Recursion

{ik} INTERVIEW KICKSTART

Real-life counterparts of programming features

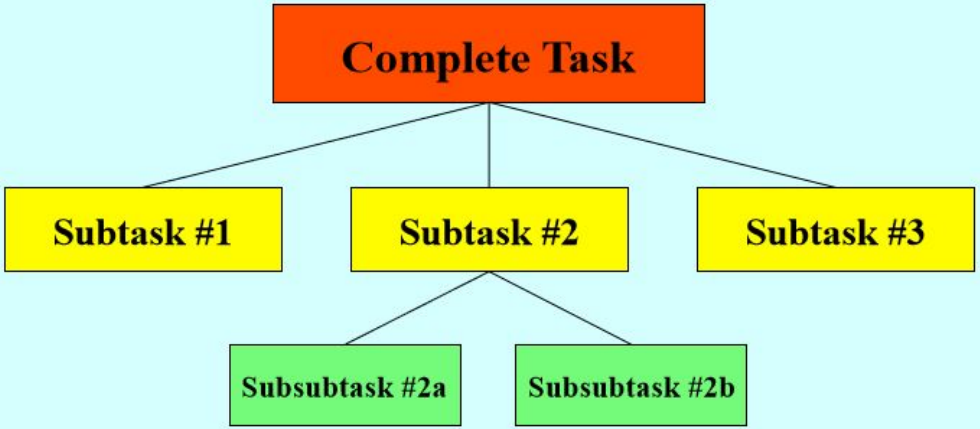
for, while loops = perform a task repeatedly (iteration)

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
```

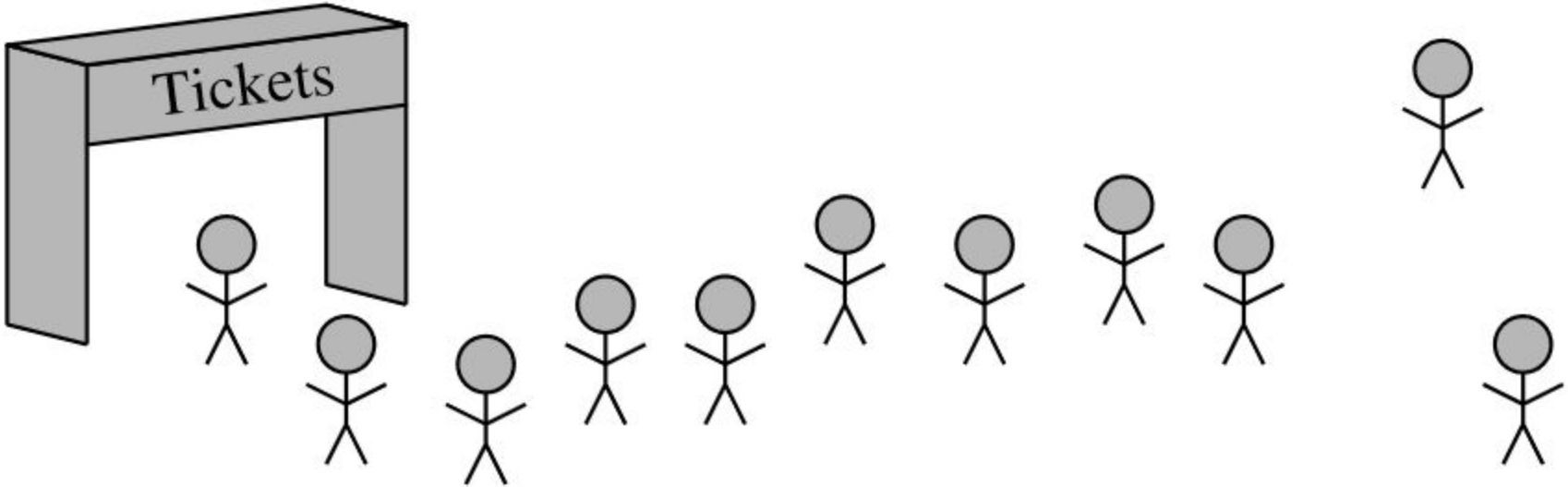
This has the same form as the original problem, and the problem size is smaller.

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)
```

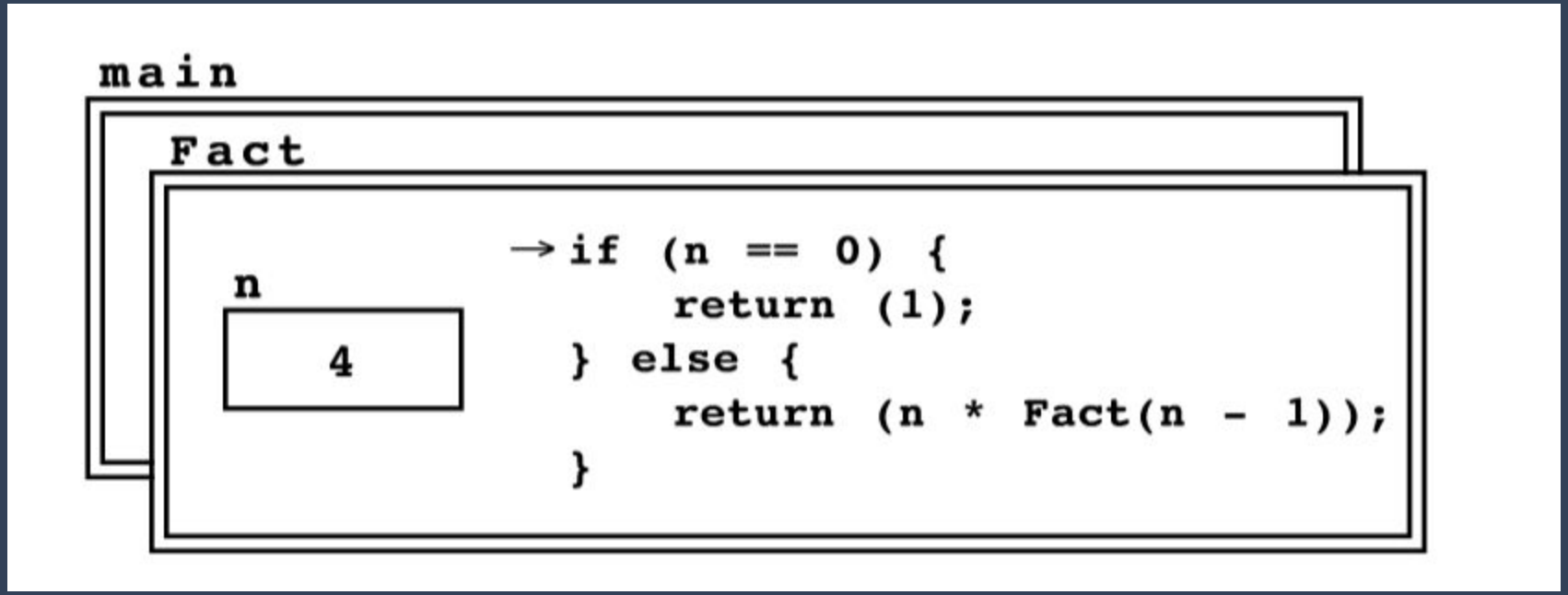
Iterative strategy

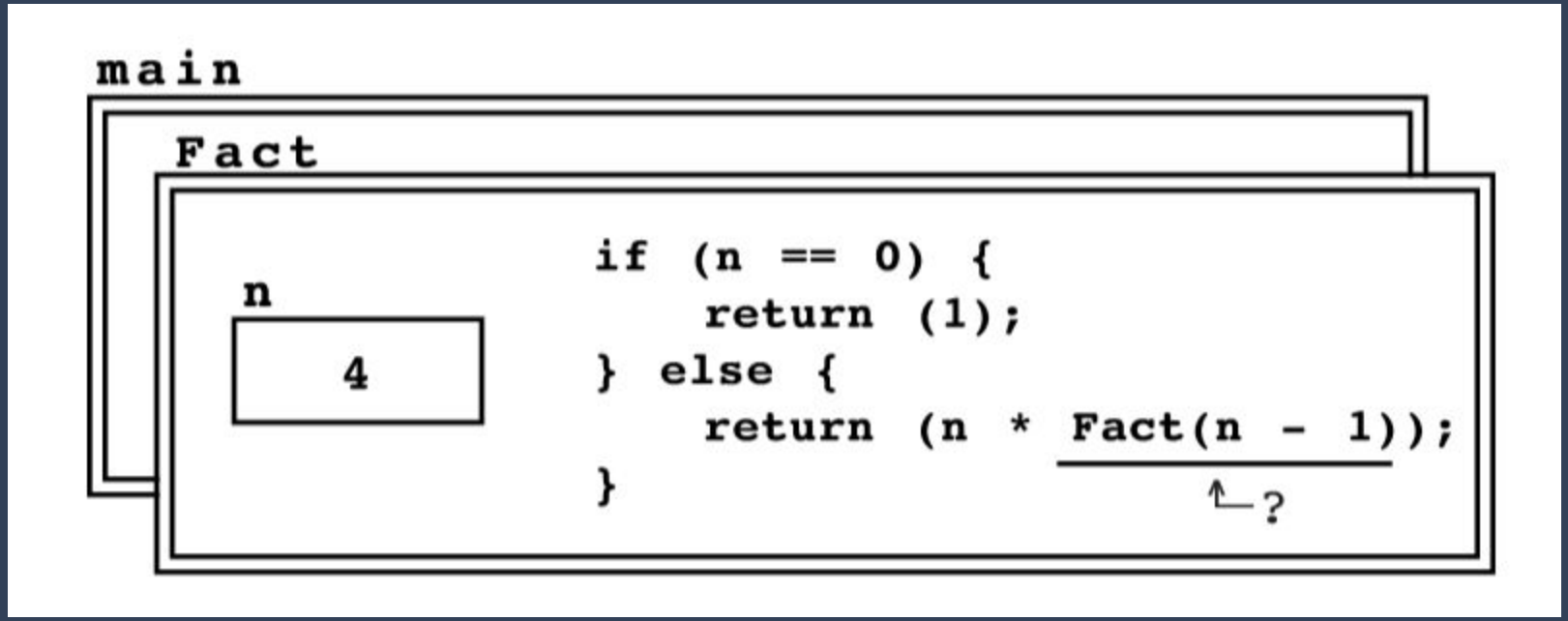
Recall our previous discussion of insertion sort: It was a decrease-and-conquer sorting algorithm that decreased a problem of size n into a subproblem of size n-1. We implemented it using a top-down recursive implementation, as well as a bottom-up iterative implementation. Can do the same here:

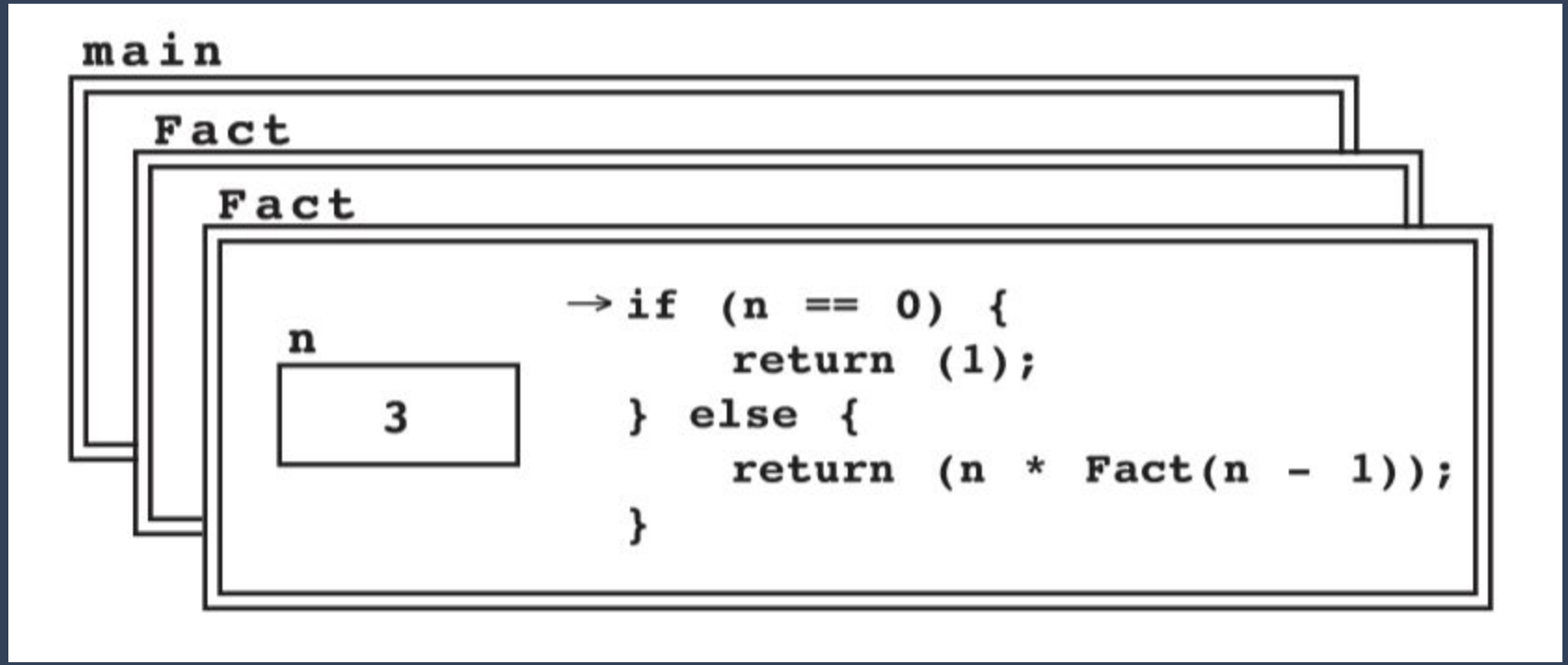
```
def fact(n):
    result = 1
    for i in 1 to n:
        result = result * i
    return result
```

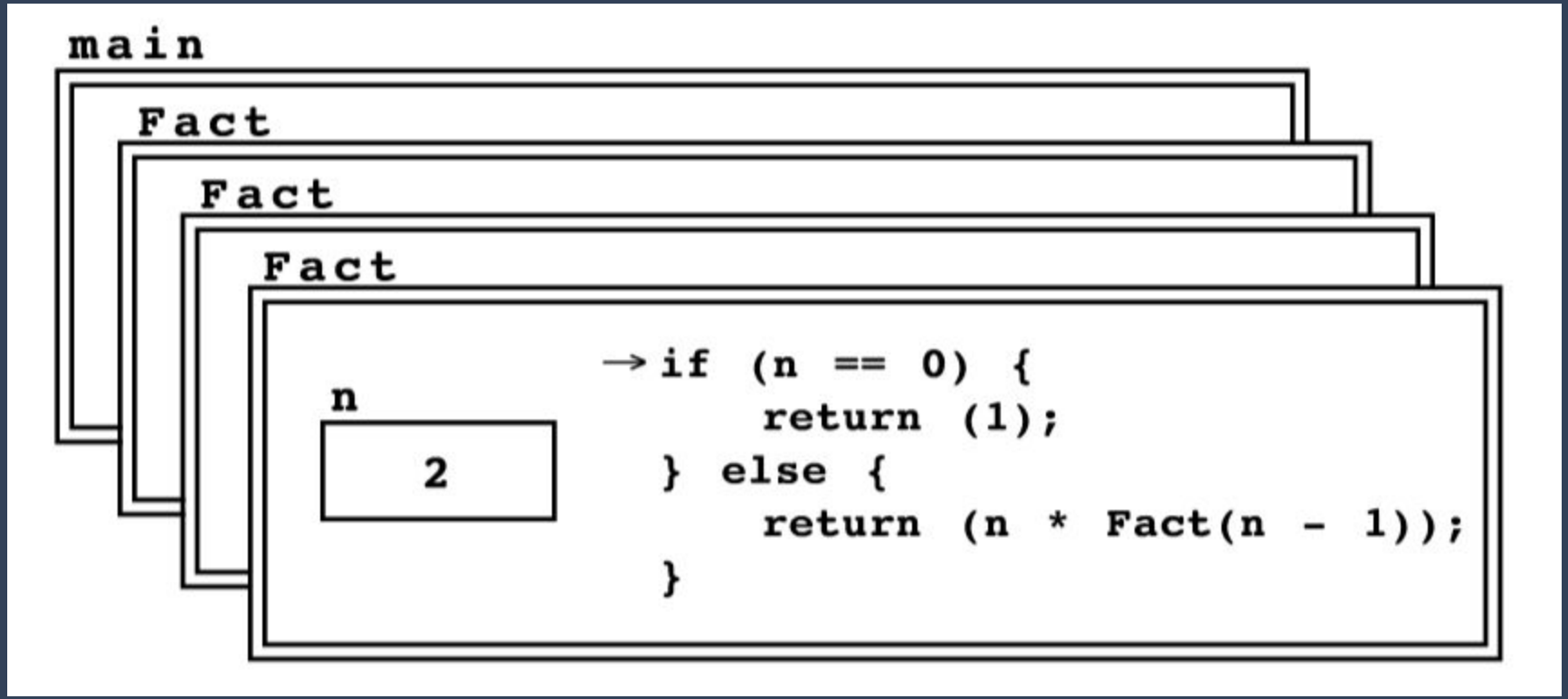
Time complexity = ?

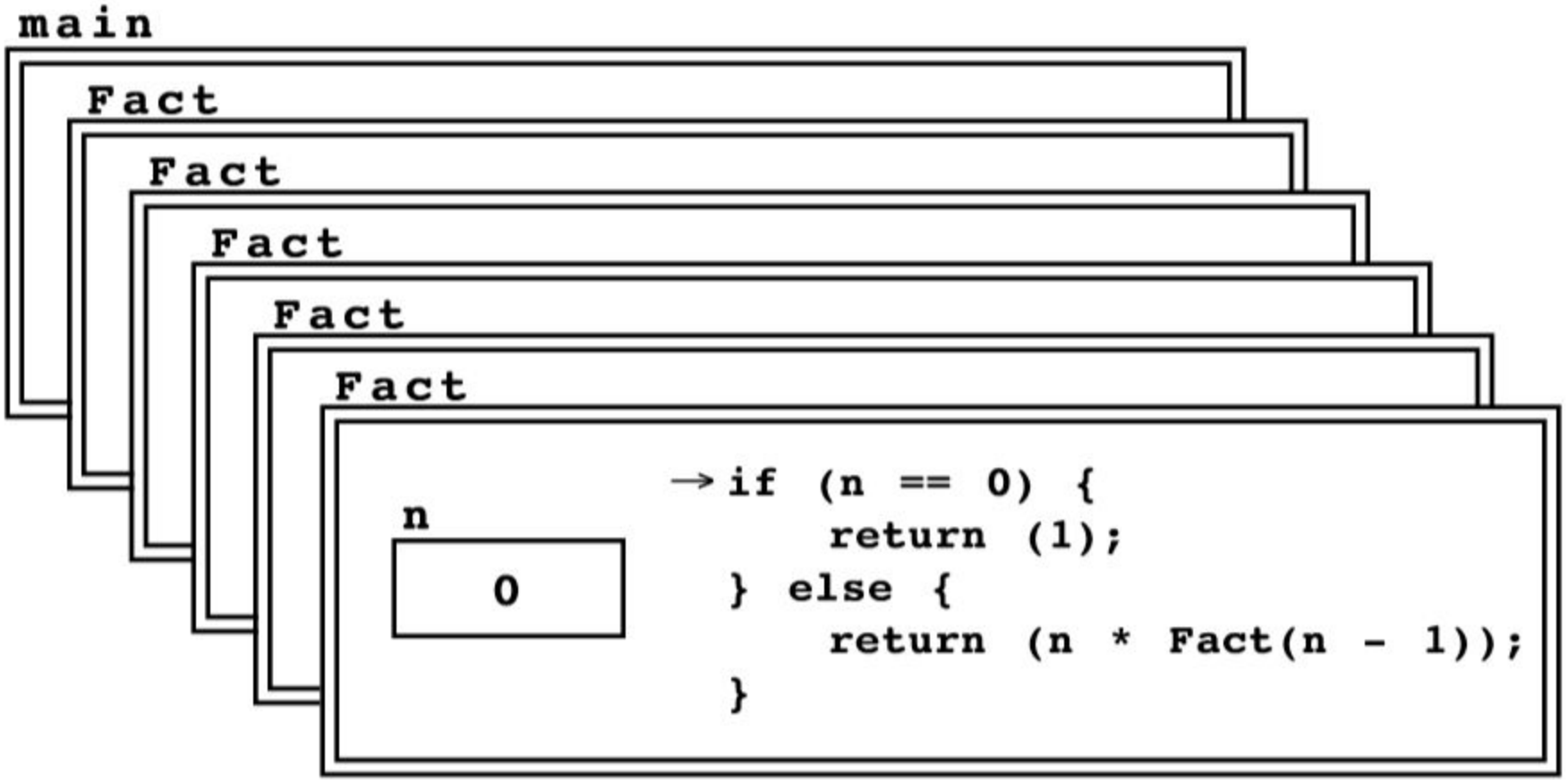
(In the main function:)
f = fact(4)

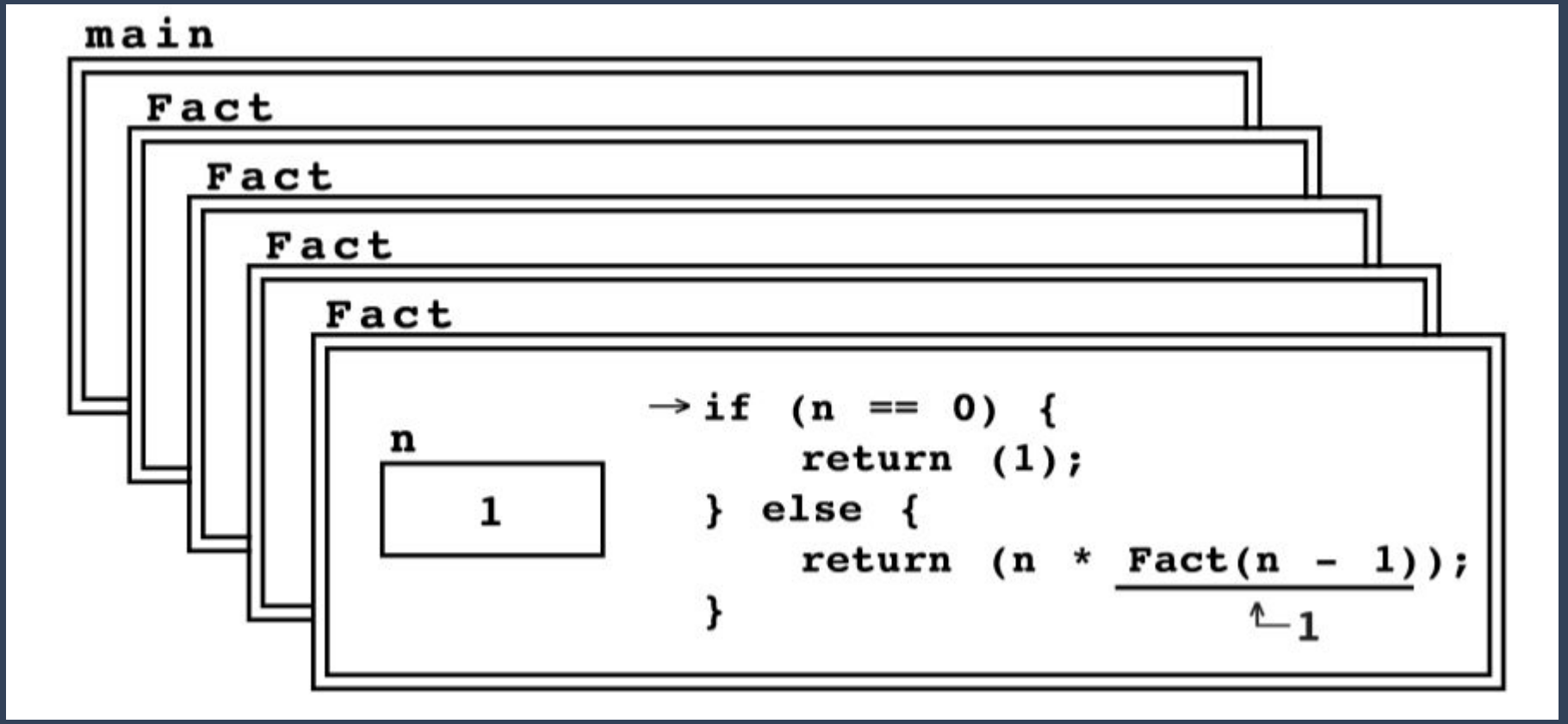


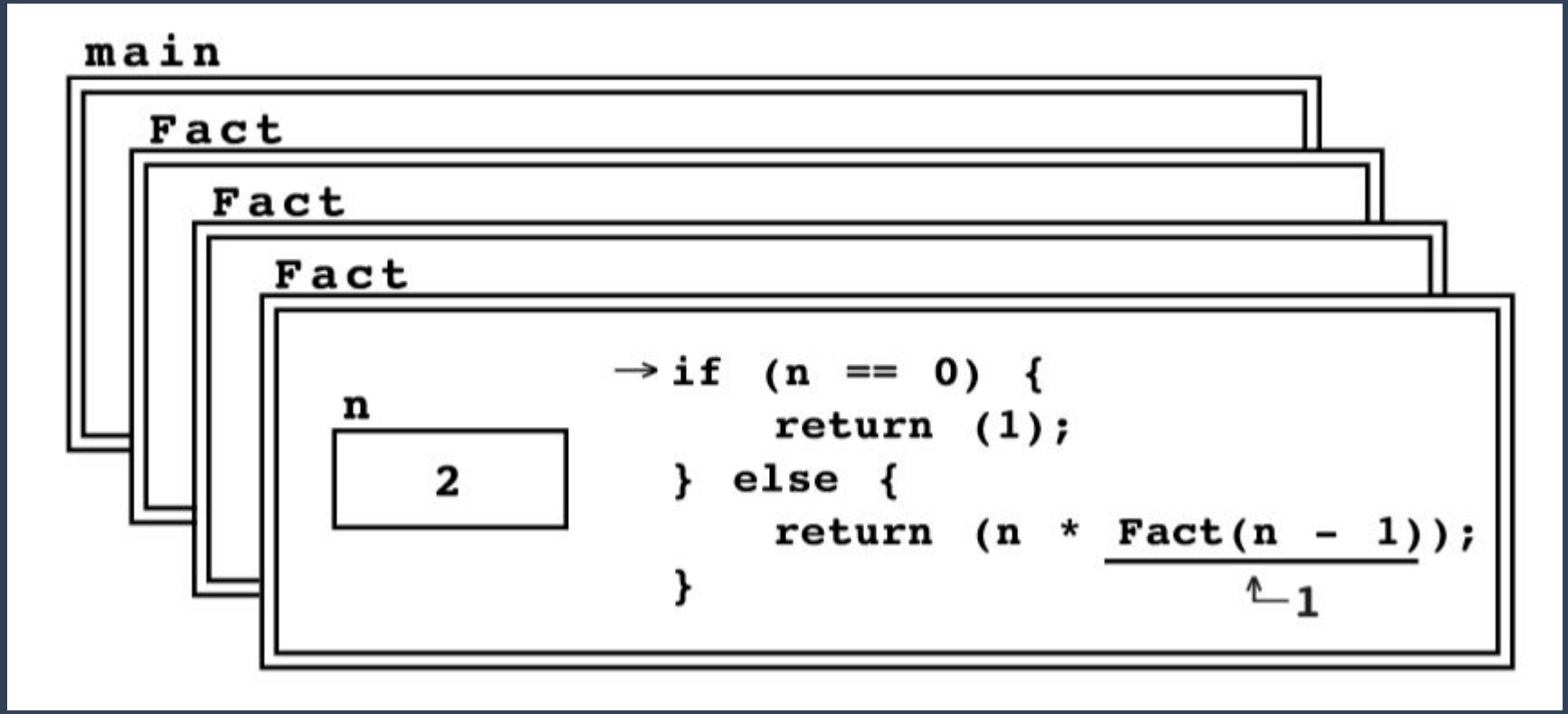


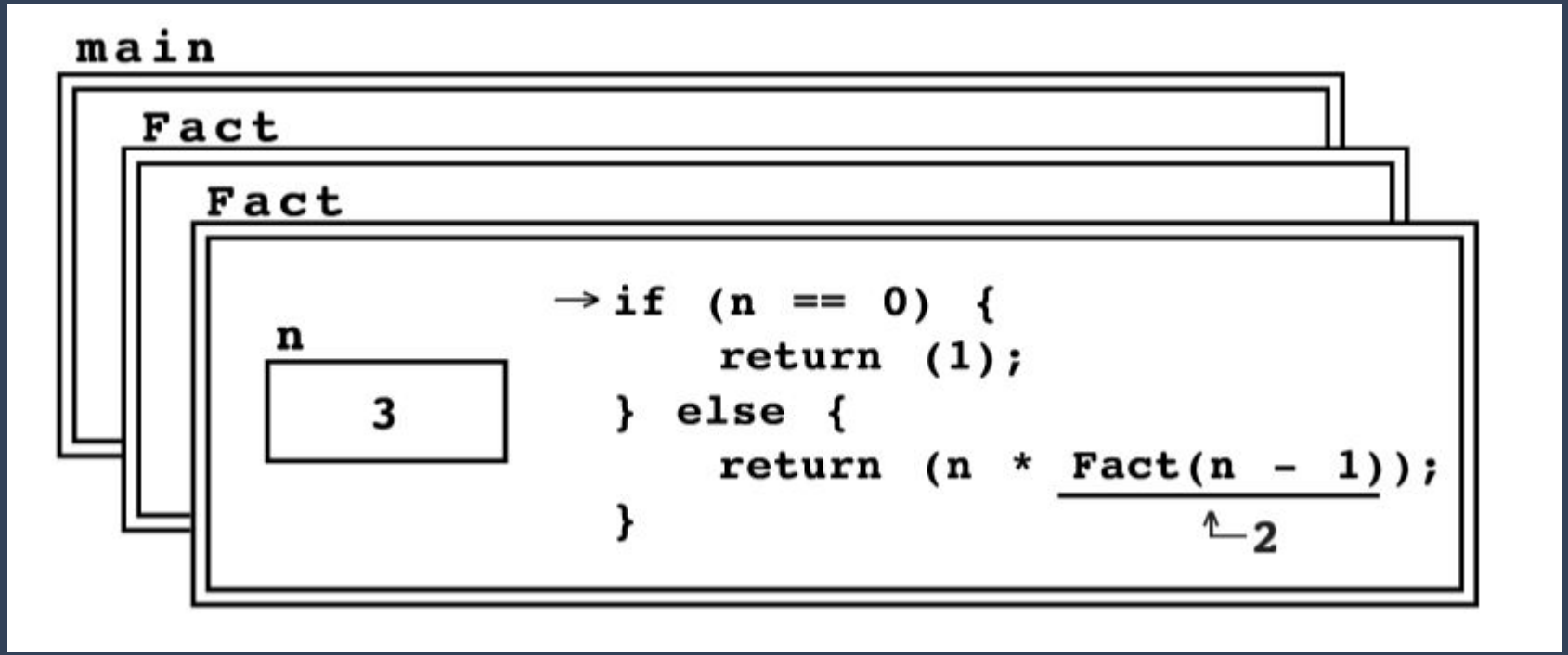


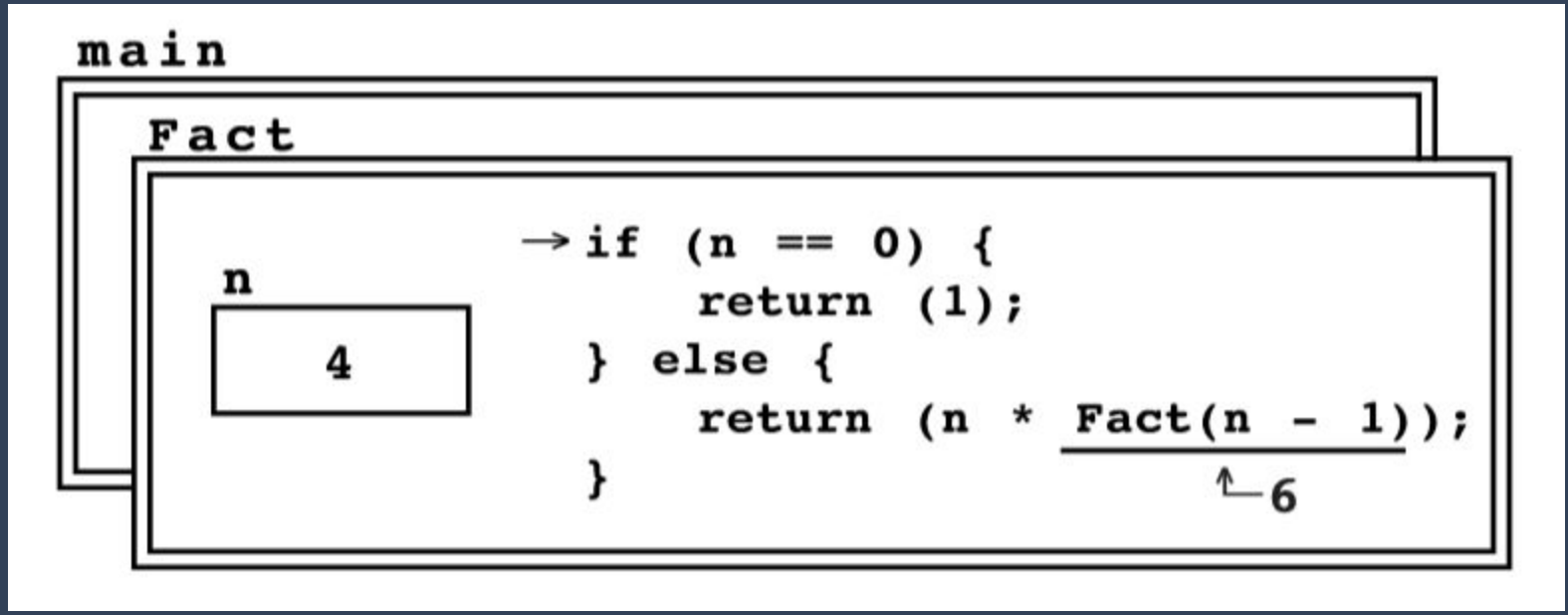












f = fact(4) ←- value of 24 returned to the main program

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.”

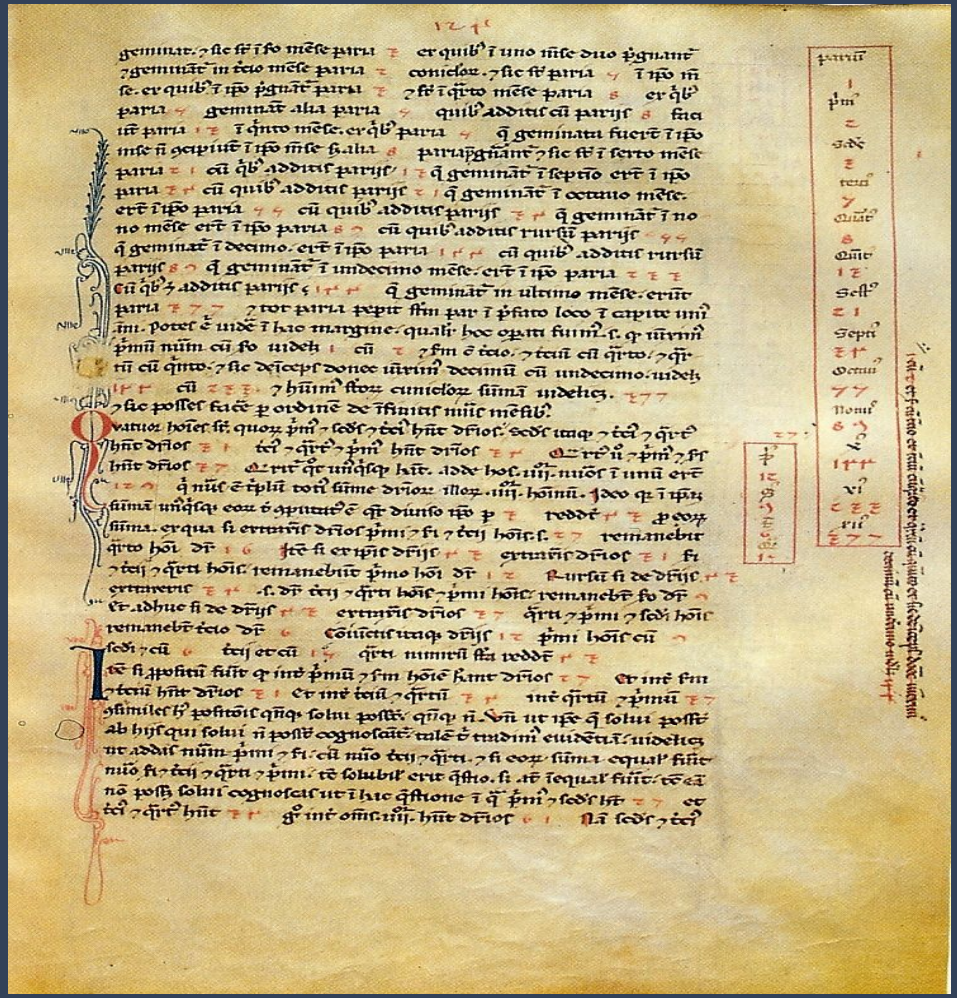
Liber abaci (Book of Counting)

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*.

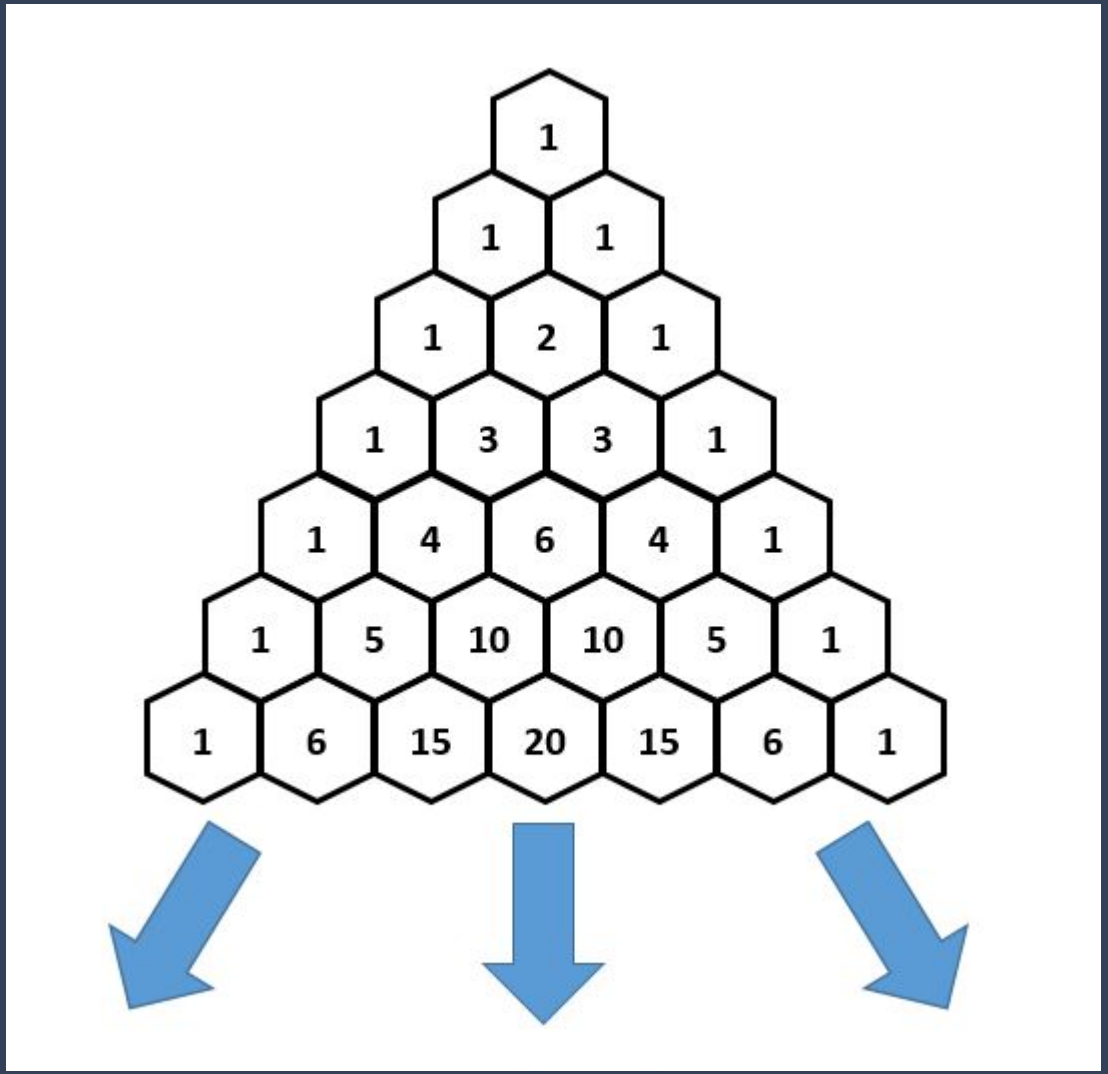
C(n,n) = ?
C(n,0) = ?

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)
```

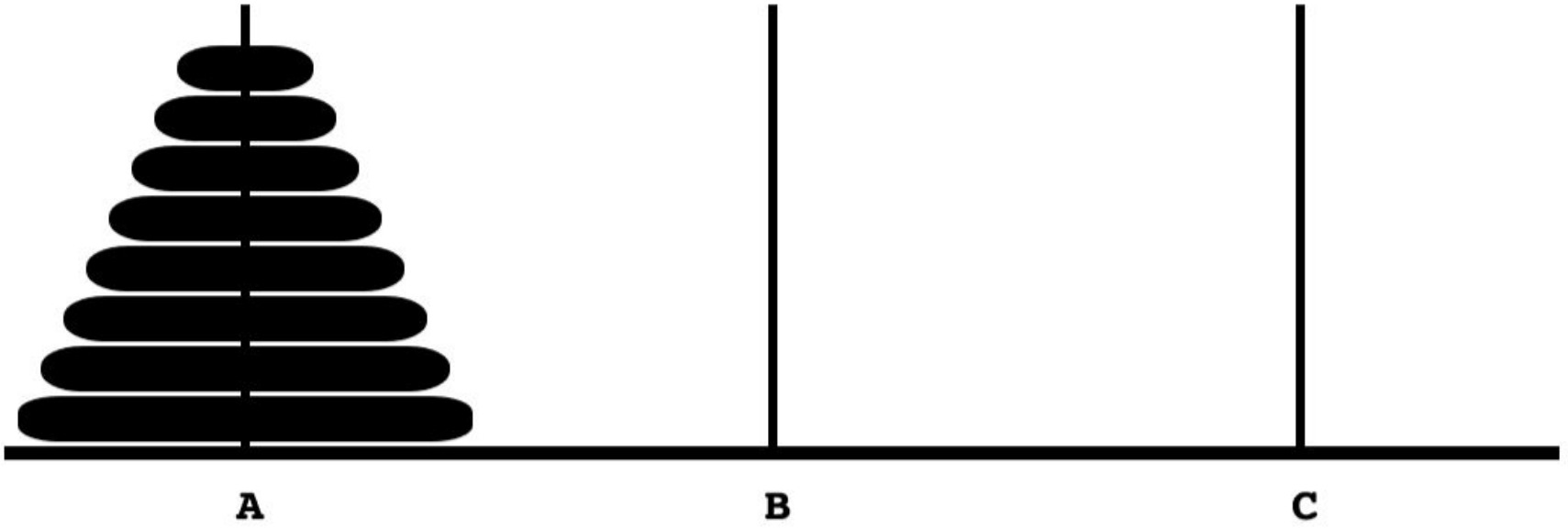
What is the sum of all the numbers in any row of Pascal’s triangle?

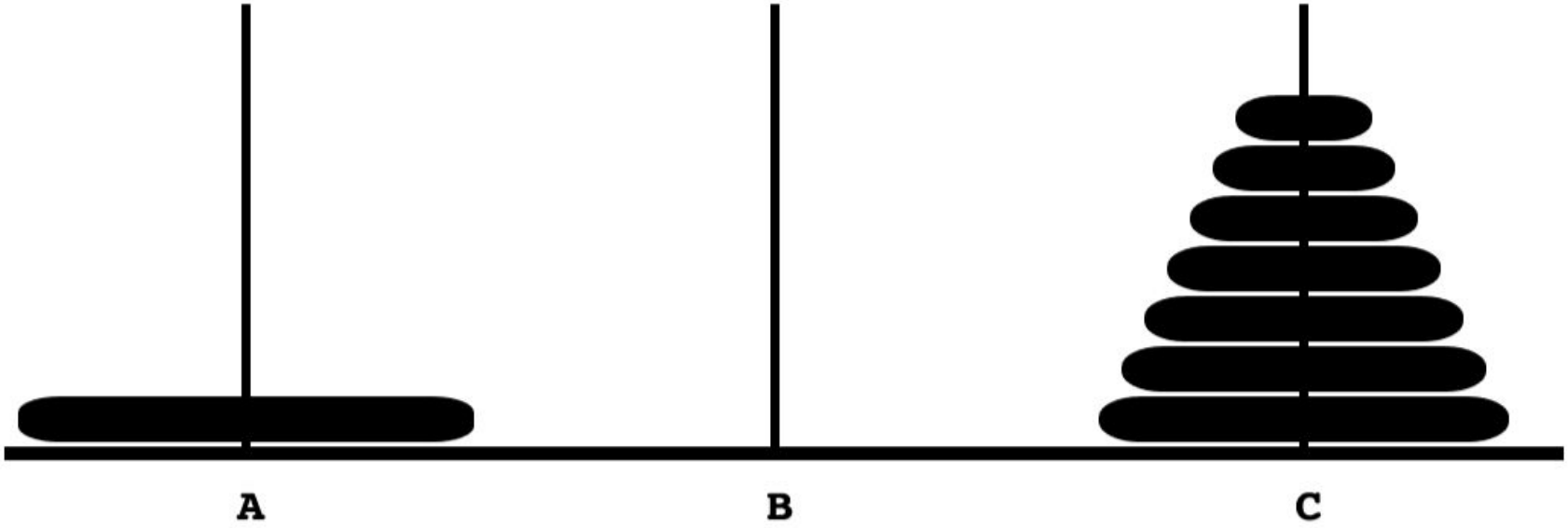
2. Recursive procedures

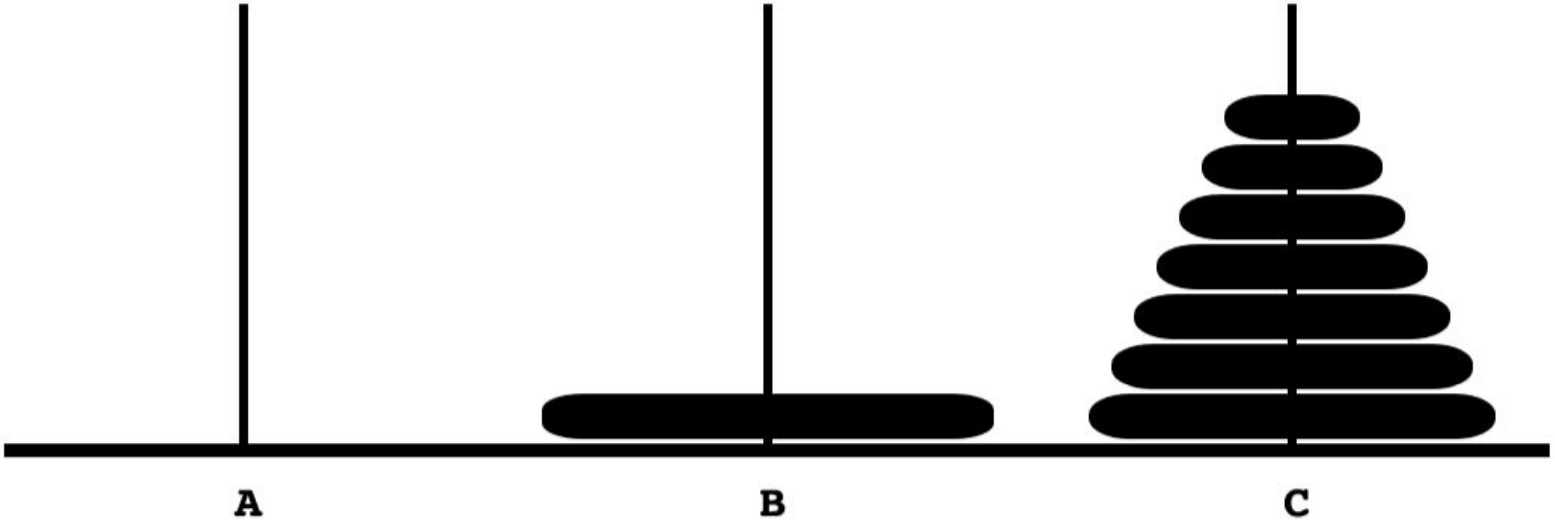
(While functions are mathematical entities, procedures are more algorithmic in character)

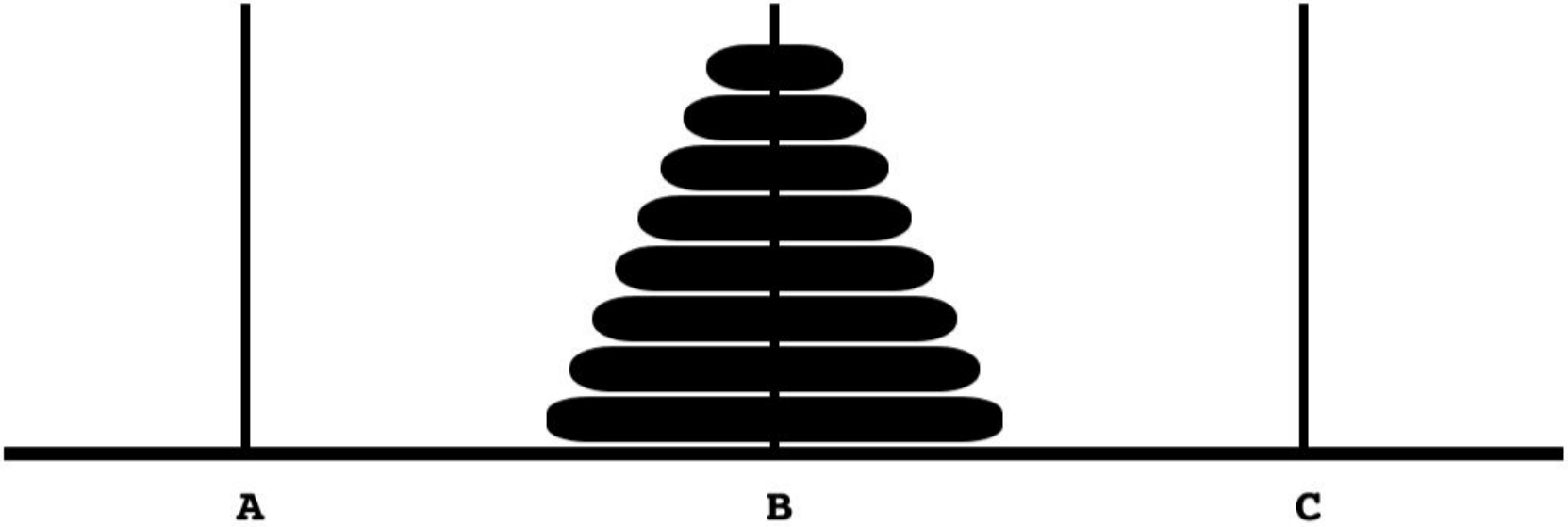
Tower of Hanoi. Invented by French mathematician Edouard Lucas in the 1880s, the Tower of Hanoi puzzle quickly became popular in Europe. Its success was due in part to the legend that grew up around the puzzle, which was described as follows in *La Nature* by the French mathematician Henri De Parville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

Recursion Live Class

{ik} INTERVIEW KICKSTART

Takeaways from preclass videos

- Why recursion is hard, and how thinking like a lazy manager is the best way to proceed.
- Examples of simple recursive mathematical functions and how they get executed on the call stack (last HW problem)
- How you come up with recursive code: Recursive implementations naturally follow from decrease-and-conquer / divide-and-conquer algorithm design strategies. Divide the overall problem into one or more subproblems ***of the same form***.
- How you think about recursive code: Make a “Recursive leap of faith”. Avoid thinking about the entire execution. Think only about the base case and how (if your subordinates do their job correctly), your code will correctly assemble the solution to the overall problem.
- Basic combinatorics (counting the number of permutations and combinations)

“Mathematics:

Some interviewers ask basic discrete math questions. This is more prevalent at Google than at other companies because we are surrounded by counting problems, probability problems, and other Discrete Math 101 situations. Spend some time before the interview refreshing your memory on (or teaching yourself) the essentials of combinatorics and probability. You should be familiar with n-choose-k problems and their ilk - the more the better.”

https://careers.google.com/how-we-hire/interview/#interviews-for-software-engineering-and-technical-roles

50. Pow(x, n)

Medium 1036 2474 Favorite Share

Implement pow(*x*, *n*), which calculates *x* raised to the power *n* (x^n).

Example 1:

```
Input: 2.00000, 10
Output: 1024.00000
```

Example 2:

```
Input: 2.10000, 3
Output: 9.26100
```

Example 3:

```
Input: 2.00000, -2
Output: 0.25000
Explanation: 2^-2 = 1/2^2 = 1/4 = 0.25
```

Note:

- -100.0 < *x* < 100.0
- *n* is a 32-bit signed integer, within the range $[-2^{31}, 2^{31} - 1]$

A recursive mathematical function for your practice

{ik} INTERVIEW KICKSTART

Today's live class

Recursion is a bit of a misnomer for today's class.
- You have seen recursion in sorting: insertion sort, merge sort and quicksort.
- You have seen recursion in searching: e.g, binary search
- Binary trees are an example of a recursive data structure. It is easy to write recursive algorithms on recursive data structures.
- You will see examples of recursion in Graphs and Dynamic Programming.

*So **recursion pervades the whole curriculum.***

- What is distinctive about today's class is the "class of problems" we are going to look at: **Combinatorial Enumeration** problems.
- Writing recursive code for these is more tricky.

Exhaustive enumeration

Requires a systematic enumeration / generation of all possible combinatorial objects (permutations or combinations).

Leads to a ***combinatorial explosion*** - exponential time complexity. Hence, *impractical for even moderate input sizes.*

Still, for many problems, we cannot do better than this.

General template

You have to fill in a series of blanks, and there is an exponential number of ways to do it.

__ __ __ __ __ __ __ __ __

This series of blanks denotes a combinatorial object (some permutation or combination).

You proceed left to right, and as a lazy manager, only take responsibility for filling in the first blank. Delegate the rest of the work to subordinates.

Write up the general strategy used by any arbitrary worker in the hierarchy as a recursive function.

General template

Note the three cases -

1. The leaf-level-workers receive the entire solution from above (and a subproblem of size 0), and their job is to print / append / filter / analyse the solutions. Their job description is mentioned in the base case of the recursive function.
2. The workers in any of the internal nodes above receive a subproblem to solve, and also the partial solution constructed by the bosses above them. They fill in the leftmost blank of the subproblem, and delegate the remaining work to their subordinates (by means of a recursive call).
3. The boss at the top (root) of the hierarchy does not receive any partial solution (or slate) from above. The boss executes the general strategy on the entire problem definition, and starts from a blank slate.

784. Letter Case Permutation

Easy 693 86 Favorite Share

Given a string S, we can transform every letter individually to be lowercase or uppercase to create another string. Return a list of all possible strings we could create.

```
Examples:
Input: S = "a1b2"
Output: ["a1b2", "a1B2", "A1b2", "A1B2"]

Input: S = "3z4"
Output: ["3z4", "3Z4"]

Input: S = "12345"
Output: ["12345"]
```

Note:

- `S` will be a string with length between `1` and `12`.
- `S` will consist only of letters or digits.

Solution with immutable parameters

```
class Solution(object):
    def letterCasePermutation(self, S):
        """
        :type S: str
        :rtype: List[str]
        """
        result = []

        def helper(S,i,slate):
            #Base case: there are no more positions left to fill in
            if i == len(S):
                result.append(slate) #Append the partial solution to the result
            else: #Recursive case: more positions left to fill in
                if S[i].isdigit(): #If the leftmost blank is a digit
                    helper(S,i+1,slate+S[i]) #Append it to the partial solution and delegate the rest
                else: #S[i] is a letter
                    helper(S,i+1,slate+S[i].upper()) #Append uppercase letter to the partial solution
                    helper(S,i+1,slate+S[i].lower()) #Append lowercase letter to the partial solution
                    #..and delegate the remaining work to a subordinate

        helper(S,0,"") #Root manager implements the overall function by calling the helper function
        #... with the same string S and i = 0 (subproblem size is the entire string)
        # ... and the partial solution is empty
        return result
```

Visualise execution using DFS tree

Time complexity = ?

Space complexity = ?

Solution with mutable parameters

```python
class Solution(object):
    def letterCasePermutation(self, S):
        """
        :type S: str
        :rtype: List[str]
        """

        result = []

        def helper(S,i,slate):
            #Base case: there are no more positions left
            if i == len(S): #Base case: there are no more positions left
                result.append("".join(slate)) #The partial solution is the full permutation
                return
            #Recursive case
            if S[i].isdigit(): #If the leftmost blank is a digit, there is only one thing to do
                slate.append(S[i]) #Add it to the partial solution
                helper(S,i+1,slate) #..and delegate the remaining blanks to a subordinate
                slate.pop() #Slate is mutable so need to revert any modifications done
            else: #S[i].isalpha() -- If the leftmost blank is a letter, there are two options
                slate.append(S[i].lower()) # Option 1 is to go with the lowercase letter
                helper(S,i+1,slate) #Delegate remaining work to a subordinate
                slate.pop()
                slate.append(S[i].upper()) # Option 2 is to go with the uppercase letter
                helper(S,i+1,slate) #Delegate remaining work to a subordinate
                slate.pop()

        helper(S,0,[]) #Root manager is called with entire string as subproblem definition
        return result

```

Optimized solution with mutable parameters

```
class Solution(object):
    def letterCasePermutation(self, S):
        """
        :type S: str
        :rtype: List[str]
        """
        result = []

        def helper(S,i):
            #Base case: there are no more positions left to fill in
            if i == len(S):
                result.append(''.join(S)) #Append the partial solution to the result
            else: #Recursive case: more positions left to fill in
                if S[i].isdigit(): #If the leftmost blank is a digit
                    helper(S,i+1) #Append it to the partial solution and delegate the rest
                else: #S[i] is a letter
                    S[i] = S[i].upper()
                    helper(S,i+1) #Append uppercase letter to the partial solution
                    S[i] = S[i].lower()
                    helper(S,i+1) #Append lowercase letter to the partial solution
                    #..and delegate the remaining work to a subordinate

        helper(list(S),0) #Root manager implements the overall function by calling the helper function
        #... with the same string S and i = 0 (subproblem size is the entire string)
        # ... and the partial solution is empty
        return result

```

Space complexity with immutable parameters = $O(n^2)$

Space complexity with mutable parameters = $O(n)$

So we will prefer mutable parameters henceforth.

78. Subsets

Medium 2142 53 Favorite Share

Given a set of **distinct** integers, *nums*, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

Example:

```
Input: nums = [1,2,3]
Output:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

```python
class Solution(object):
    def subsets(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """

        result = []

        def helper(S,i,slate):
            #Base case: No more elements left to be examined.
            if i == len(S):
                #Add the partial solution as a subset
                result.append(slate)
            else: #Recursive case:
                #There are still some elements left to be examined
                #For the leftmost element, make a choice: include it or exclude it
                #Exclude S[i] and delegate the remaining problem to a subordinate
                helper(S,i+1,slate)
                #Include S[i] and delegate the remaining problem to a subordinate
                helper(S,i+1,slate+[S[i]])

        helper(nums,0,[])

        return result
```

```python
class Solution(object):
    def subsets(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        result = []

        #Solution using mutable slate
        def helper(S,i,slate):
            #Base case: No more elements left to be examined
            if i == len(S):
                result.append(slate[:])
            else: #Recursive case
                #Exclude S[i]
                helper(S,i+1,slate)
                #Include S[i]
                slate.append(S[i])
                helper(S,i+1,slate)
                slate.pop()

        helper(nums,0,[])

        return result
```

46. Permutations

Medium 2215 67 Favorite Share

Given a collection of **distinct** integers, return all possible permutations.

Example:

```
Input: [1,2,3]
Output:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

```
class Solution(object):
    def permute(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        result = []

        def helper(slate,A):
            #slate contains the partial solution
            #Array A contains the remaining elements that we want to permute
            #Its length defines the size of the subproblem left
            #Base case:
            if len(A) == 0: #The partial solution coming down is the full solution
                result.append(slate) #Just print it out
            else: #Recursive case: we still have some elements left to permute
                for i in range(len(A)): #For each choice to fill in the leftmost blank
                    helper(slate+[A[i]],A[:i]+A[i+1:])
                    #Make the choice. Append the ith integer to the partial solution
                    #Remove the ith integer from A, and create a new array defining the subproblem
                    #... which is to be delegated to a subordinate

        helper([],nums) #Solving the original problem is the same as calling the helper function with empty slate
        #... and the complete list of input numbers in nums
        return result
```

Immutable parameters

```
    def permute(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """

        result = []

        def helper(S, i, slate):
            #Base case
            if i == len(S):
                result.append(slate[:])
                return

            #Recursive case
            for pick in range(i,len(S)):
                S[i],S[pick] = S[pick],S[i]
                slate.append(S[i])
                helper(S, i+1, slate)
                slate.pop()
                S[i],S[pick] = S[pick],S[i]

        helper(nums, 0, [])
        return result
```

Mutable parameters

```python
class Solution(object):
    def permute(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """

        result = []

        #Solution using mutable parameters

        def helper(A,i):
            #Base case: No subproblem left to solve
            if i == len(A):
                result.append(A[:])
            else: #Recursive case
                for pick in range(i,len(A)):
                    A[i],A[pick] = A[pick],A[i]
                    helper(A,i+1) #Delegate a slightly smaller subproblem
                    #A[0..i] is the partial solution
                    #A[i..n-1] is the subproblem definition
                    A[i],A[pick] = A[pick],A[i]
                    #Restore the original state before making the next choice to create the next subproblem

        helper(nums,0)

        return result
```

Mutable parameters (no separate slate)

47. Permutations II

Medium 1101 46 Favorite Share

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

Example:

```
Input: [1,1,2]
Output:
[
  [1,1,2],
  [1,2,1],
  [2,1,1]
]
```

What if there are duplicate items and we don't want to print duplicate permutations?

```python
class Solution(object):
    def permuteUnique(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """

        result = []

        #Immutable parameters
        def helper(array,i):
            #array[0..i-1] is the partial solution so far
            #array[i..n-1] is the subproblem definition
            #Base case:
            if i == len(array):
                result.append(array[:])
            else: #Recursive case
                hmap = {}
                for pick in range(i,len(array)):
                    if array[pick] not in hmap:
                        #Pick the number at index i
                        hmap[array[pick]] = 1
                        array[pick],array[i] = array[i],array[pick]
                        helper(array,i+1)
                        array[pick],array[i] = array[i],array[pick]

        helper(nums,0)

        return result
```

90. Subsets II

Medium 970 49 Favorite Share

Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

Example:

```
Input: [1,2,2]
Output:
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

```python
class Solution(object):
    def subsetsWithDup(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """

        result = []
        nums.sort() #All duplicate values will be bunched together. This allows us to count easily how many copies of each value there are.

        def helper(S,i,slate):
            #Base case: No more elements left to be considered
            if i == len(S): #The partial solution is the next subset, to be appended to the result
                result.append(slate[:])
            else: #Recursive case: S[i] is the next element to examine
                #Since there may be multiple copies of S[i], count how many there are
                count = 1 #Of course, we have already one copy
                j = i+1 #But how many more are there?
                while j <= len(S) - 1 and S[j] == S[i]:
                    #Peek forward until you see copies of S[i]. Stop when you see something different or you reach the end.
                    count += 1
                    j += 1
                #"count" now stores the number of occurrences of S[i]
                #When you make a choice for S[i], it is not just about exclude or include now
                #It is also about "how many times" to include...
                for copies in range(0,count+1): #For each choice we make about how many "copies" of S[i] to include,
                    for op in range(copies): #Append those many copies of S[i] to the slate
                        slate.append(S[i])
                    helper(S,i+count,slate) #Delegate the rest of the work to a subordinate
                    for op in range(copies): #After the subordinate comes back, undo the work done on the mutable slate
                        slate.pop()

        helper(nums,0,[])

        return result
```

17. Letter Combinations of a Phone Number

Medium 2390 319 Favorite Share

Given a string containing digits from `2-9` inclusive, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

Example:

```
Input: "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
```

Note:

Although the above answer is in lexicographical order, your answer could be in any order you want.

```python
class Solution(object):
    def letterCombinations(self, digits):
        """
        :type digits: str
        :rtype: List[str]
        """

        result = []
        hmap = { "2": ["a","b","c"], "3":["d","e","f"], "4":["g","h","i"], "5":["j","k","l"],
                 "6":["m","n","o"], "7":["p","q","r","s"], "8":["t","u","v"],
                 "9":["w","x","y","z"] }

        def helper(digitstring, i, slate):
            #Base case: If there are no more digits left to be examined
            if i == len(digitstring):
                if len(slate) > 0:
                    result.append("".join(slate))
            else: #Recursive case
                for letter in hmap[digitstring[i]]:
                    slate.append(letter)
                    helper(digitstring,i+1,slate)
                    slate.pop()

        helper(digits,0,[])

        return result
```

Backtracking

Sometimes, we can evaluate the partially constructed solutions to a combinatorial search problem: if none of the ways to fill in the remaining blanks can lead to a solution (or if we can directly identify the one, single solution that will emerge), we do not need to work to fill in the remaining blanks.

Instead, we backtrack from that internal node, and thus prune away the subtree rooted at it.

Though it leads to an exponential improvement in combinatorial search, the resulting time is not polynomial, but an “improved exponential time” (it is difficult to give an exact expression for the time complexity). It is best treated as a heuristic than something amenable to worst-case analysis.

77. Combinations

Medium 843 48 Favorite Share

Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.

Example:

```
Input: n = 4, k = 2
Output:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

```python
class Solution(object):
    def combine(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: List[List[int]]
        """

        result = []

        def helper(n, i, k, slate):
            #Backtracking case:
            if len(slate) == k:
                result.append(slate[:])
                return
            #Base case:
            if i == n+1:
                return
            #Recursive case
            #Exclude i
            helper(n,i+1,k,slate)
            #Include i
            slate.append(i)
            helper(n,i+1,k,slate)
            slate.pop()

```

Check backtracking case before base case. Why?

Count number of subsets that sum to k

```
#Backtracking

def subsetsum(S,k):
  return shelper(S,0,k,[],0)

def shelper(S,i,k,slate,slatesum):
  if slatesum > k:
    return 0
  if slatesum == k:
    return 1
  if i == len(S):
    return 0
  else:
    return shelper(S,i+1,k,slate,slatesum) + shelper(S,i+1,k,slate+[S[i]],slatesum+S[i])


print subsetsum([1,2,3,4,5],4)
```

After enumeration:
Decision problem: True/False answer
Counting problem: return count
Optimization problem: return best subset
(Knapsack problem)

22. Generate Parentheses

Medium 3017 188 Favorite Share

Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given *n* = 3, a solution set is:

```
[
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

```python
class Solution(object):
    def generateParenthesis(self, n):
        """
        :type n: int
        :rtype: List[str]
        """

        result = []

        def helper(numleft, numright, slate):
            #Backtracking case
            if numleft > numright or numleft < 0 or numright < 0:
                return

            #Base case
            if numleft == numright == 0:
                result.append("".join(slate))
                return

            #Recursive case
            #Include (
            slate.append('(')
            helper(numleft-1,numright,slate)
            slate.pop()

            #Include )
            slate.append(')')
            helper(numleft, numright-1,slate)
            slate.pop()

        helper(n,n,[])

        return result
```

131. Palindrome Partitioning

Medium 1013 39 Favorite Share

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of *s*.

Example:

```
Input: "aab"
Output:
[
  ["aa","b"],
  ["a","a","b"]
]
```

```python
class Solution(object):
    def partition(self, s):
        """
        :type s: str
        :rtype: List[List[str]]
        """

        result = []

        def ispalindrome(st):
            #return st == st[::-1] -- short cut
            i = 0
            while i <= len(st)/2:
                if st[i] != st[len(st)-1-i]:
                    return False
                i += 1
            return True

        def helper(array,i,slate):
            #Backtracking case
            #If the last string on the slate is not a palindrome, backtrack
            if len(slate) > 0 and not ispalindrome(slate[-1]):
                    return

            #Base case
            if i == len(array):
                result.append(slate[:])
                return

            #Recursive case
            for pick in range(i,len(array)):
                #append array[i..pick] on the slate
                slate.append(array[i:pick+1])
                helper(array,pick+1,slate)
                slate.pop()

        helper(s,0,[])
        return result
```

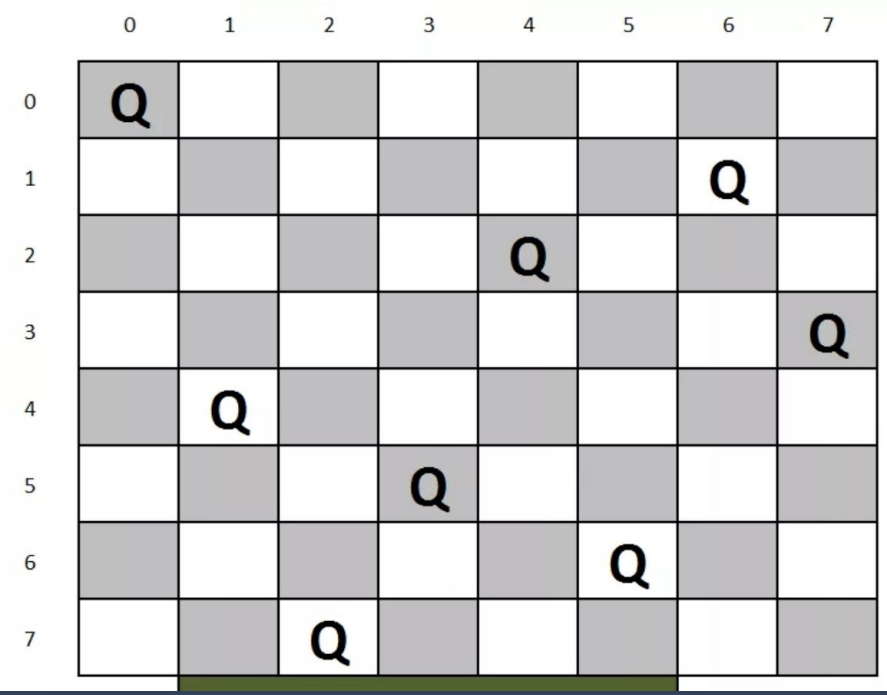
```python
class Solution(object):
    def partition(self, s):
        """
        :type s: str
        :rtype: List[List[str]]
        """

        result = []

        def ispalindrome(st):
            #return st == st[::-1] -- short cut
            i = 0
            while i <= len(st)/2:
                if st[i] != st[len(st)-1-i]:
                    return False
                i += 1
            return True

        def helper(array,i,slate):
            #Backtracking case
            #If the last string on the slate is not a palindrome, backtrack
            #if len(slate) > 0 and not ispalindrome(slate[-1]):
            #        return

            #Base case
            if i == len(array):
                result.append(slate[:])
                return

            #Recursive case
            for pick in range(i,len(array)):
                #append array[i..pick] on the slate
                if ispalindrome(array[i:pick+1]):
                    slate.append(array[i:pick+1])
                    helper(array,pick+1,slate)
                    slate.pop()

        helper(s,0,[])
        return result
```

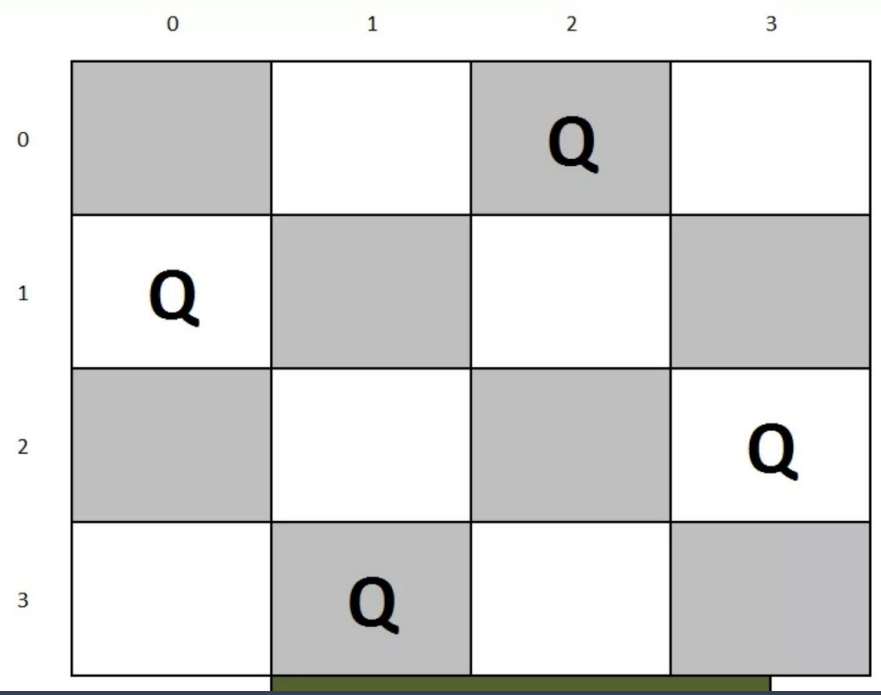
Our combinatorial objects are not always built out of numbers or characters.

They can also be built from squares on a game board.

(Later: you will see that they can also be built from vertices or edges of a graph)

The N-queens problem





Max Bezzel, 1848

HISTORIA MATHEMATICA 4 (1977), 397-404

GAUSS AND THE EIGHT QUEENS PROBLEM: A STUDY IN MINIATURE OF THE PROPAGATION OF HISTORICAL ERROR

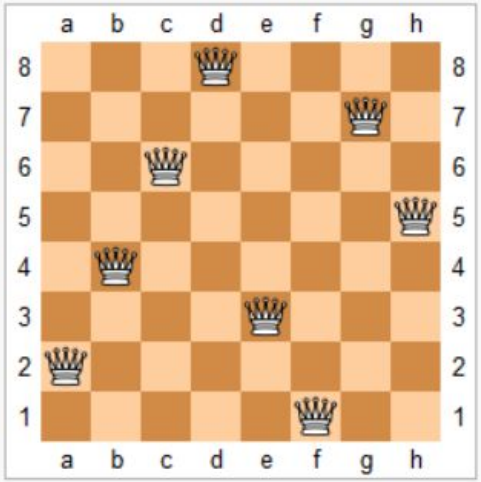
BY PAUL J. CAMPBELL,
BELOIT COLLEGE, BELOIT, WI 53511

no proof that there are not more. To resolve the question by a careful enumeration of solutions via trial and error, continued Gauss, would take only an hour or two. Apparently such inelegant work held little attraction for Gauss, for he does not seem to have carried it out, despite outlining in detail how to go about it. He continued on in the letter to reformulate the problem as an arithmetic one, and to relate it to the representation of

51. N-Queens

Hard 1213 55 Favorite Share

The *n*-queens puzzle is the problem of placing *n* queens on an *n*×*n* chessboard such that no two queens attack each other.



One solution to the eight queens puzzle

Given an integer *n*, return all distinct solutions to the *n*-queens puzzle.

Each solution contains a distinct board configuration of the *n*-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.

```python
class Solution(object):
    def solveNQueens(self, n):
        """
        :type n: int
        :rtype: List[List[str]]
        """
        result = []

        def noconflict(slate,col):
            #Return true if a new queen placed at row = len(slate) and col is attacking
            #... or being sttacked by an existing queen
            #Row numbers are clearly different already
            #If any other queen lies in the same col, then return False
            for queenrow in range(len(slate)):
                if slate[queenrow] == col:
                    return False
                #If any other queen lies on the same diagonal, then also: return False
                rowdiff = abs(len(slate) - queenrow)
                coldiff = abs(col - slate[queenrow])
                if rowdiff == coldiff:
                    return False
            return True

        def helper(slate,i,n): #place queens i onwards
            if i == n:
                #All n queens (from 0 to n-1) have been placed, so append to result
                result.append(slate[:])
                return
            #Recursive case
            for col in range(0,n):
                if noconflict(slate,col):
                    slate.append(col)
                    helper(slate,i+1,n)
                    slate.pop()

        helper([],0,n)
        return result
```

Please implement all the problems we did in the class. You will be good for the test.

If you have more time, do problems 2, 3, 4.

If you have more time, then do problems 1, 5 (problem 1 is the hardest!)

Appendix

```
def nqueens(n):
  nqhelper(n,1,[])
  #Place n queens on the board starting with queen 1

def nqhelper(n,i,slate): #Place ith queen, and delegate rest of the work
  if i == n+1: #All n queens from 1 to n have been placed
    print slate
  else:
    #Try to place the ith queen in every possible position from cols 1 to n
    for col in range(1,n+1):
      slate.append(col)
      if noconflict(slate):
        nqhelper(n,i+1,slate)
      slate.pop()


def noconflict(slate):
  last = len(slate)
  if last < 2:
    return True
  else:
    #Check for conflict between the last queen and all earlier queens
    for i in range(last-1):
      rowdiff = abs(last-1-i)
      coldiff = abs(slate[last-1] - slate[i])
      if rowdiff == coldiff or slate[i] == slate[last-1]:
        return False
    return True

nqueens(6)
```

A solution that works with the same copy of input array and slate

```
def permutelist(array):
  phelper(array,0,[])

def phelper(array,index,slate):
  if index == len(array):
    print slate
  else:
    for i in range(index,len(array)):
      (array[index],array[i]) = (array[i],array[index])
      slate.append(array[index])
      phelper(array,index+1,slate)
      slate.pop()
      (array[index],array[i]) = (array[i],array[index])

permutelist([1,2,3,4,5])
```

Is slate needed?

Print only those subsets of S which add to a given number k

```
def subsetsum(S,k):
  shelper(S,0,k,[])

def shelper(S,i,k,slate):
  if i == len(S):
    #Check if the subset adds up to k
    sum = 0
    for number in slate:
      sum += number
    if sum == k:
      print slate
  else:
    shelper(S,i+1,k,slate)
    shelper(S,i+1,k,slate+[S[i]])


subsetsum([1,2,3,4,5],4)
```

Print only those subsets of S which add to a given number k

```
def subsetsum(S,k):
  shelper(S,0,k,[],0)

def shelper(S,i,k,slate,slatesum):
  if i == len(S):
    if slatesum == k:
      print slate
  else:
    shelper(S,i+1,k,slate,slatesum)
    shelper(S,i+1,k,slate+[S[i]],slatesum+S[i])


subsetsum([1,2,3,4,5],4)
```

Print only those subsets of S which add to a given number k

```
#Backtracking

def subsetsum(S,k):
  shelper(S,0,k,[],0)

def shelper(S,i,k,slate,slatesum):
  if slatesum > k:
    return
  if slatesum == k:
    print slate
    return
  if i == len(S):
    return
  else:
    shelper(S,i+1,k,slate,slatesum)
    shelper(S,i+1,k,slate+[S[i]],slatesum+S[i])


subsetsum([1,2,3,4,5],4)
```

```
def numsteps(n,S):
  if len(S) == 0:
    print "Invalid input"
    return -1
  return numstepshelper(n,S)

def numstepshelper(k,S):
  if k == 0:
    return 1
  #else...
  sum = 0
  for stepsize in S:
    if k - stepsize >= 0:
      sum += numstepshelper(k-stepsize,S)
  return sum

print numsteps(10,[1,8])
```

```
#How many paths from (1,1) to (m,n)?

def numpaths(m,n):
  return numpathshelper(1,1,m,n)

def numpathshelper(i,j,m,n):
  if i == m and j == n:
    return 1
  if i > m or j > n:
    return 0
  else:
    return numpathshelper(i+1,j,m,n) + numpathshelper(i,j+1,m,n)



print numpaths(2,2)
print numpaths(3,3)
```

Credits: Omkar Deshpande

{ik} INTERVIEW KICKSTART