Recursion

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

Lecture Outline

- Recursion: Basic Idea
- Iteration versus Recursion
- How Recursion Works
- Recursion: How to
- More Examples on Recursion
  - Printing a Linked List
  - Printing a Linked List in Reverse
  - Choosing *k* out of *n* Items
  - Tower of Hanoi
  - Fibonacci Numbers

Recursion: Basic Idea

- The process of solving a problem with a function that **calls itself** directly or indirectly
  - The solution can be derived from solution of smaller problem of the same type
- Example:
  - `Factorial(4) = 4 * Factorial (3)`
- This process can be repeated
  - `Factorial(3)= 3 * Factorial(2)`
- Eventually, the problem is so simple that it can solve immediately
  - `Factorial(0) = 1`
- The solution to the larger problem can then be derived from this …

Recursion: The Main Ingredients

Base Case

- Identify the "**simplest**" instance that we can solve *without* recursion

Recursive Case: Sub-Problem

- Identify "**simpler**" instances of the same problem that we can make recursive calls to solve

Recursive Case: Build-up

- Identify how the solution from the simpler problem can help to construct the final result

- Check that we are able to reach the "**simplest**" instance to avoid **infinite recursion**

Example: Factorial

- Lets write a recursive function `factorial(k)` that finds `k!`

Base Case

- Return `1` when `k = 0`
- Corresponds to this bit of Python code:

```
if k == 0:
    return 1
```

Recursive Case

- Return `k * (k-1)!`

```
return k * factorial(k-1)
```

Note the "sub-problem" and "build-up"

Example: Factorial (code)

- Full code for `factorial`:

```
def factorial( k ):
    if k == 0:
        return 1

    return k * factorial(k-1)
```

Base Case:

factorial(0) = 1

Recursive Case:

factorial(k) = k * factorial(k – 1)

```
def factorial( k ):
    if k == 0:
        return 1
    else:
        return k * factorial(k-1)
```

Alternative way to write:

Note the "`else:`" part, the version above is preferred as it is more readable especially when there are many return paths

Understanding Recursion

- A recursion always goes through two phases:
  1. A **wind-up phase:**
     - When the **base case** is *not* satisfied i.e. the function calls itself
     - This phase carries on **until** we reach the **base case**
  2. An **unwind phase**:
     - The recursively called functions return their values to previous "instances" of the function call
     - Eventually reaches the very first function, which computes the **final value**

Factorial: **Wind-up Phase**

- Let's trace the execution of `factorial(3)` (`factorial` abbreviated as **`fact`**)

`fact( 3 )`
`k = 3`

***k* is not zero**
`returns 3 * fact( 2 )`

`fact( 2 )`
`k = 2`

***k* is not zero**
`returns 2 * fact( 1 )`

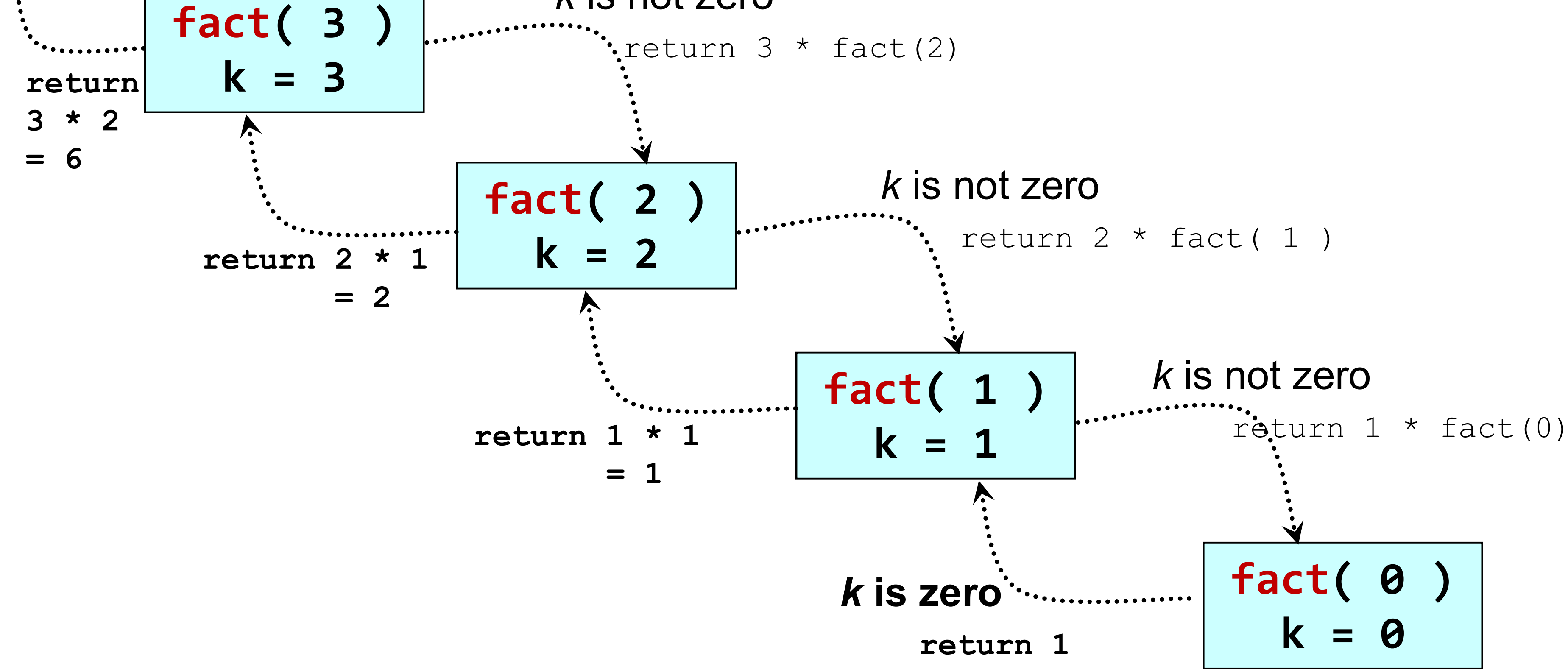
`fact( 1 )`
`k = 1`

`fact( 0 )`
`k = 0`

```
def factorial( k ):
    if k == 0:
        return 1

    return k * factorial(k-1)
```

Factorial: **Unwind Phase**

```python
def factorial( k ):
    if k == 0:
        return 1

    return k * factorial(k-1)
```



Recursions vs. Loops

- Most recursions essentially accomplishes a loop (iterations)
  + Recursions are usually much **more elegant** than its iterative equivalent
  + Recursions are **conceptually simpler and easier to implement**
  - Iterative version using loops is **usually faster** and use **less memory**

- Common practice:
  - Figure out the solution using recursion
  - Convert to iterative version if ***feasible***

Recursive vs. Iterative Versions

```
def factorial( k ):

    result = 1
    for i in range(2, k+1 ):
        result *= i
    return result
```

Iterative Version

```
def factorial( k ):
    if k == 0:
        return 1

    return k * factorial(k-1)
```

Recursive Version

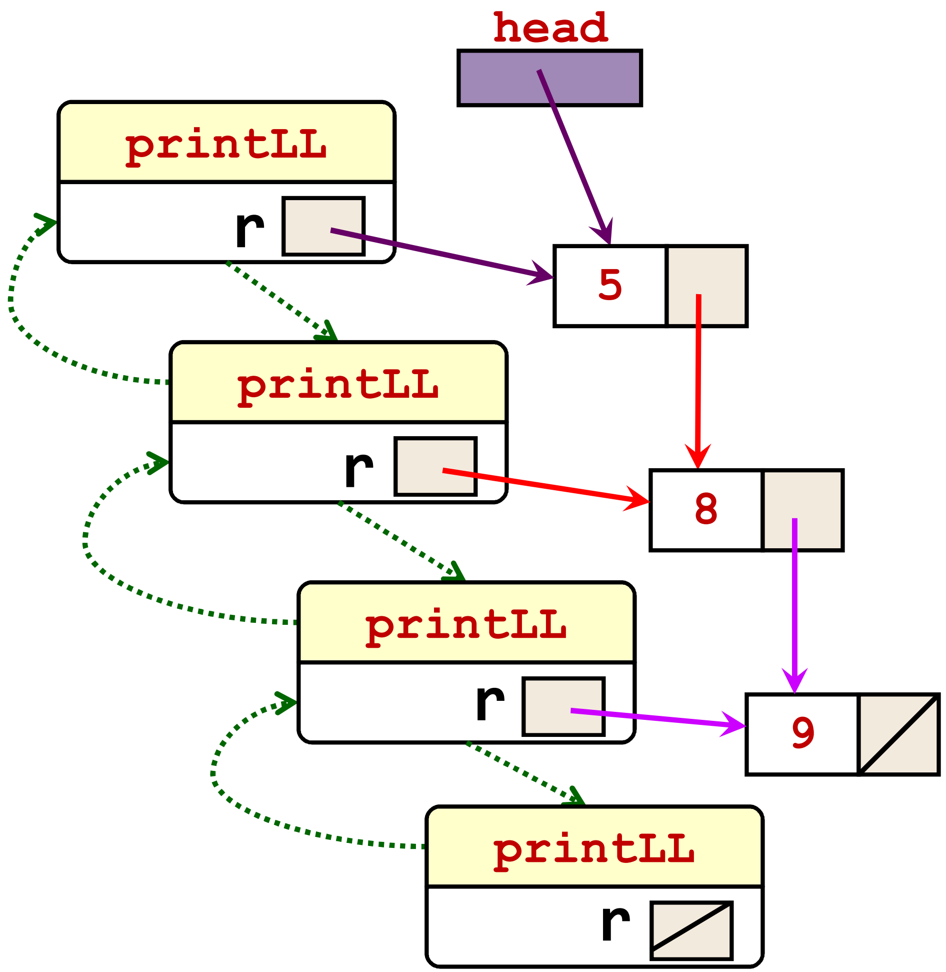
MORE examples to convince your brain ☺

RECURSION EXAMPLES

Example : **Linked List Printing**

- Print out the whole linked list given the pointer to a `SinglyNode` (from Lecture 4)

```
def printLL(r):
    if r != None:
        print( r.item )
        printLL( r.next )
```



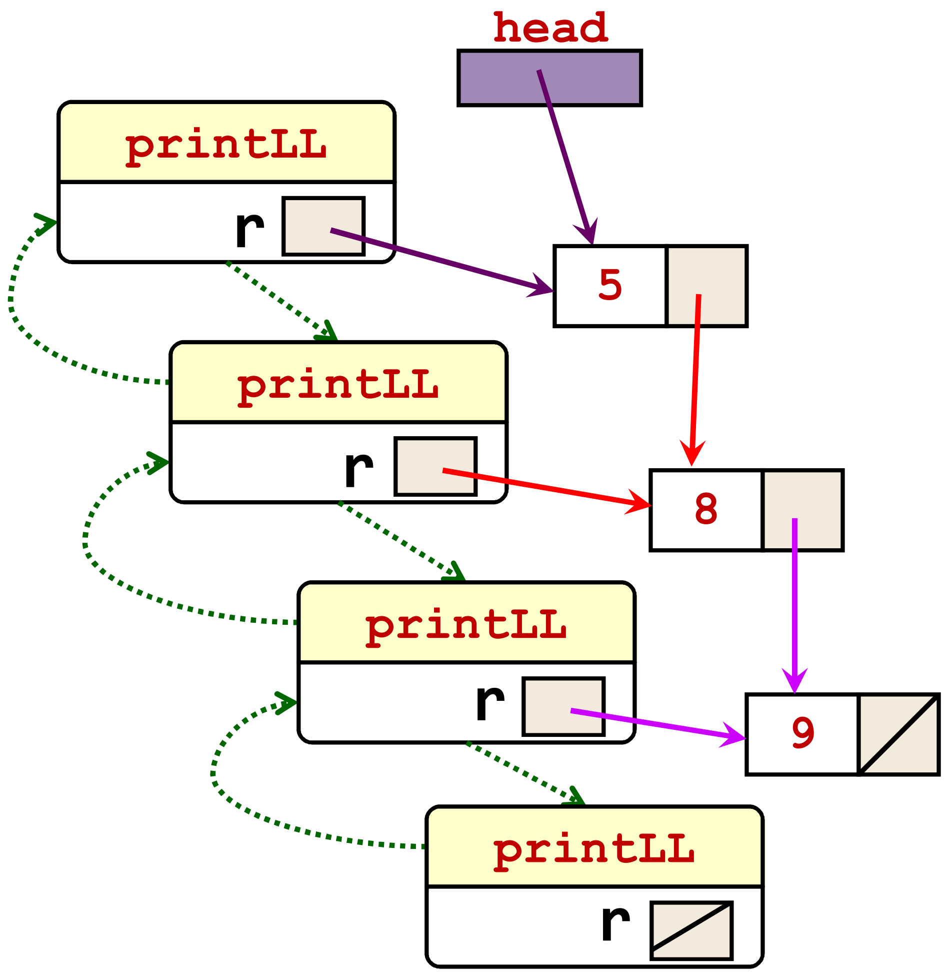
Output:

5 8 9

Example: **Linked List Printing II**

- How to print out the whole list in **reverse order**?

```
def printLL(r):
    if r != None:
        printLL(r.next)
        print(r.item )
```

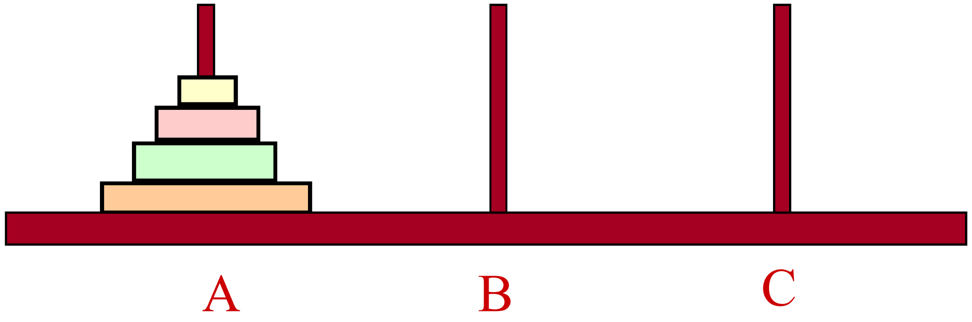


Output:

9 8 5

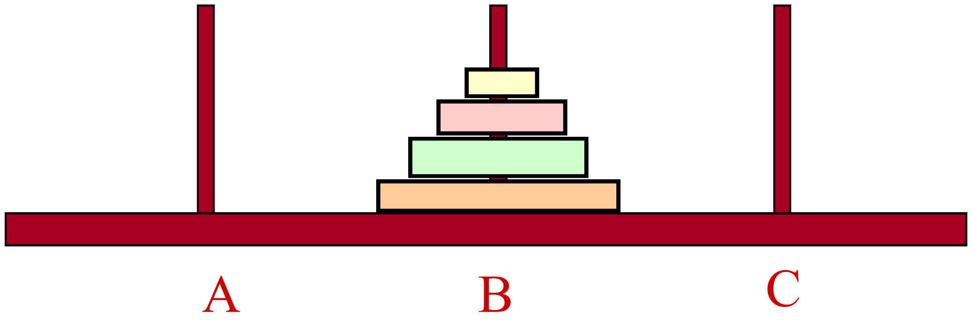
Example: **Tower of Hanoi (Stack ADT)**

Initial state

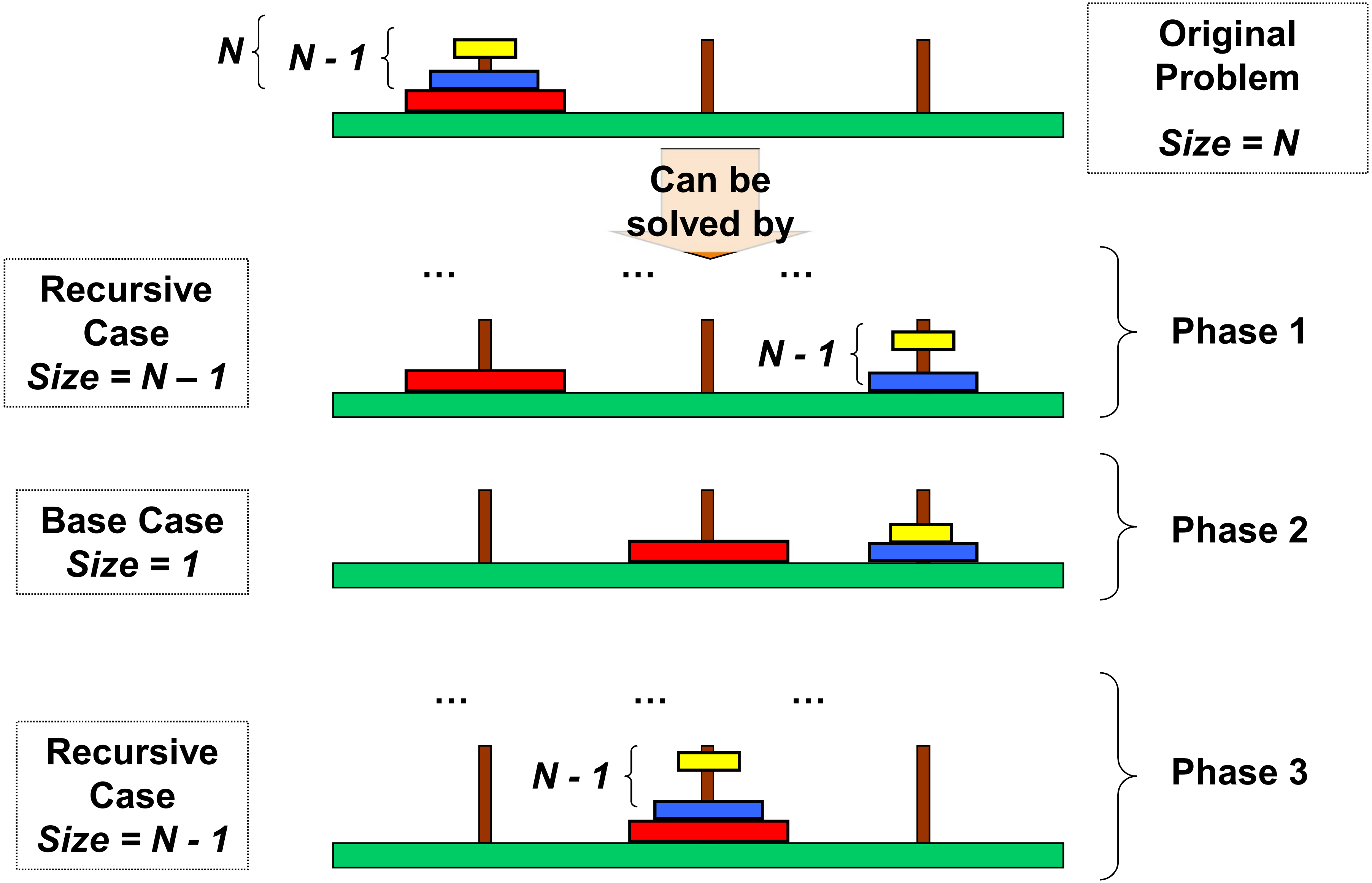


- How do we move all the disks from pole "**A**" to pole "**B**", using pole "**C**" as temporary storage
  - One disk at a time
  - Disk must rest on top of larger disk

Final state



Tower of Hanoi: **Recursive Solution**



Tower of Hanoi: **Solution**

```
def tower(N, Src, Dst, Tmp):
    if N == 1:
        move( Src, Dst)
    else:
        tower(N-1, Src, Tmp, Dst)
        move(Src, Dst)
        tower(N-1, Tmp, Dst, Src)


def move( From, To):
    print("Move from pole %s to pole %s" % (From, To))
```

Perform the "move". Many implementations. Below is one possibility.

- Fun Challenge:
  - Add this code to the Tower of Hanoi class from the stack lecture
  - You get an visualized solver for Tower of Hanoi!

Number of Moves Needed

Num of discs, n	Num of moves, f(n)	Time (1 sec per move)
1	1	1 sec
2	3	3 sec
3	3+1+3 = 7	7 sec
4	7+1+7 = 15	15 sec
5	15+1+15 = 31	31 sec
6	31+1+31 = 63	1 min
…	…	…
16	65,536	18 hours
32	4.295 billion	136 years
64	1.8 * 10^10 billion	584 billion years

Note the pattern

$f(n) = 2^n - 1$

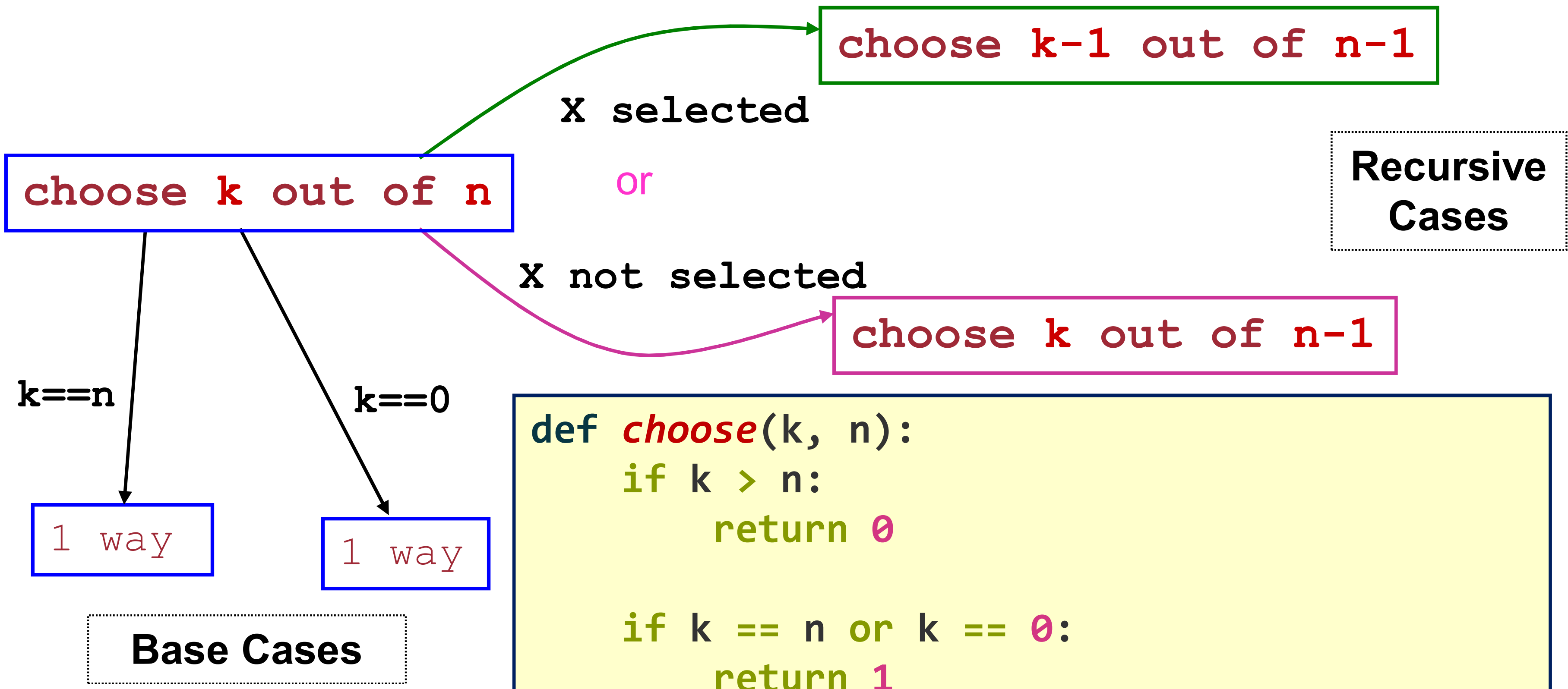
"Crazy" Questions

- For a 10-disk Tower of Hanoi problem, what is the 512nd move?

- What is the 256th move?

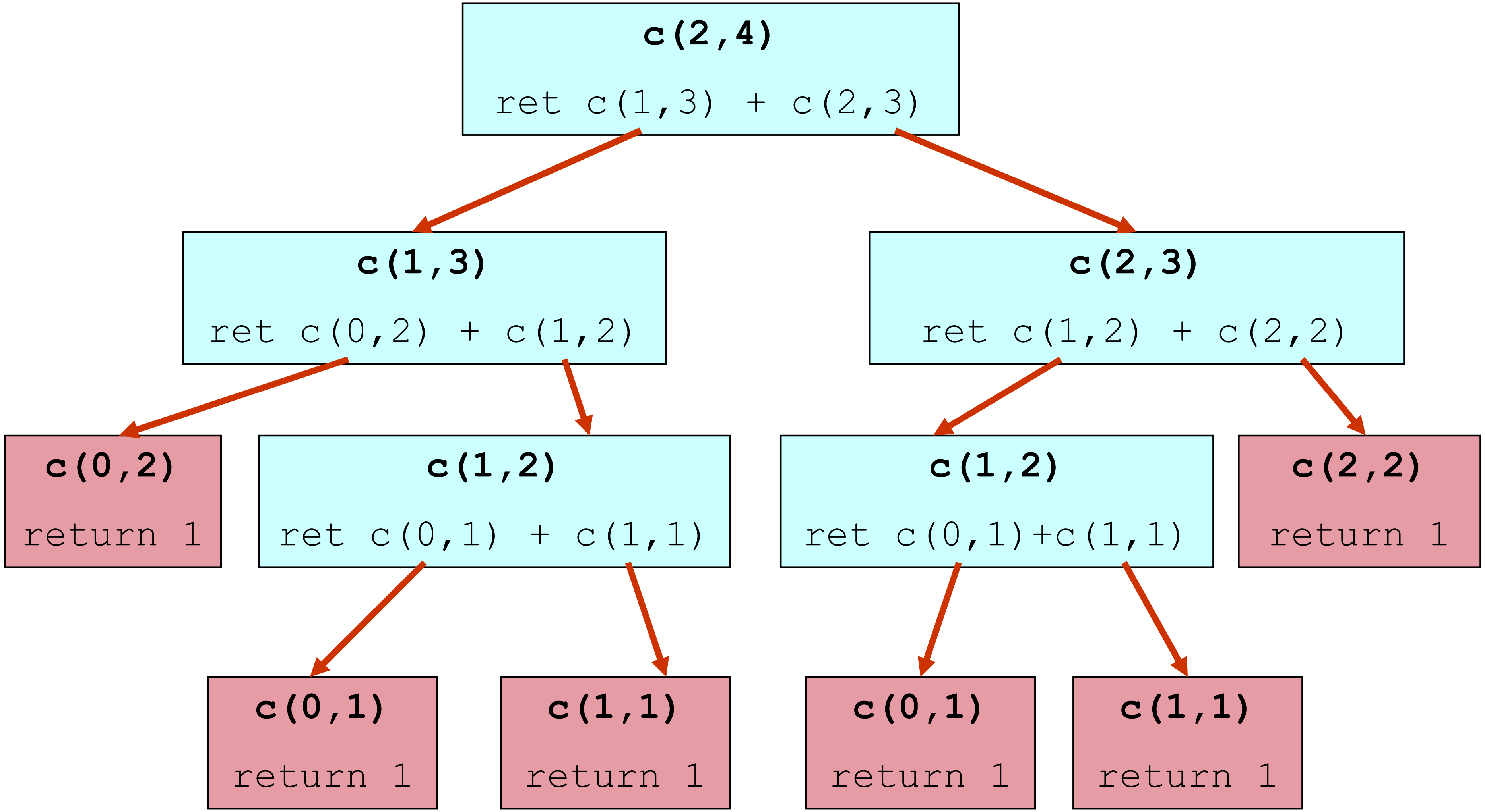
Example: **Combinatorial**

- How many **ways** can we choose ***k*** items out of ***n*** items?




```python
def choose(k, n):
    if k > n:
        return 0

    if k == n or k == 0:
        return 1

    return choose(k-1, n-1) + choose(k, n-1)
```

Execution Trace: *choose*(2, 4)

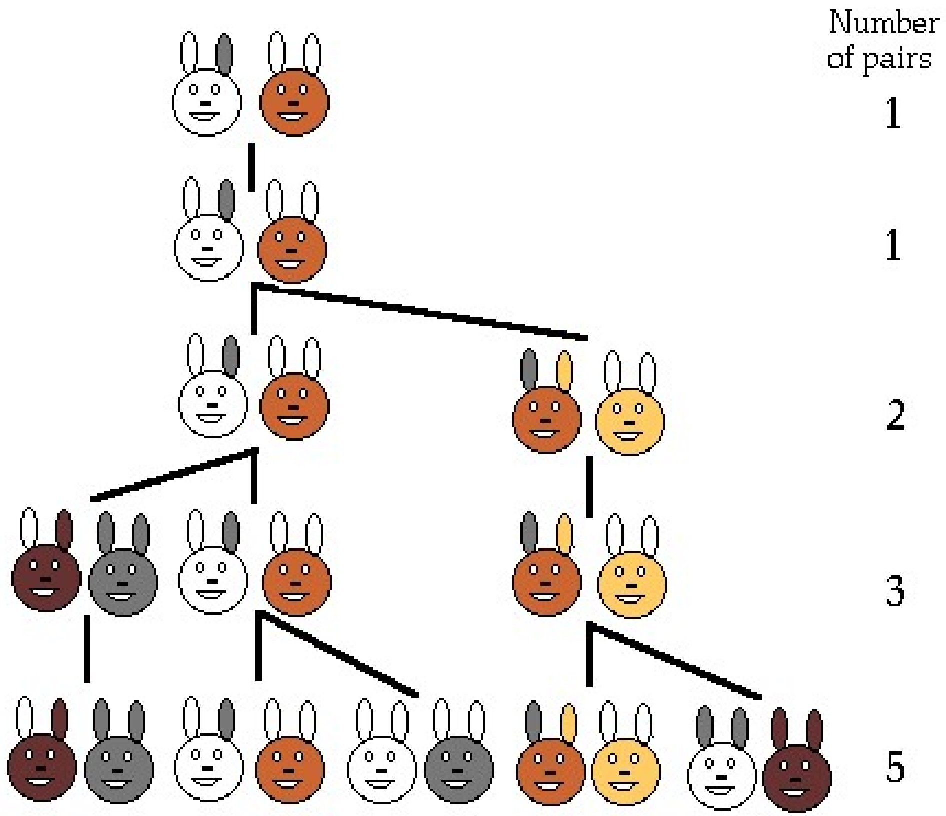


The final answer is the sum of the base cases

Example: Fibonacci Numbers

Rabbits give birth monthly once they are 3 months old and they always conceive a single male-female pair.

Given a pair of male-female rabbits, assuming rabbits never die, how many pairs of rabbits are there after *n* months?



The Fibonacci Series

- `Rabbit(N)`= # pairs of rabbit at Nth month
  - All rabbit pairs in the previous month (N – 1)th month stay
    - Rabbits never die
  - Additionally, new rabbit pairs = the total rabbit pairs two months ago (N – 2)th month
    - Rabbits give birth at the 3rd month

- Hence:
  - `Rabbit(N) = Rabbit(N – 1) + Rabbit(N – 2)`

- Special cases:
  - `Rabbit(1) = 1` One pair in the 1st month
  - `Rabbit(2) = 1` Still one pair in the 2nd month

- `Rabbit(N)` is the famous `Fibonacci(N)`

Fibonacci Number: Implementation

```
def fibonacci( n ):

    if n <= 2:
        return 1

    return fibonacci(n-1) + fibonacci(n-2)
```

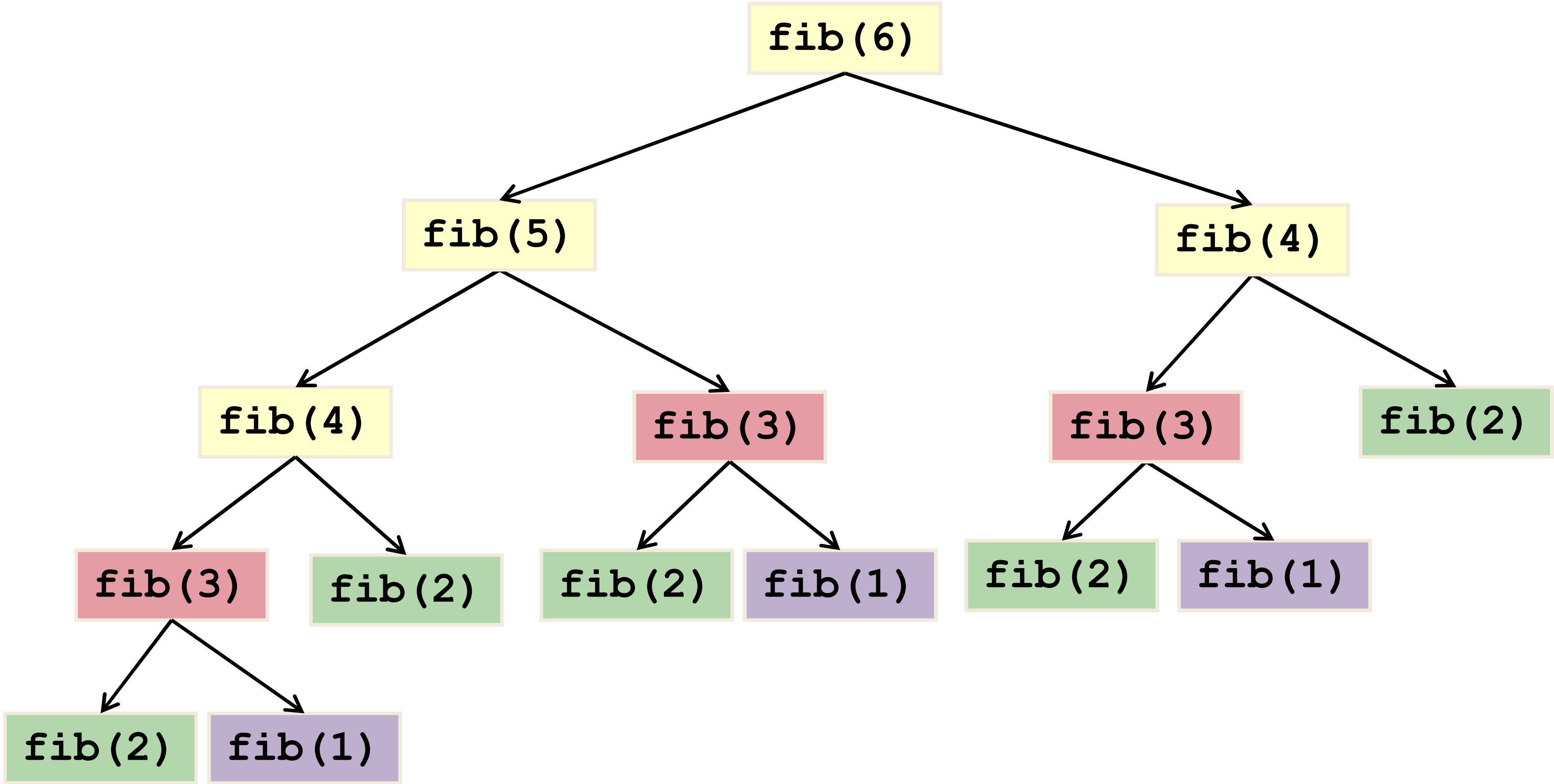
Base Cases:

fibo(1) = 1
fibo(2) = 1

Recursive Case:

fibo(n) = fibo(n – 1) + fibo(n – 2)

Execution Trace: Fibonacci



- Many duplicated calls:
  - The **same computations** are done over and over again!

Fibonacci Number: Iterative Solution

```
def fibonacciI( n ):
    if n <= 2:
        return 1

    prev1 = prev2 = 1
    for i in range(3, n+1):
        cur = prev1 + prev2
        prev2 = prev1
        prev1 = cur

    return cur
```

Iterative Version

- How many time do we calculate a particular *fibonacci* number?

Example: **Searching in Sorted Array**

Given a sorted array **`a`** of **`n`** elements and **`x`**, determine if **`x`** is in **`a`**.

`a` `=`

1	5	6	13	14	19	21	24	32

`x` `= 15`

- How do you reduce the number of checking?
- Idea: Narrow the search space by half at every iteration until a single element is reached

Binary Search

```
def binarySearch( array, target, low, high ):

    if low > high:
        return -1

    mid = (low + high) // 2

    if target > array[mid]:
        return binarySearch(array, target, mid+1, high)

    elif target < array[mid]:
        return binarySearch(array, target, low, mid-1)

    else:
        return mid
```

Exhausted array: Target not in array!

Find the middle element

Search upper half

Search lower half

Found!

Example: Find kth Smallest Number

- Locate **kth smallest** number in an **unsorted** array

```
def KthSmallest(array, k):

    Choose any element p from a[]
    Partition the array into 2 parts where
      L = elements that are <= p (so p is in L).
      R = elements that are larger than p.
    nL = number of elements in L.

    if k == nL:
       return p

    if k < nL:
        return ksmall(L, k)

    return ksmall(R, k - nL)
```

Find the K^{th} Smallest Number

- E.g. **Find the 7th smallest number**

32	5	28	4	24	19	21	1	2

5	4	1	2	**19**

5 items

32	28	24	21

Find (7-5)th item

- E.g. Find the 3rd smallest number

5	4	1	2	**19**

Find 3rd item

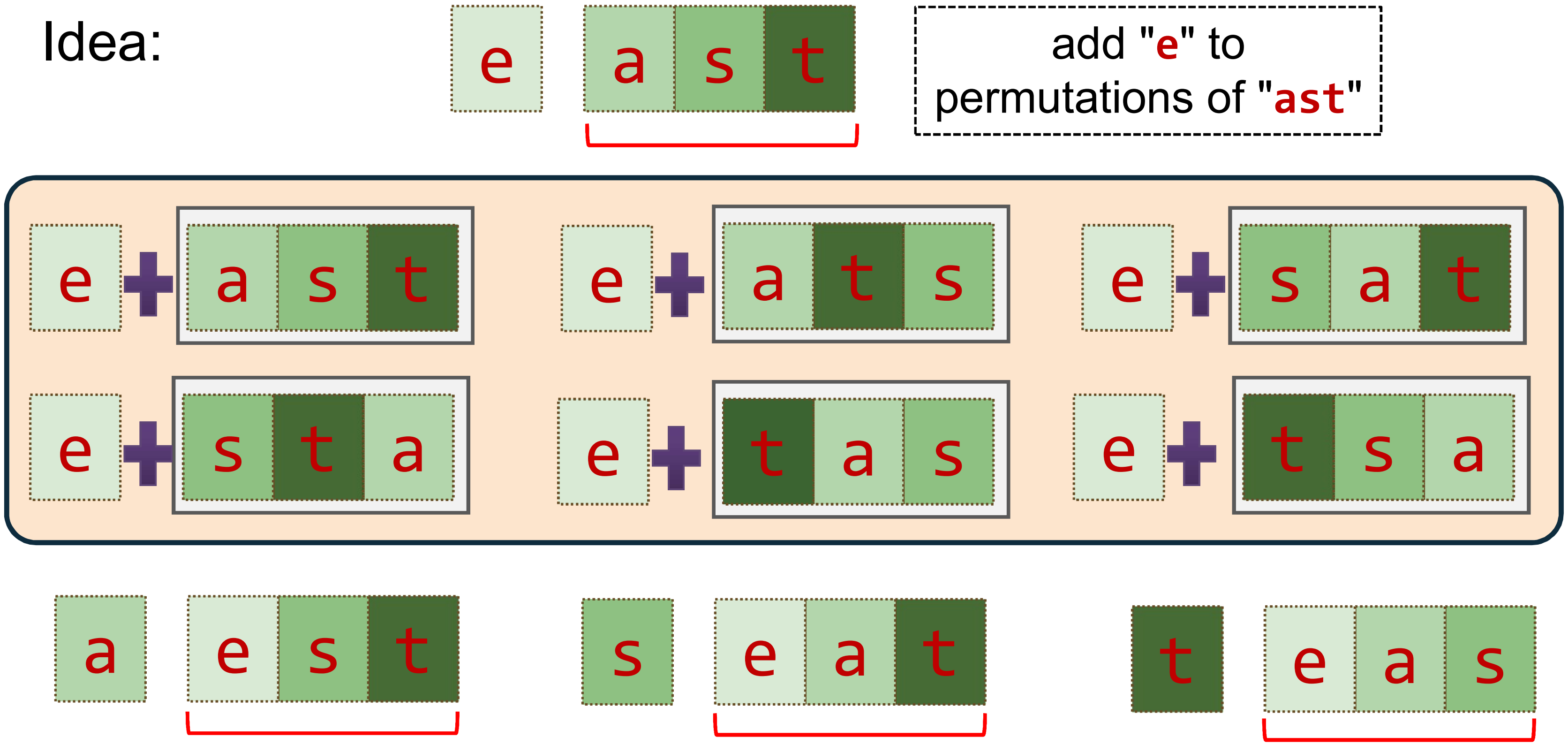
32	28	24	21

Example: Find kth Smallest Number

```
def KthSmallest( array, k ):
    p = array.pop(-1)   #use the last element
    left = []
    right = []

    for item in array:
        if item <= p:
            left.append(item)
        else:
            right.append(item)
    nLeft = len(left) + 1   # + 1 for the p itself

    if k == nLeft:
        return p

    if k < nLeft:
        return KthSmallest(left, k)

    return KthSmallest(right, k-nLeft)
```

Example: **Find all Permutations of a String**

- Generate all permutations of a given word
  - E.g. Given **`east`**, there are 24 permutations, including **`eats`**, **`etas`**, **`teas`**, and non-words like **`tsae`**
- Idea:



Similarly for {"a", "est"} {"s", "eat"} and {"t", "eas"}

Example: **Find all Permutations of a String**

```
def permutate( str ):
    if len(str) <= 1:
        return [str]

    result = []
    for idx in range(len(str)):
        letter = str[idx]
        leftover = str[:idx]+str[idx+1:]

        for substr in permutate(leftover):
            result.append(letter+substr)

    return result
```

***Base case:** empty string or 1 letter string*

Take out each character in turn. "leftover" is the remaining substring

For each permutation, add the letter at the front to form new permutation

- This implement returns all permutation in a Python list!

Summary

- Recursion is not just a way of programming, it is also a powerful approach to problem solving and formulating a solution
- A recursive function has base cases and recursive cases
- Relationship between recursion and stack
- Watch out for duplicate computations!

END