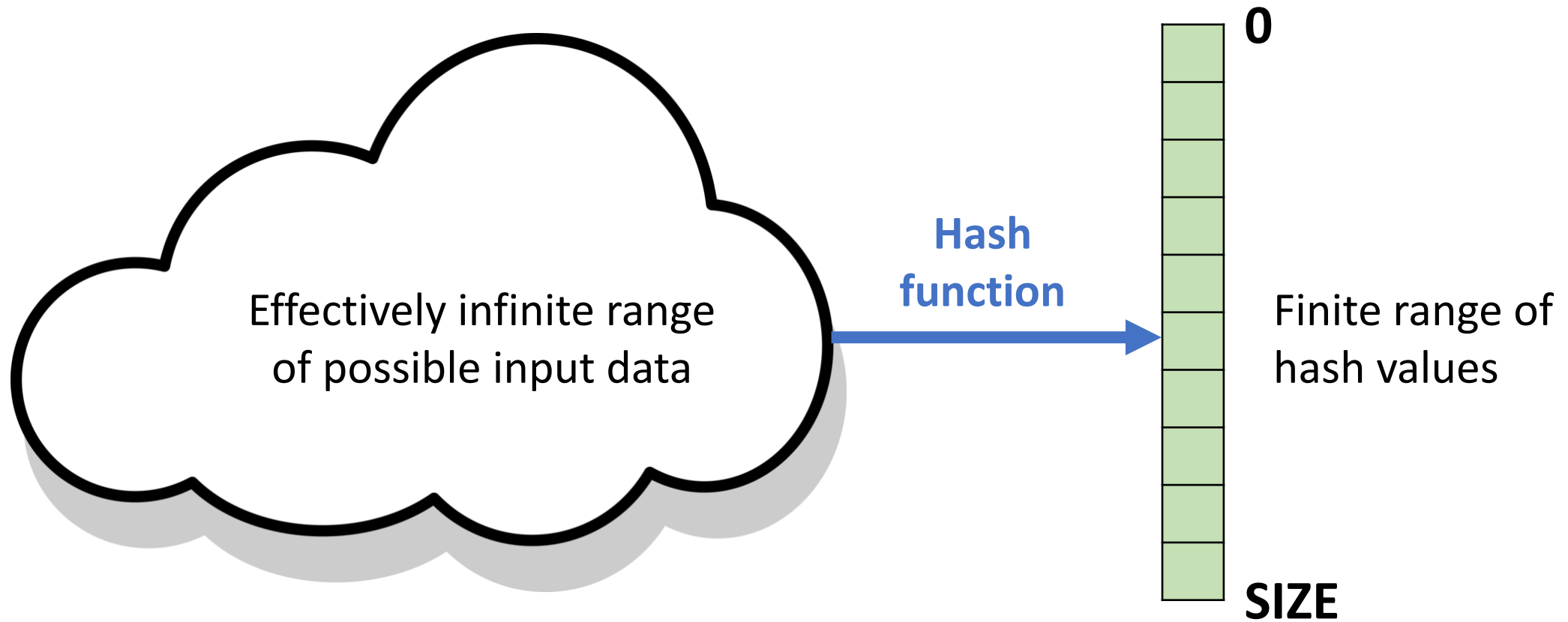
Hash Function: Required Properties

Hash Function

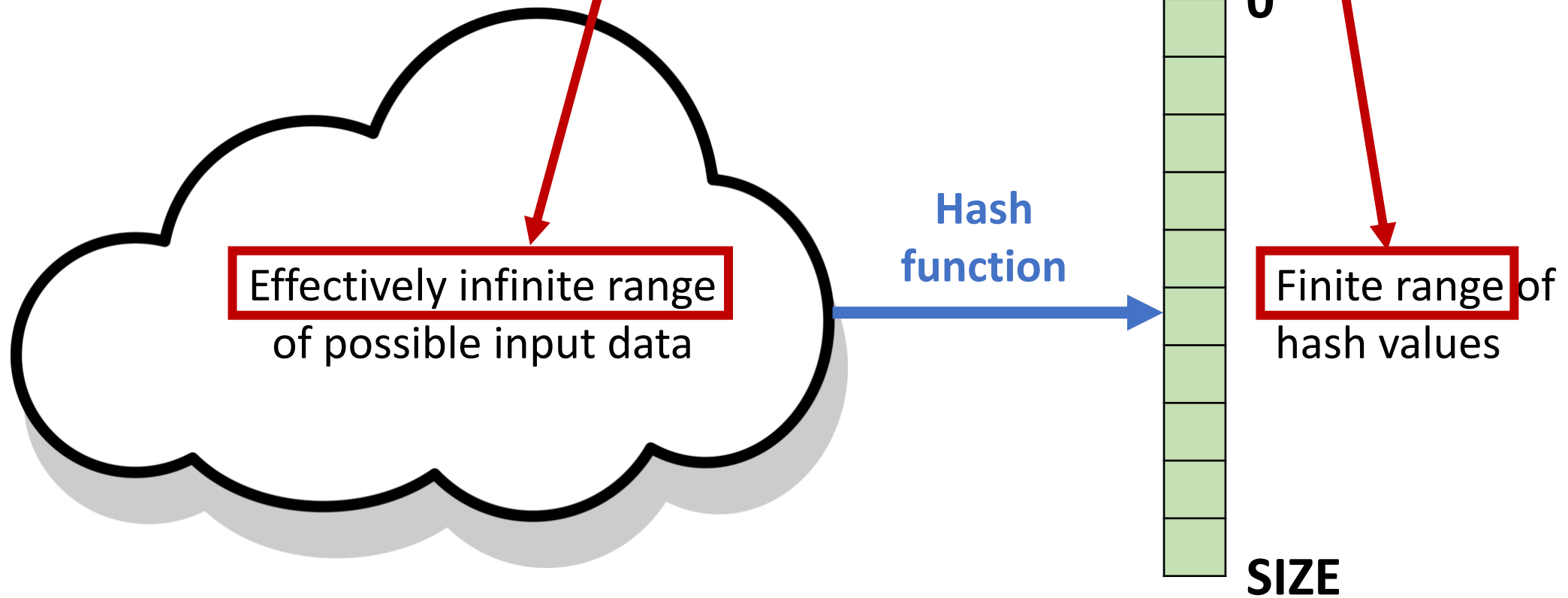
Fixed Range:

- Hash functions should produce values within a fixed range.
- Typically, **0** to **TABLE_SIZE-1**
- This allows hash codes to be used as indexes into arrays.

Hash Function: Required Properties

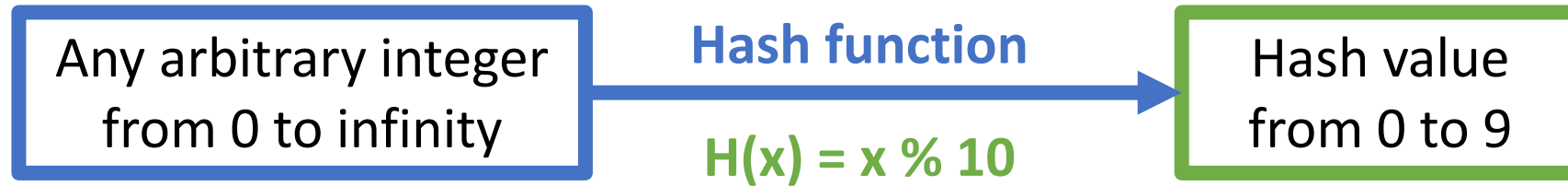


Different input can also produce the same hash code!



Hash Function: Simple Example

The humble remainder function:



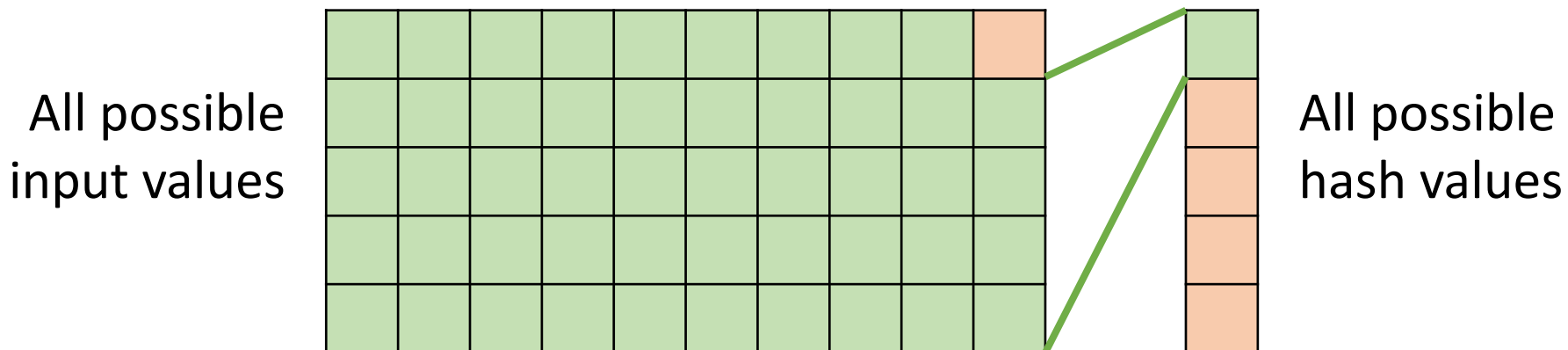
This is a very simple hash function

- If x is our original key, only the rightmost digit is contributing to the hash value.
- A good hash function ensures that the entire key contributes to the hash value.

Hash Function: Desirable Properties

Uniformity:

- A good hash function should map the expected range of all possible inputs to a uniformly distributed series of hash values.
- A hash function that maps 95% of its input into 5% of the possible hash values is going to degrade performance.



98% of input data maps to 20% of possible hash values. **BAD!**

Hash Function: Desirable Properties

Avoid Clustering

- Similar keys should ***not*** produce similar hash values.
- This will cause clusters to form when hashing similar keys.
- When clusters form, odds of a collision resulting in a long probe goes up (***what does this mean? Coming up***)

Hash Function: Summary

1) Deterministic

Required!

- Hash function should always return the same value for the same key.

2) Fixed Range

- Hash function should return some value between **0** and **TABLE_SIZE**.

3) Use Entire Key

Nice to have

- Entire key should contribute to hash value.

4) Uniform Distribution

- Hash function should map expected key values uniformly over hash values

5) Avoid Clustering

- Similar keys should not produce similar hash values.
- This will cause clusters to form when hashing similar keys.

Hash Function: One Last Requirement?

- The hash value should be *efficient* to compute.
- Once we have a hash value, lookup is $O(1)$
 - Not quite that simple, depends on collision resolution.
 - More on collisions coming up.
- If computing hash is $O(n^2)$, we're already sunk.

Hash Values ✓

produced by

Hash Functions ✓

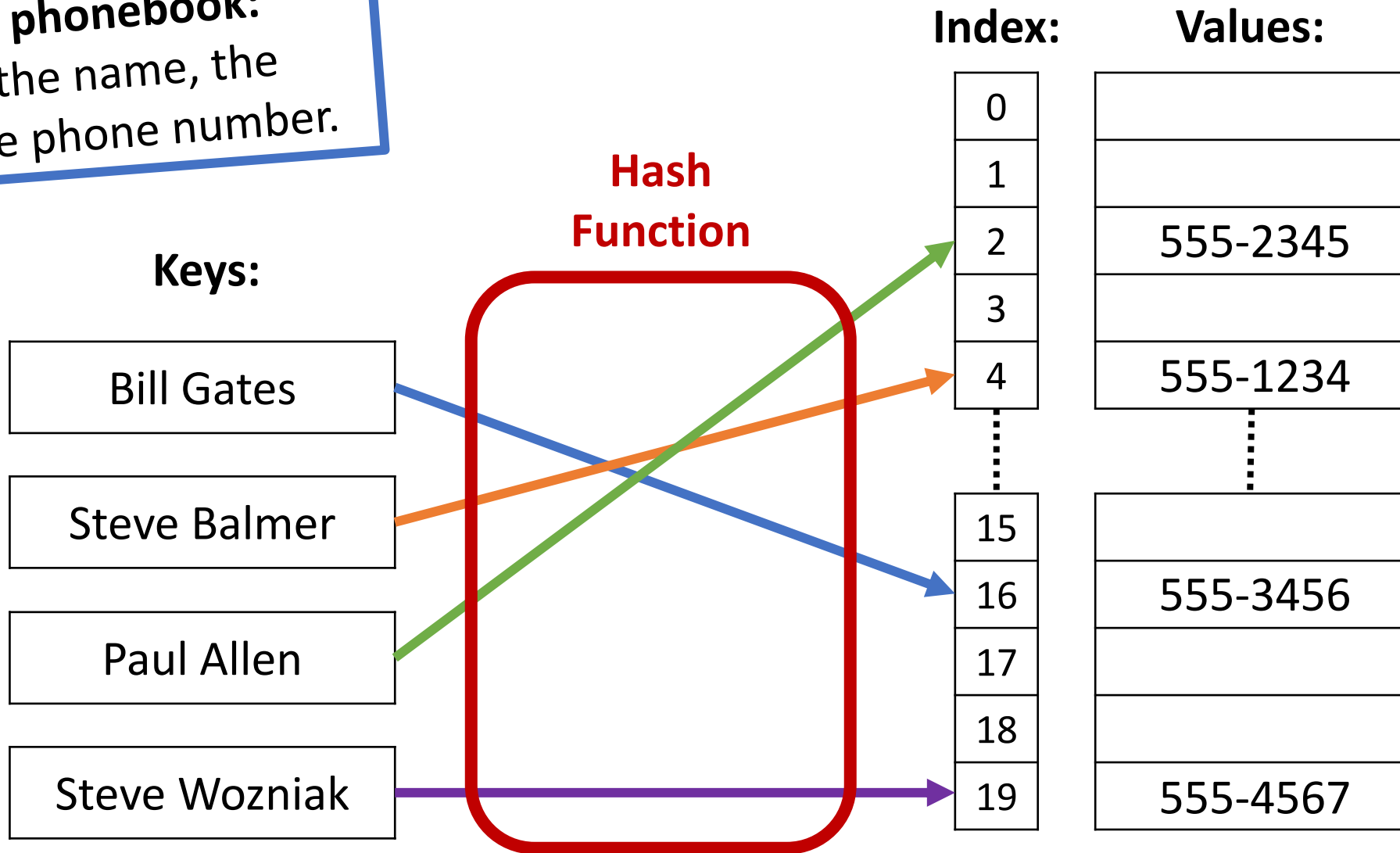
index into

Hash Tables ?

Hash Tables

- An associative array that maps *keys* to *values*
- The contents of the array are the values.
- The key is used to index into the array and obtain its value.
- However! The keys are not used as indexes *directly*.
- We apply a hash function to the key, which produces a hash value
- This hash value is used as an index.

Consider a phonebook:
The key is the name, the
value is the phone number.



**There's lots more to say
about hash tables.**

Next class.

In Summary

Recursion:

- Stack overflow
- Tail recursion, tail call optimization
- Branching recursion, overlapping subproblems

Keys and Values:

- A-LISTS in LISP
- Hash functions, hash values

