Recursion

{ik} INTERVIEW KICKSTART

Real-life counterparts of programming features

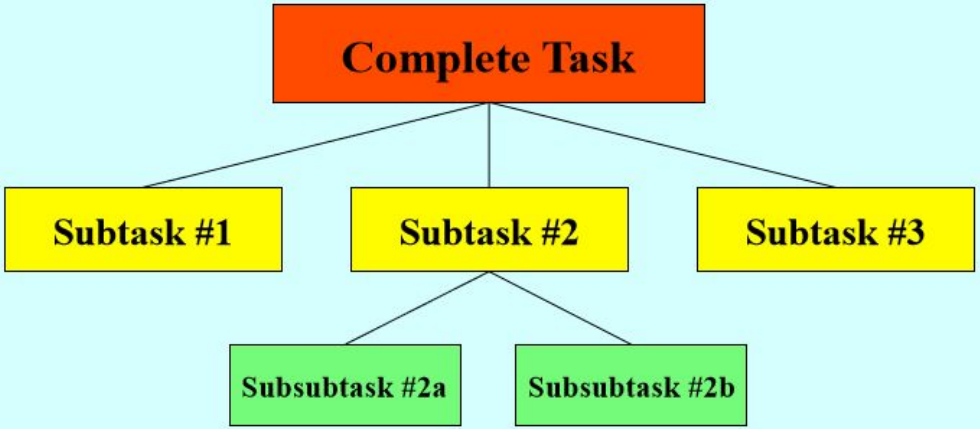
for, while loops = perform a task repeatedly (iteration)

if-then-else conditional test = make a decision based on some condition

Top-down design or Stepwise refinement via functions = Subdivide an overall high-level problem into a set of lower-level steps

Stepwise Refinement

- The most effective way to solve a complex problem is to break it down into successively simpler subproblems.
- You start by breaking the whole task down into simpler parts.
- Some of those tasks may themselves need subdivision.
- This process is called ***stepwise refinement*** or ***decomposition***.



Recursion = Solve large problems by reducing them to smaller problems **of the same form.**

Reduce a large problem to one or more subproblems that are
1) Identical in structure to the original problem
2) Somewhat simpler to solve

Then use the same decomposition technique to divide the subproblems into new ones that are even simpler... until they become so simple that you can solve them without further subdivision.
Reassemble the solved components to obtain the complete solution to the original problem.

“When students first encounter recursion, they often react with suspicion to the entire idea, as if they have just been exposed to some conjurer’s trick, rather than a critically important programming methodology. That suspicion arises because recursion has few analogues in everyday life and requires students to think in an unfamiliar way.” (Eric Roberts, *Thinking Recursively*)

A crude real-life counterpart to recursion

You have been appointed as the growth manager for a non-profit company. Your job is to raise 100000$.

This task greatly exceeds your own capacity. So you need to delegate the work to others.

You find 10 volunteers, and give them each the task of raising 10000$ each.

Each of them finds 10 volunteers, each of who is tasked with raising 1000$. They in turn could find volunteers who only need to raise only 100$.

Pseudocode for strategy

```
function raiseMoney (int n):
    if n <= 100:
        collect the money from a single donor
    else:
        find 10 volunteers
        get each of them to collect (n/10) dollars
        combine the money raised by the volunteers
```

Pseudocode for strategy

```
function raiseMoney (int n):
    if n <= 100:
        collect the money from a single donor
    else:
        find 10 volunteers
        get each of them to collect (n/10) dollars
        combine the money raised by the volunteers
```

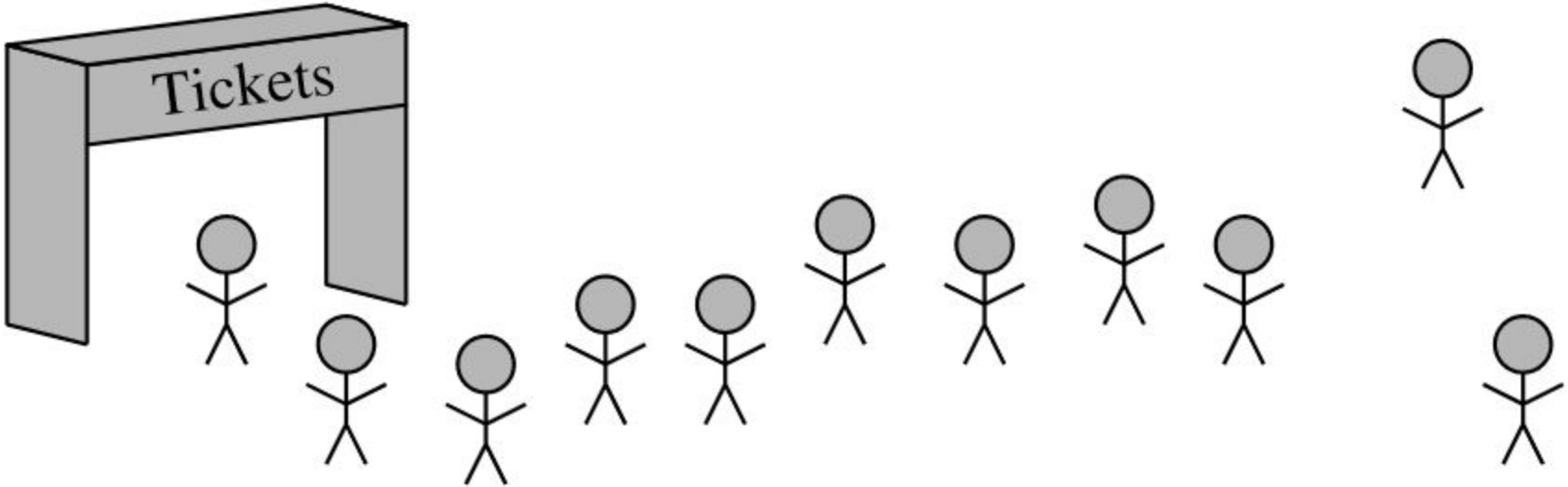
This has the same form as the original problem, and the problem size is smaller.

Pseudocode for strategy

```
function raiseMoney (int n):
    if n <= 100:
        collect the money from a single donor
    else:
        find 10 volunteers
        for each volunteer: call raiseMoney(n/10)
        combine the money raised by the volunteers
```

Pseudocode for general recursive solution

```
if (test for a simple case):
    compute a simple solution without using recursion
else:
    #Divide-and-conquer or Decrease-and-conquer
    break the problem into subproblems of the same form
    solve each of the subproblems by calling this function recursively
    reassemble the subproblem solutions into a solution for the whole
```



Types of recursive problems

1. Recursive mathematical functions
2. Recursive procedures
3. Recursive backtracking
4. Recursive data

1. Recursive mathematical functions

(Start with simple mathematical functions in which the recursive structure follows directly from the statement of the problem, and is therefore easy to see)

The factorial function

fact(n) = n * (n-1) * (n-2) * ... * 2 * 1

Rule of sum: If an action can be performed by choosing one of A different options, **OR** one of B different options, then it can be performed in A + B ways.
e.g, 4 short-sleeved shirts + 6 long-sleeved shirts in the closet; can choose a shirt in 4 + 6 = 10 ways

Rule of product: If an action can be performed by choosing one of A different options **followed by** one of B different options, then it can be performed in A x B ways.
e.g, 10 shirts and 8 pants in the closet; can choose a shirt and pant in 10 x 8 = 80 ways

The factorial function

Arrange the four different letters a, b, c, d in a straight line. In how many ways can this be done?

Number of ways to arrange n different objects in a straight line = ?

“Permutations”

(So far, repetition not allowed)

Repetition allowed: Number of binary strings of length n = ? Number of 4 digit passcodes = ?

“Arrangements”

Calculate the value of n! using Decrease-and-conquer

Conventional mathematical definition:

n! = 1 if n = 0

= n x (n-1)! Otherwise

The mathematical definition itself is recursive, so it provides a template for an easy recursive implementation.

```
def fact(n):
    if n == 0:
        return 1
    else:
        #Chip away at the problem by reducing it to size n-1
        #Ask a worker clone to solve that smaller problem
        #Construct the solution to the overall problem using that
        return n * fact(n-1)
```

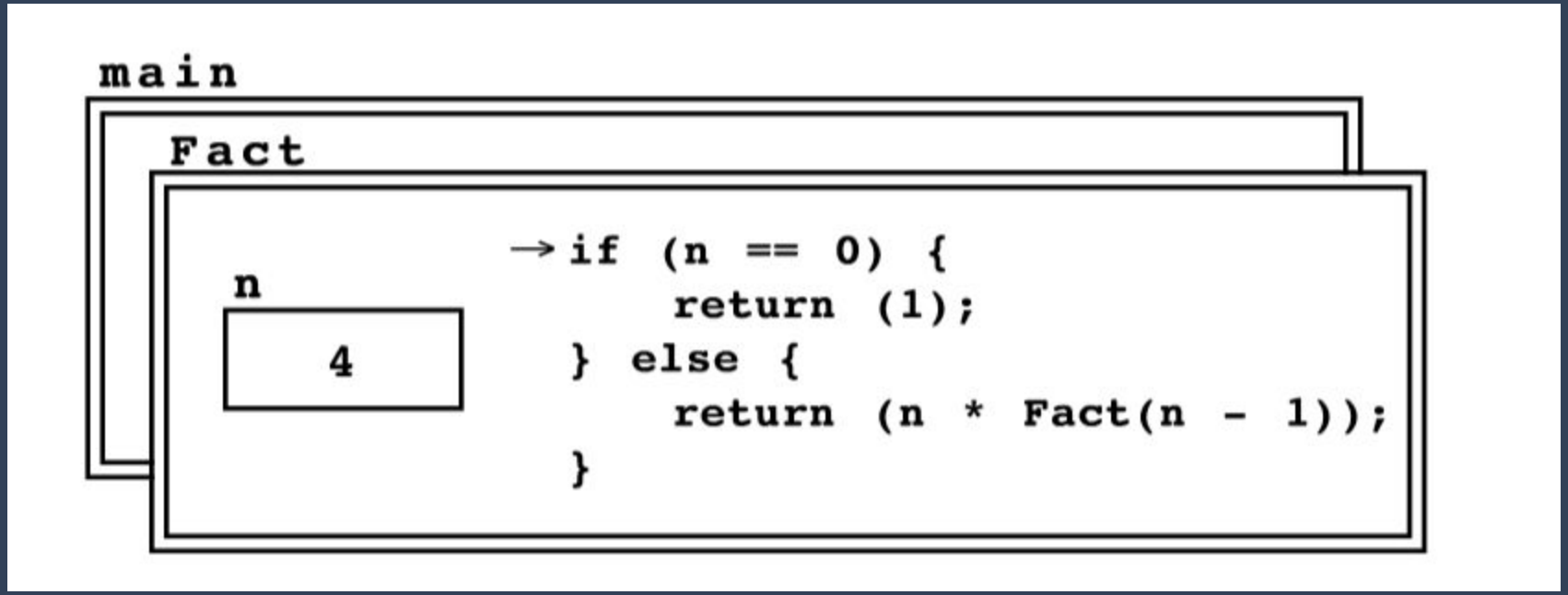
Iterative strategy

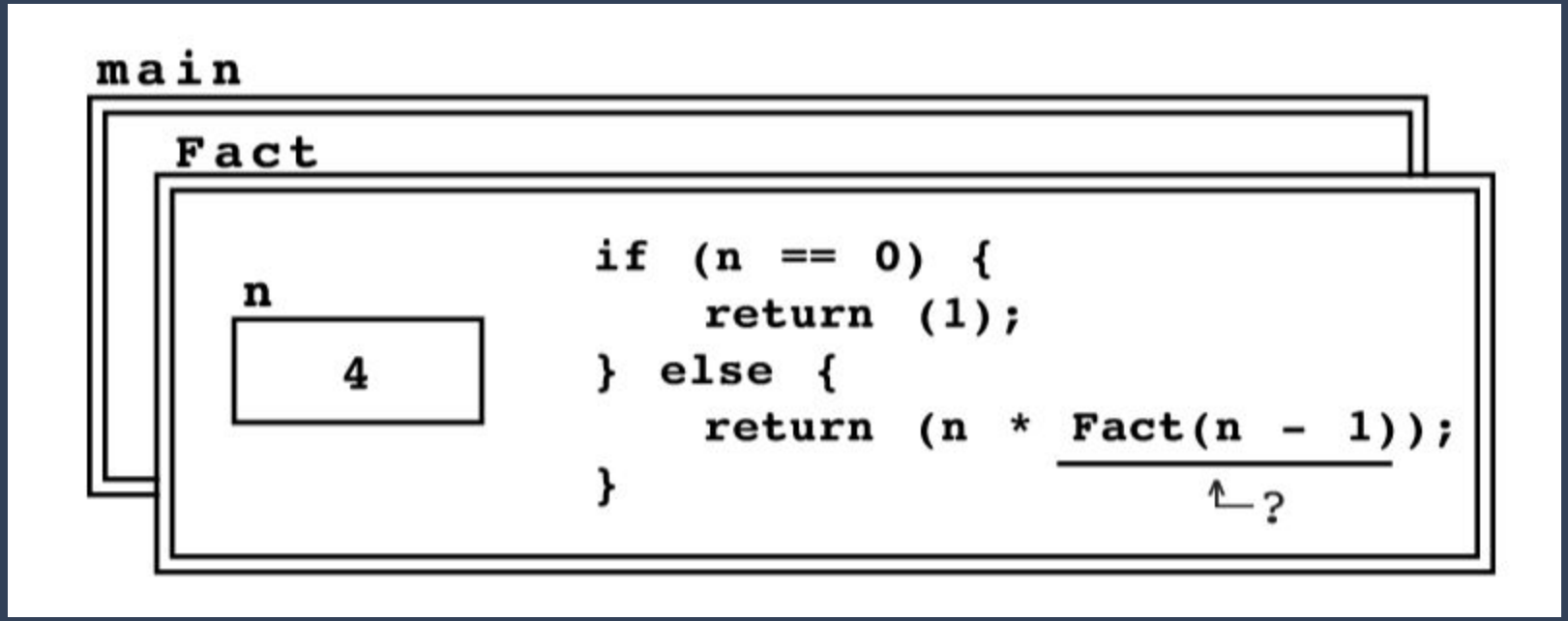
Recall our previous discussion of insertion sort: It was a decrease-and-conquer sorting algorithm that decreased a problem of size n into a subproblem of size n-1. We implemented it using a top-down recursive implementation, as well as a bottom-up iterative implementation. Can do the same here:

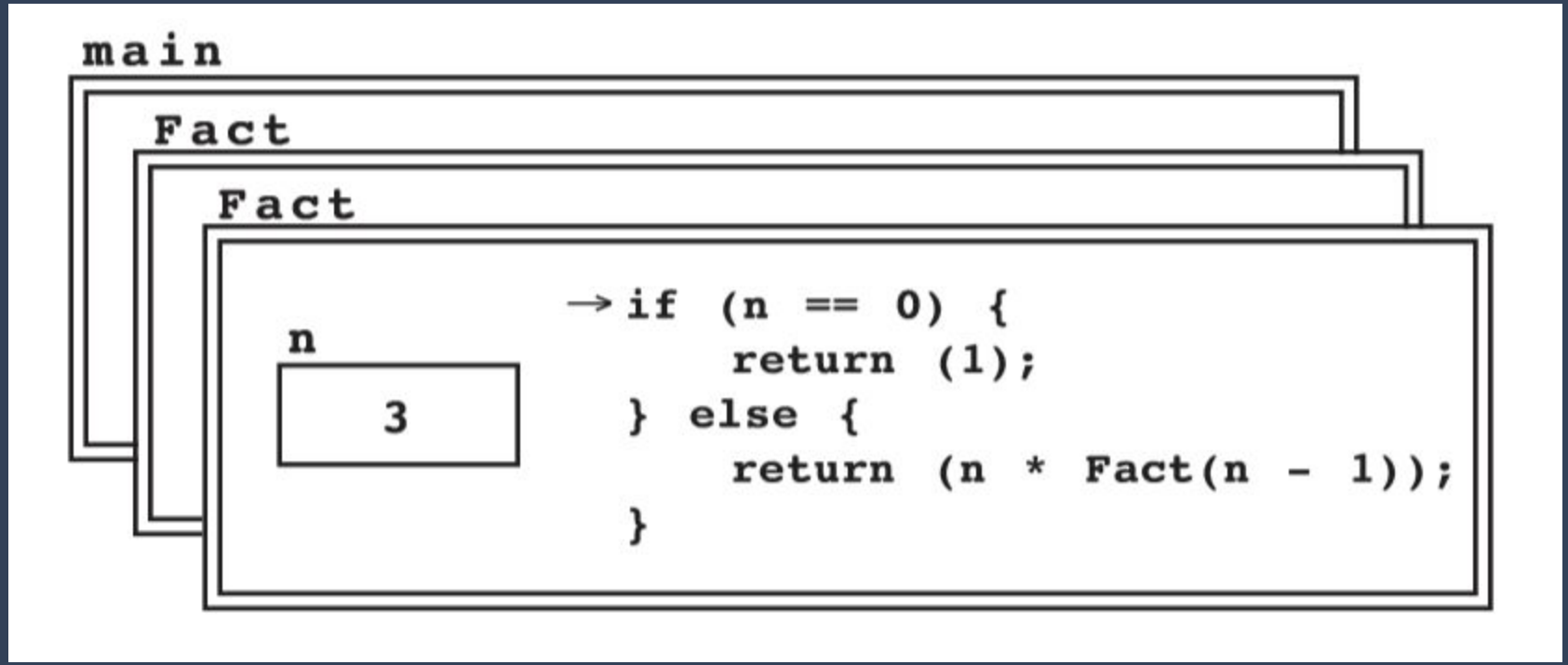
```
def fact(n):
    result = 1
    for i in 1 to n:
        result = result * i
    return result
```

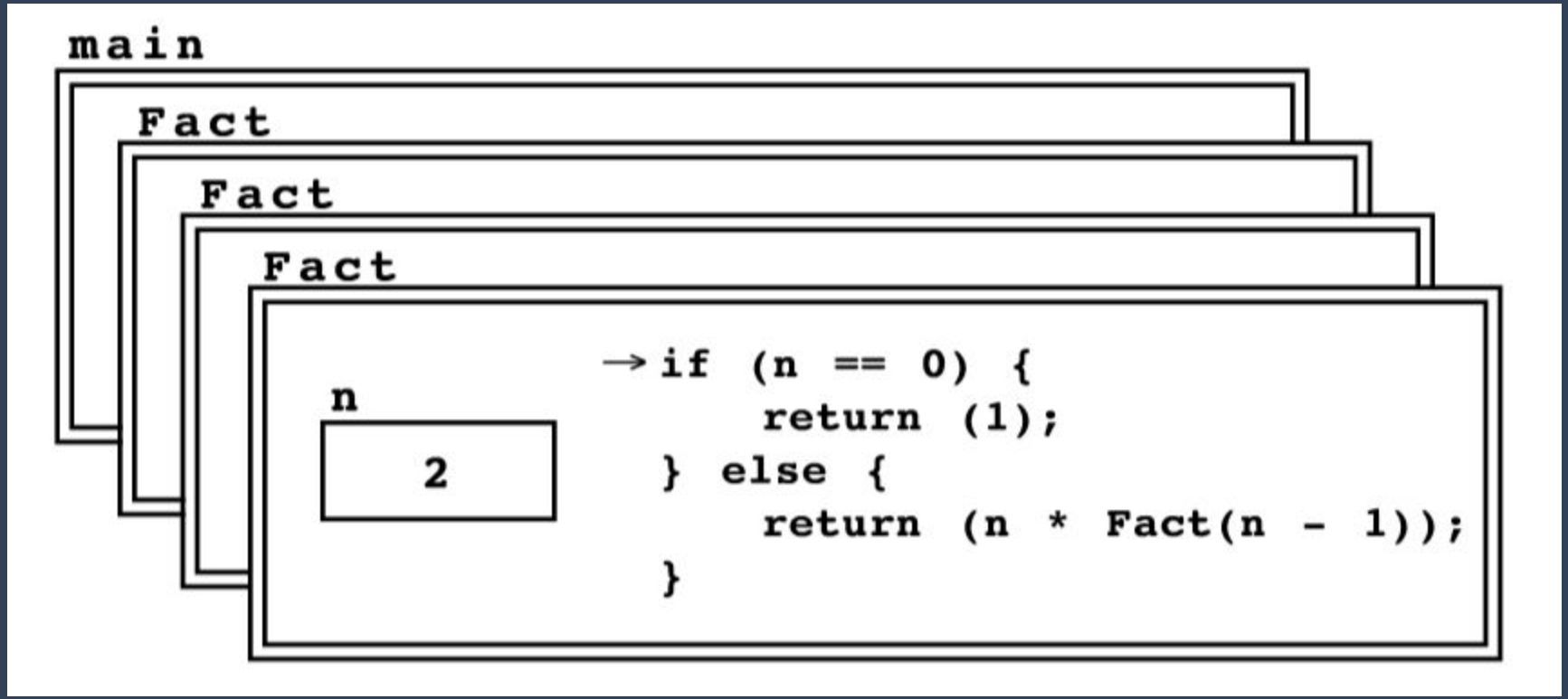
Time complexity = ?

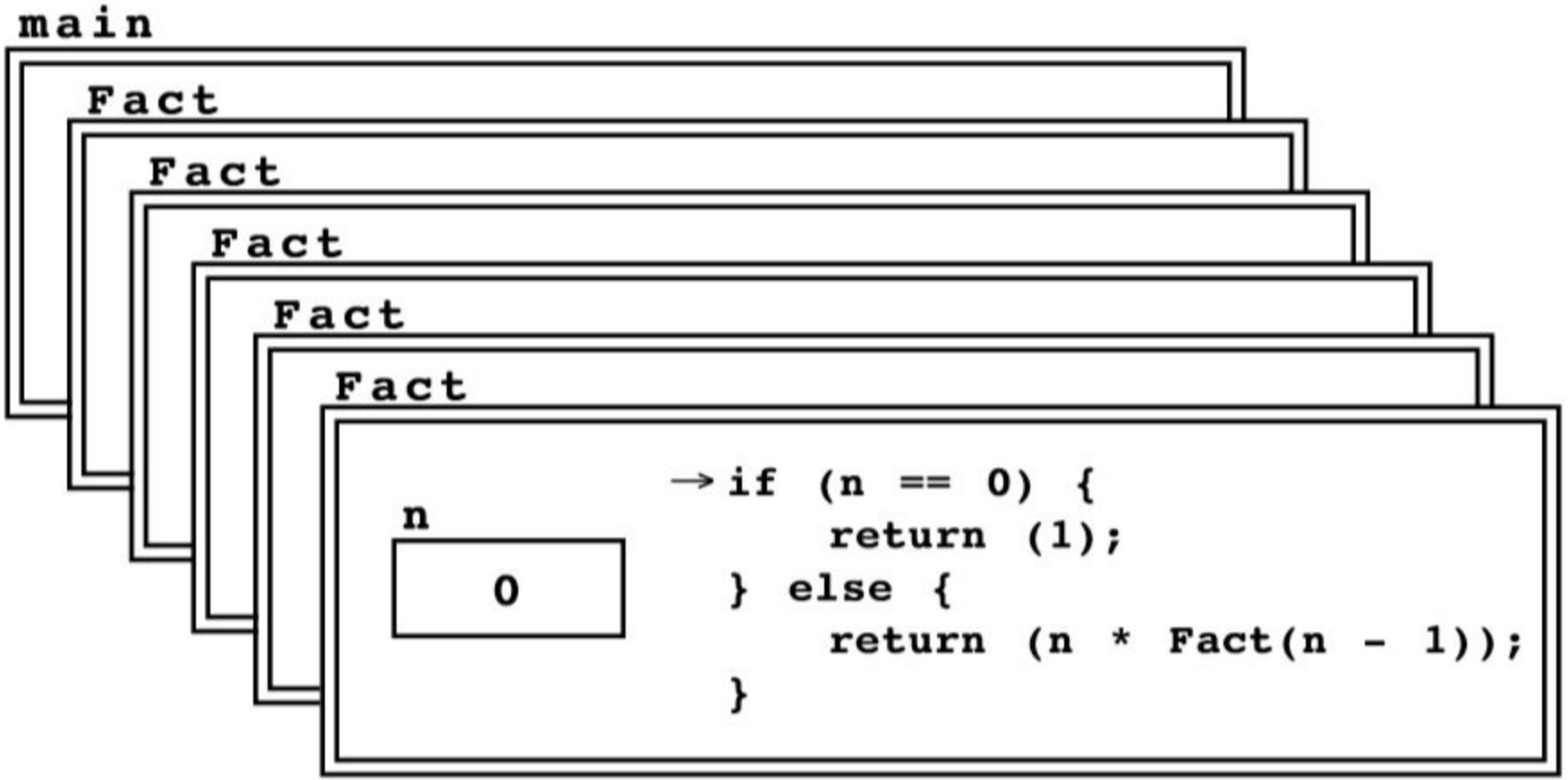
(In the main function:)
f = fact(4)

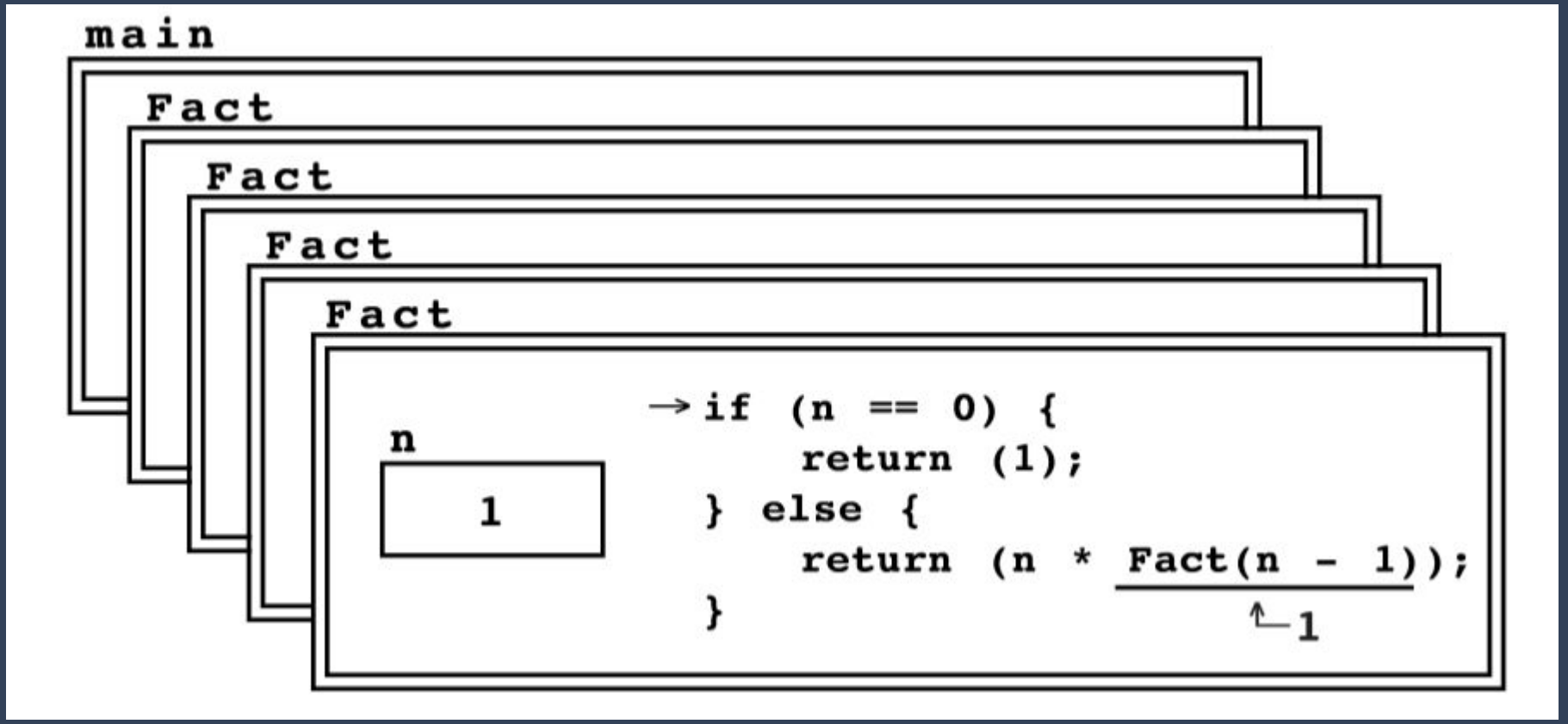


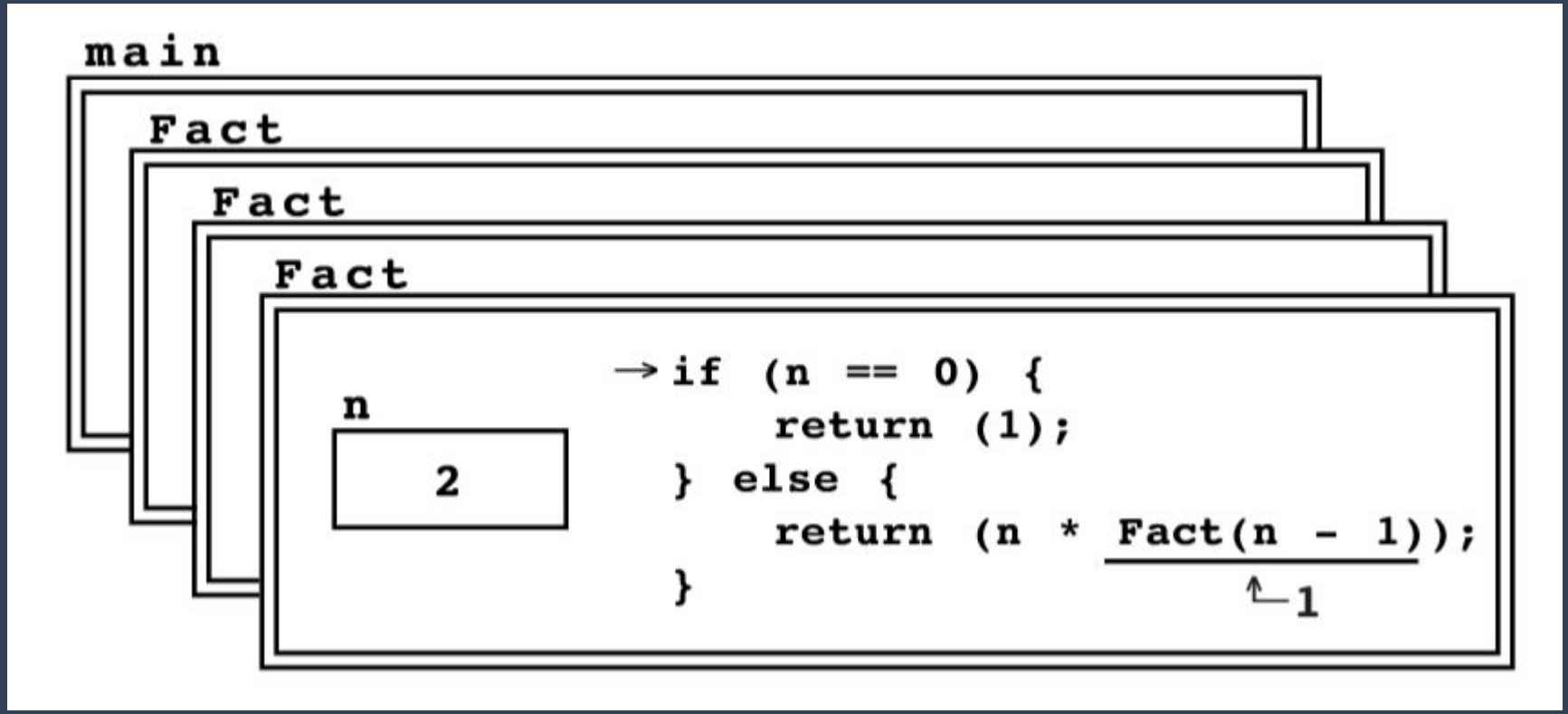


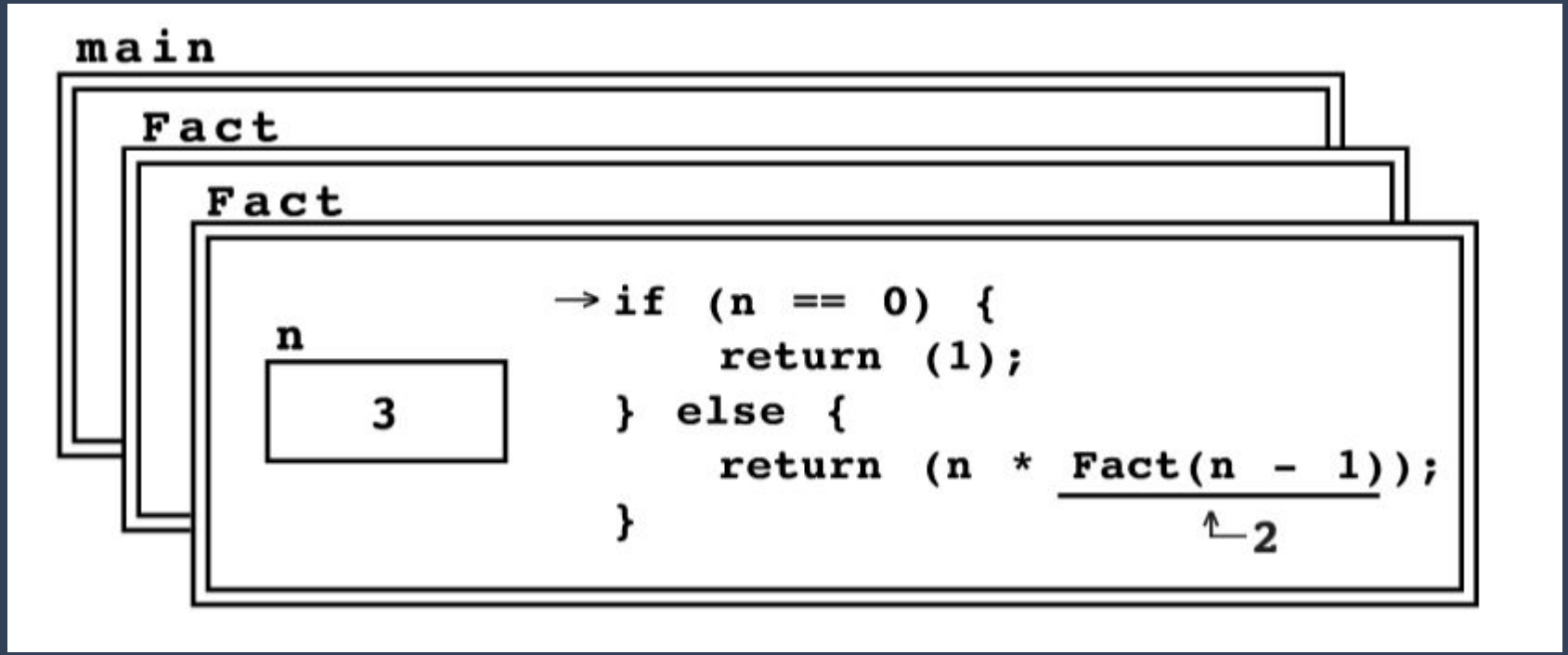


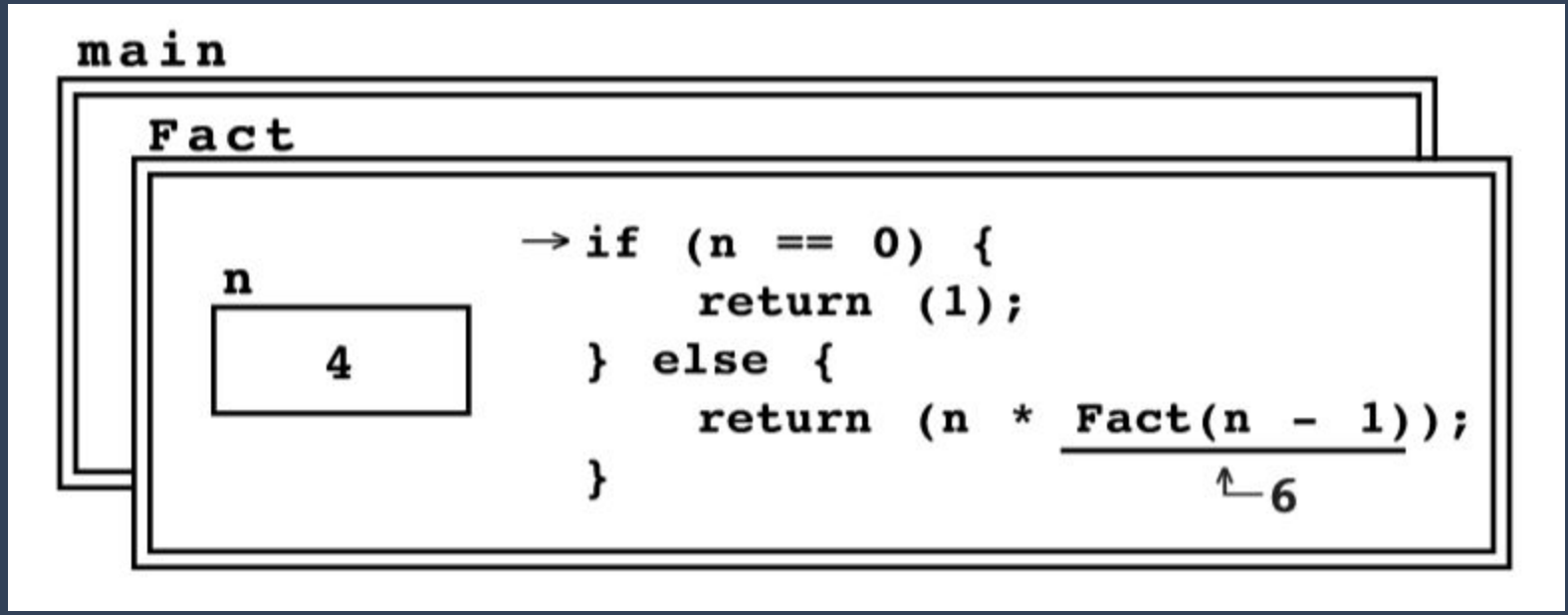












f = fact(4) ←- value of 24 returned to the main program

Recursive leap of faith

In general, you don’t want to trace the chain of execution for every problem. Focus instead only on a single level of recursion.

Assume that any recursive call automatically gets the right answer as long as the arguments are simpler than the original arguments.

(If you really want a formal proof, the combination of base case and analysis of the move from n-1 ---> n can both be used to build up a *proof by mathematical induction*)

Write a recursive implementation of a function that raises a number to a power.

```
int RaiseIntToPower(int n, int k)
```

Use this mathematical definition:

$$n^k = \begin{cases} 1 & \text{if } k = 0 \\ n \times n^{k-1} & \text{otherwise} \end{cases}$$

```
def RaiseIntToPower(n, k):
    if k == 0:
        return 1
    else:
        return n * RaiseIntToPower(n, k-1)
```

Again, a decrease-and-conquer strategy like this (problem of size n constructed from solution to subproblem of size n-1) means it can be easily implemented iteratively as well.

Time complexity = ?

How many subsets of a set of size n are there?

How many subsets of a set of size n are there?

Hint: What if I knew how many subsets of size n-1 are there?

How many subsets of a set of size n are there?

Hint: What if I knew how many subsets of size n-1 are there?

S(n) = Number of subsets of size n

S(n) = S(n-1) + S(n-1) = 2S(n-1)

How many subsets of a set of size n are there?

Hint: What if I knew how many subsets of size n-1 are there?

S(n) = Number of subsets of size n

S(n) = S(n-1) + S(n-1) = 2S(n-1)

Base case: S(0) = 1

How many subsets of a set of size n are there?

Hint: What if I knew how many subsets of size n-1 are there?

S(n) = Number of subsets of size n

S(n) = S(n-1) + S(n-1) = 2S(n-1)

Base case: S(0) = 1

```
def subsets(n):
    if n == 0:
        return 1
    else:
        return 2*subsets(n-1)
```

Time complexity = ?

What if we wrote the code like this?

```
def subsets(n):
    if n == 0:
        return 1
    else:
        return subsets(n-1) + subsets(n-1)
```

Time complexity = ?

Leonardo of Pisa (1170-1250)

Pisa

The leaning tower of Pisa

Location of Pisa in Italy

Leonardo of Pisa (1170-1250)

filius Bonaccio, “son of Bonaccio”: Fibonacci

- Travelled with his father to Algeria as a young boy.
- Learnt about the Hindu-Arabic numeral system there

- Fibonacci recognized the advantages of using the Hindu-Arabic numeral system (with a positional notation and zero symbol) over the clumsy Roman system still used in Italy.
- **He returned to Pisa in 1202 and wrote a book explaining the virtues of this number system “in order that the Latin race may no longer be deficient in that knowledge.”**

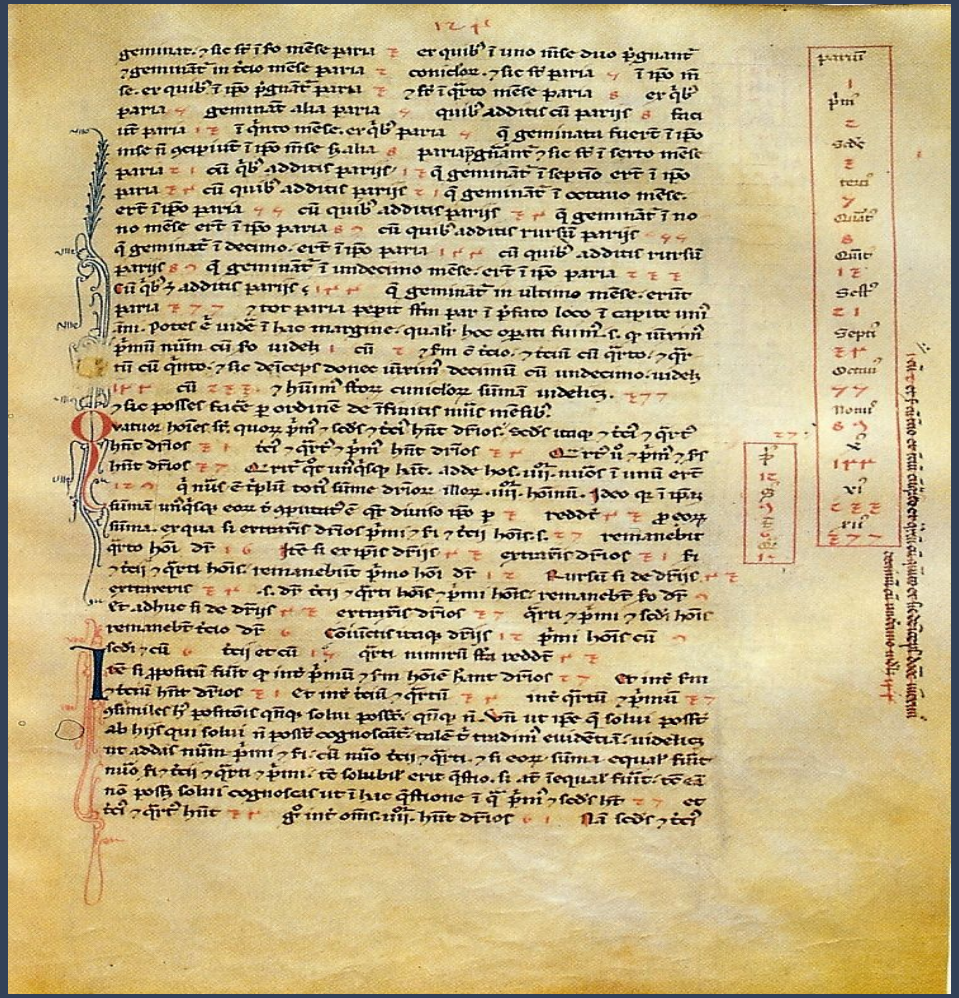
Liber abaci (Book of Counting)

Chapter 1:

“These are the nine figures of the Indians:

9 8 7 6 5 4 3 2 1.

With these nine figures, and with this sign 0... any number may be written, as will be demonstrated below.”



"It is ironic that Leonardo, who made valuable contributions to mathematics, is remembered today mainly because a 19th-century French number theorist, Édouard Lucas... attached the name Fibonacci to a number sequence that appears in a trivial problem in Liber abaci"

(Martin Gardner, Mathematical Circus)

Liber abaci, chapter 12

A man put one pair of (newborn) rabbits in a certain place entirely surrounded by a wall. How many pairs of rabbits can be produced from that pair in a year, if the nature of these rabbits is such that every month, each pair bears a new pair which from the second month on becomes productive?

(Assume no rabbits die)

New month’s adults = All the rabbit pairs from previous month

New month’s young = Previous month’s adults = Previous to previous month’s rabbit pairs

Fibonacci sequence

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

Need to specify two starting terms (base cases):

fib(1) = 1

fib(0) = 0

Recursive implementation of Fibonacci function fib(n)

Recursive implementation of Fibonacci function fib(n)

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Time complexity = ?

Golden Ratio

Divide a line segment into two unequal parts so that the whole segment will have the same ratio to its larger part that its larger part has to its smaller part.

A child is trying to climb a staircase. The maximum number of steps he can climb at a time is two; that is, he can climb either one step or two steps at a time. If there are n steps in total, in how many different ways can he climb the staircase?

{ik} INTERVIEW KICKSTART

C(n,k) - “Combinations”

Number of ways to choose k objects out of n, where repetition is not allowed *and order is also not important*.

C(n,n) = ?
C(n,0) = ?

C(n,k) - “Combinations”

Number of ways to choose k objects out of n, where repetition is not allowed *and order is also not important*.

C(n,n) = ?
C(n,0) = ?

C(n,k) = C(n,n-k)

C(n,k) - “Combinations”

Number of ways to choose k objects out of n, where repetition is not allowed *and order is also not important*.

C(n,n) = ?
C(n,0) = ?

C(n,k) = C(n,n-k)

C(n,k) = n!/k!(n-k)!

C(n,k) - “Combinations”

Number of ways to choose k objects out of n, where repetition is not allowed *and order is also not important*.

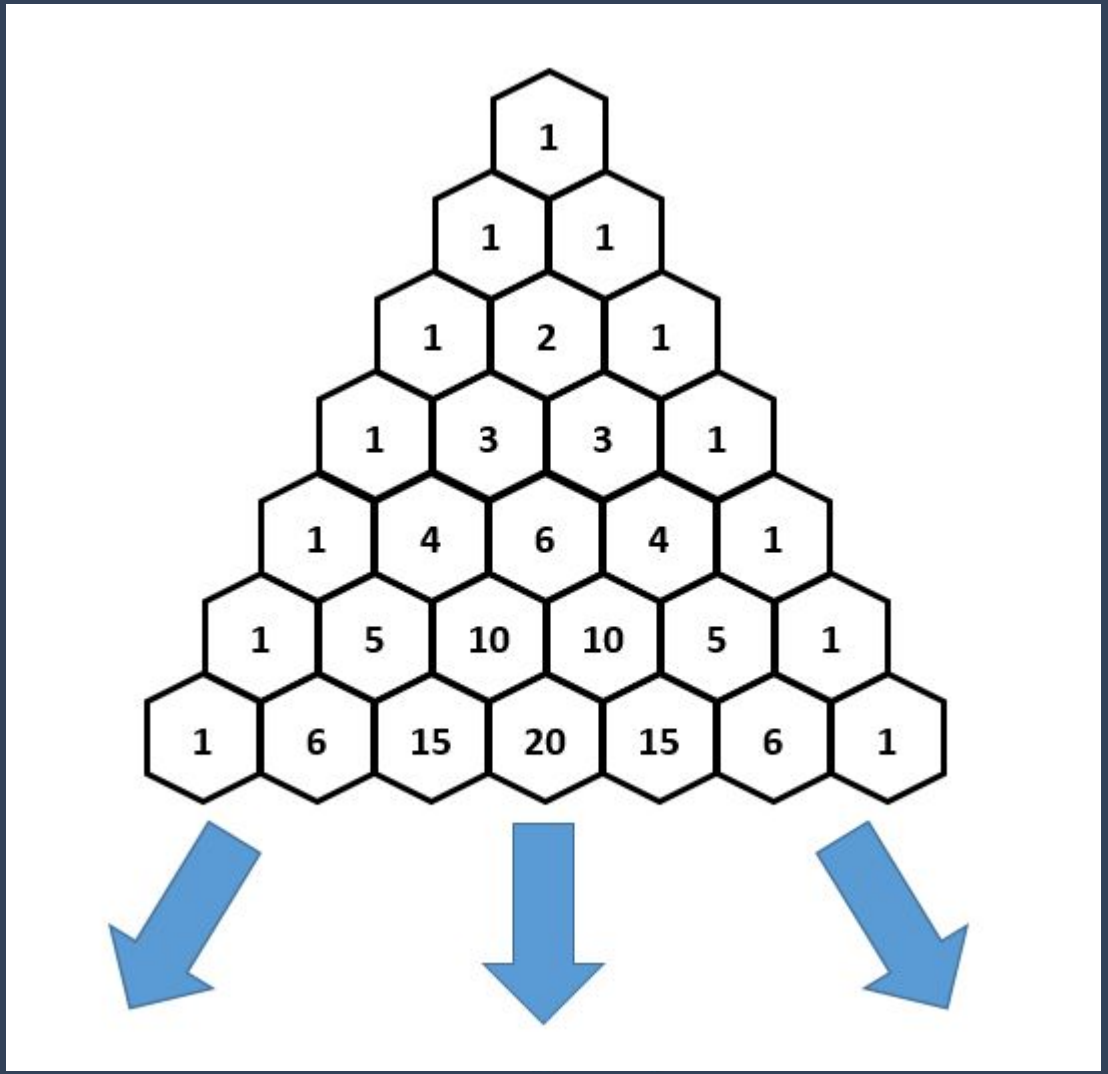
C(n,n) = ?
C(n,0) = ?

C(n,k) = C(n,n-k)

C(n,k) = n!/k!(n-k)!

C(n,k) = C(n-1,k) + C(n-1,k-1)

Pascal’s triangle



Write a recursive implementation of the C(n,k) function that uses no loops, no multiplication and no calls to fact.

Write a recursive implementation of the C(n,k) function that uses no loops, no multiplication and no calls to fact.

```
def C(n,k):
    if n <= 1 or k == 0 or k == n:
        return 1
    else:
        return C(n-1,k) + C(n-1,k-1)
```

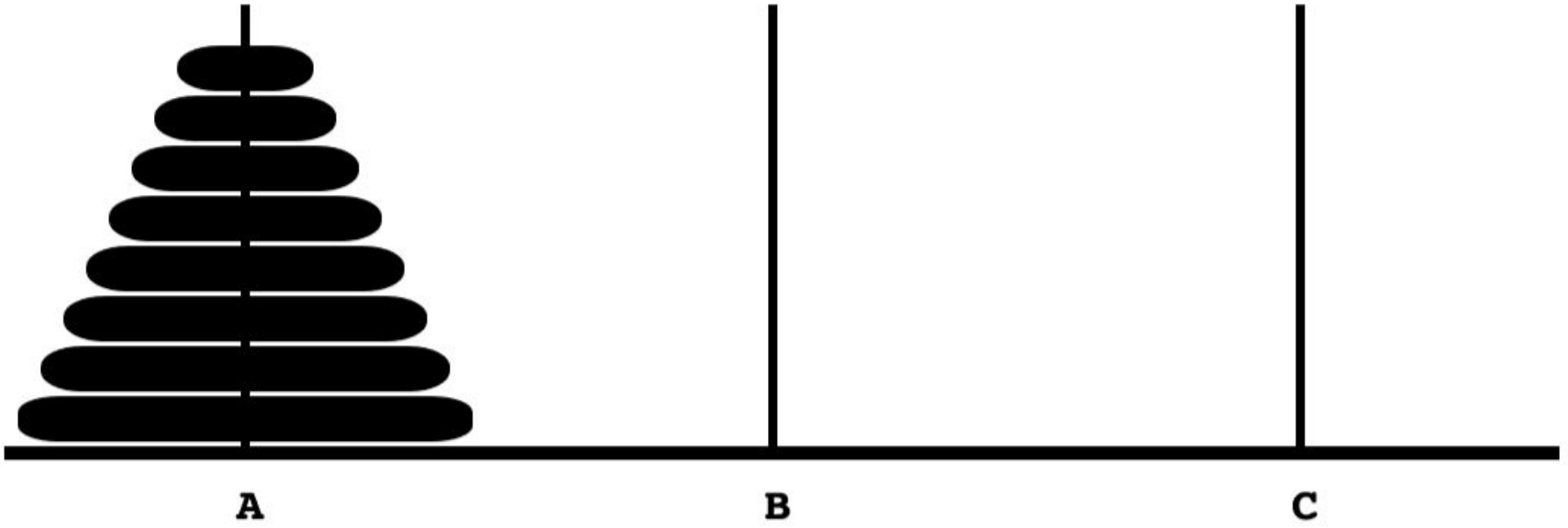
What is the sum of all the numbers in any row of Pascal’s triangle?

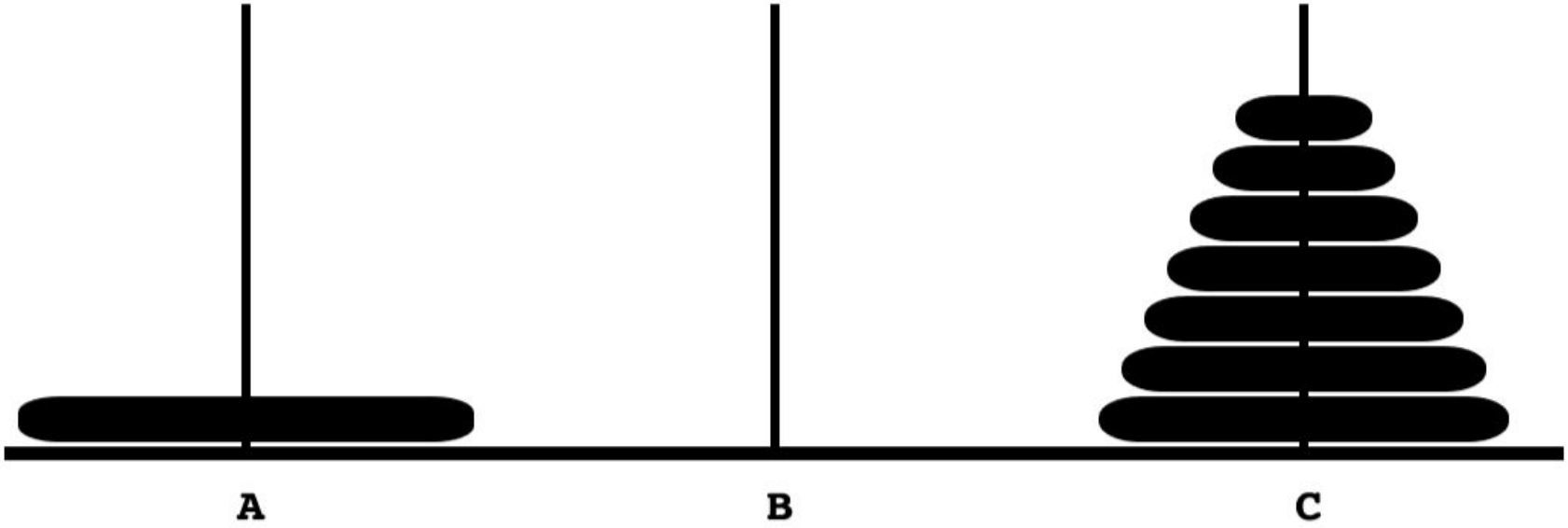
2. Recursive procedures

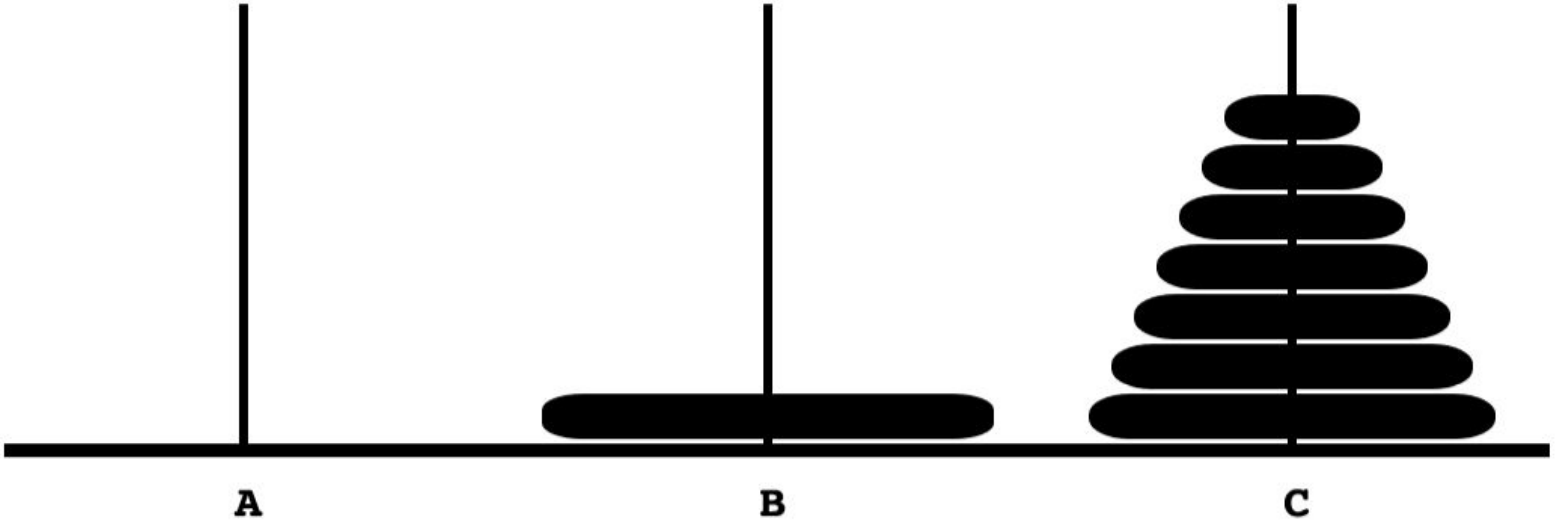
(While functions are mathematical entities, procedures are more algorithmic in character)

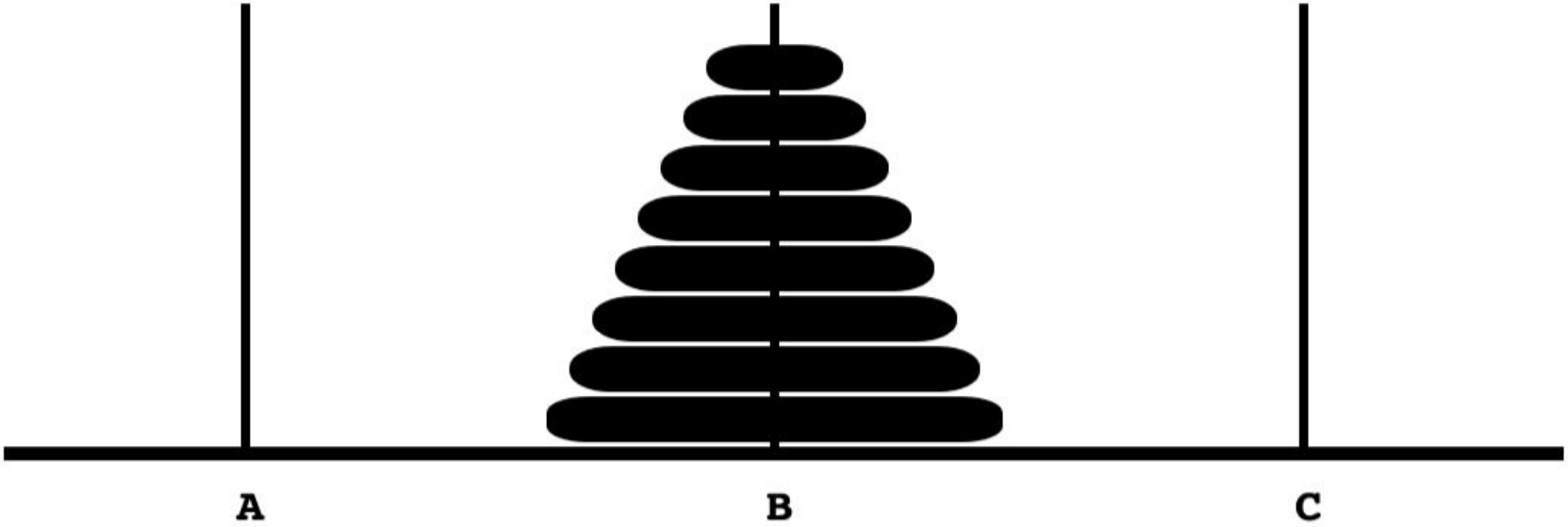
Tower of Hanoi. Invented by French mathematician Edouard Lucas in the 1880s, the Tower of Hanoi puzzle quickly became popular in Europe. Its success was due in part to the legend that grew up around the puzzle, which was described as follows in *La Nature* by the French mathematician Henri De Parville (as translated by the mathematical historian W. W. R. Ball):

> In the great temple at Benares beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four disks of pure gold, the largest disk resting on the brass plate and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and night unceasingly, the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disk at a time and that he must place this disk on a needle so that there is no smaller disk below it. When all the sixty-four disks shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.









```
def towerofhanoi(disks,src,dst,temp):
  if disks == 1:
    print "Move disk from " + src + " to " + dst
  else:
    towerofhanoi(disks-1,src,temp,dst)
    print "Move disk from " + src + " to " + dst
    towerofhanoi(disks-1,temp,dst,src)

towerofhanoi(5,"A","B","C")
```

Print all binary strings of length 5

```
def binarystrings5():
  for i in range(2):
    for j in range(2):
      for k in range(2):
        for x in range(2):
          for y in range(2):
            print str(i) + str(j) + str(k) + str(x) + str(y)

binarystrings5()
```

This won't work for a binary string of length n as we cannot have a variable number of for loops.

Print all binary strings of length n (decrease-and-conquer from right end)

```
def binarystringsi(n):
  if n == 1:
    return [str(0),str(1)]
  else:
    prev = binarystringsi(n-1)
    result = []
    for s in prev:
      result.append(s + "0")
      result.append(s + "1")
    return result

print binarystringsi(5)
```

This can be made into an iterative solution. The problem is that the space complexity is exponential. We need to have all subproblems of size n-1 stored.

Print all binary strings of length n: left-to-right, divide and conquer

```
def binarystrings(n):
  bshelper(n,"")

def bshelper(n,slate):
  if n == 0:
    print slate
  else:
     bshelper(n-1,slate + "0")
     bshelper(n-1, slate + "1")

binarystrings(5)
```

Print all decimal strings of length n

```
def decimalstrings(n):
  dshelper(n,"")

def dshelper(n, slate):
  if n == 0:
    print slate
  else:
    for i in range(10):
      dshelper(n-1,slate + str(i))

decimalstrings(3)
```