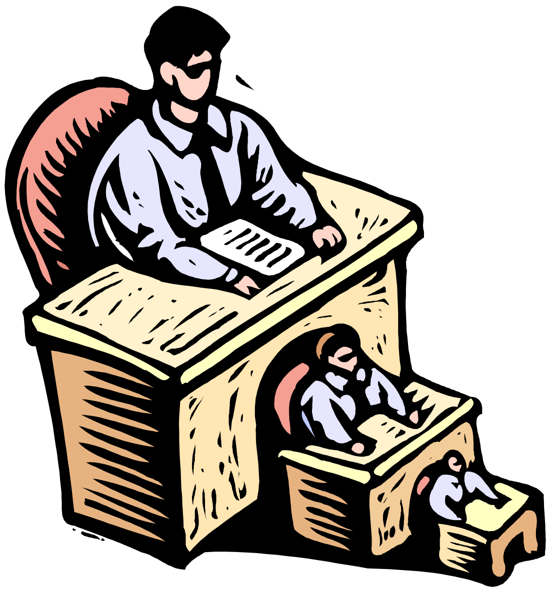
Recursion



Definition

- Programming technique

  A function can call itself

- One of the central ideas of computer science

- It's super effective!

  "The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions." – *Niklaus Wirth*

  "I'm lovin' it" – *Charles Ponzi*

World's Simplest Recursion Program

```
def count (index):
        print (index)
        if index < 2:
                count(index+1)
        return

if __name__ == '__main__':
        count(0)
```

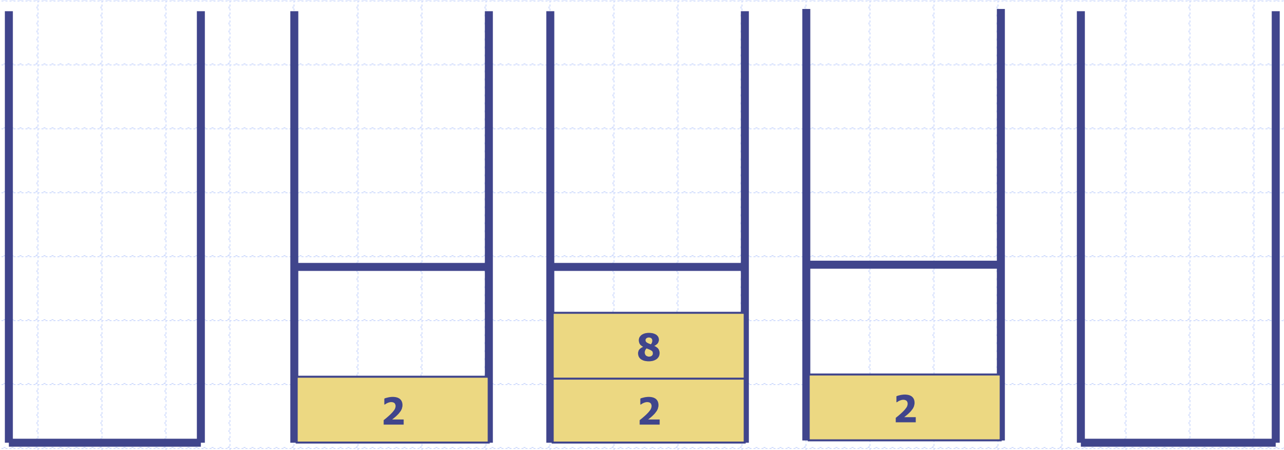
This is where the recursion occurs. You can see that the count() function calls itself.

Visualizing Recursion

- To understand how recursion works, it helps to visualize what's going on.
- To help visualize, we will use a common concept called the *Stack*.
- A stack basically operates like a container of trays in a cafeteria. It has only two operations:
  - Push: you can push something onto the stack.
  - Pop: you can pop something off the top of the stack.
- Let's see an example stack in action.

Stacks

The diagram below shows a stack over time.
We perform two pushes and one pop.



Time: 0
Empty Stack

Time 1:
Push "2"

Time 2:
Push "8"

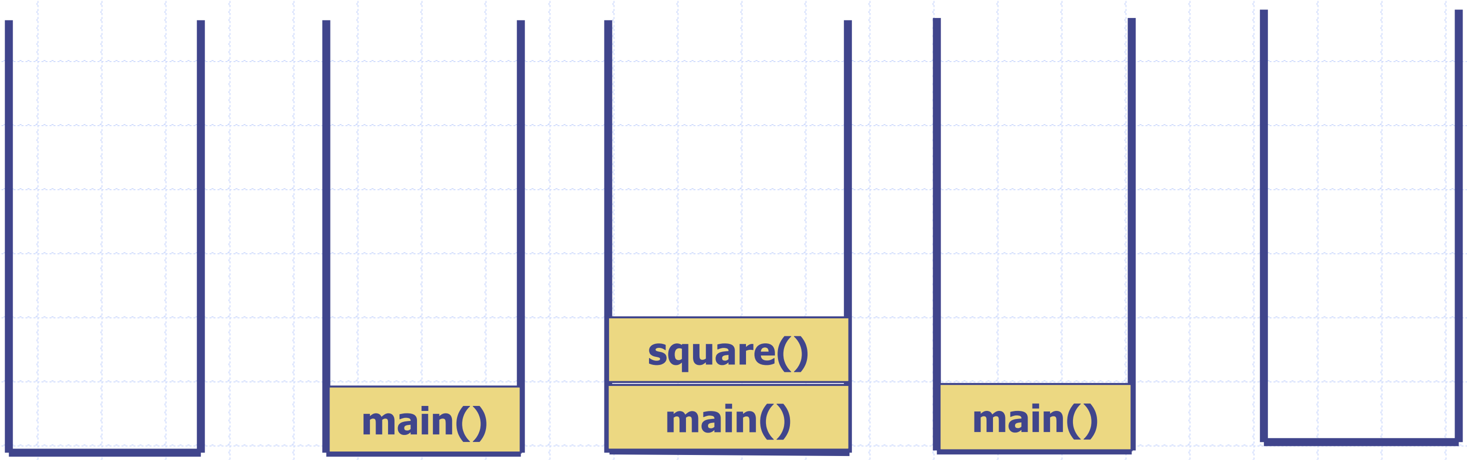
Time 3:
Pop: Gets 8

Time 4:
Pop: Gets 2

Stacks and Methods

- When you run a program, the computer creates a stack for you.
- Each time you invoke a method, the method is placed on top of the stack.
- When the method returns or exits, the method is popped off the stack.
- The diagram on the next page shows a sample stack for a simple program.

Stacks and Methods



Time: 0
Empty Stack

Time 1:
Push: main()

Time 2:
Push: square()

Time 3:
Pop: square()
returns a value.
method exits.

Time 4:
Pop: main()
returns a value.
method exits.

Stacks and Recursion

- Each time a method is called, you *push* the method on the stack.
- Each time the method returns or exits, you *pop* the method off the stack.
- If a method calls itself recursively, you just push another copy of the method onto the stack.
- We therefore have a simple way to visualize how recursion really works.

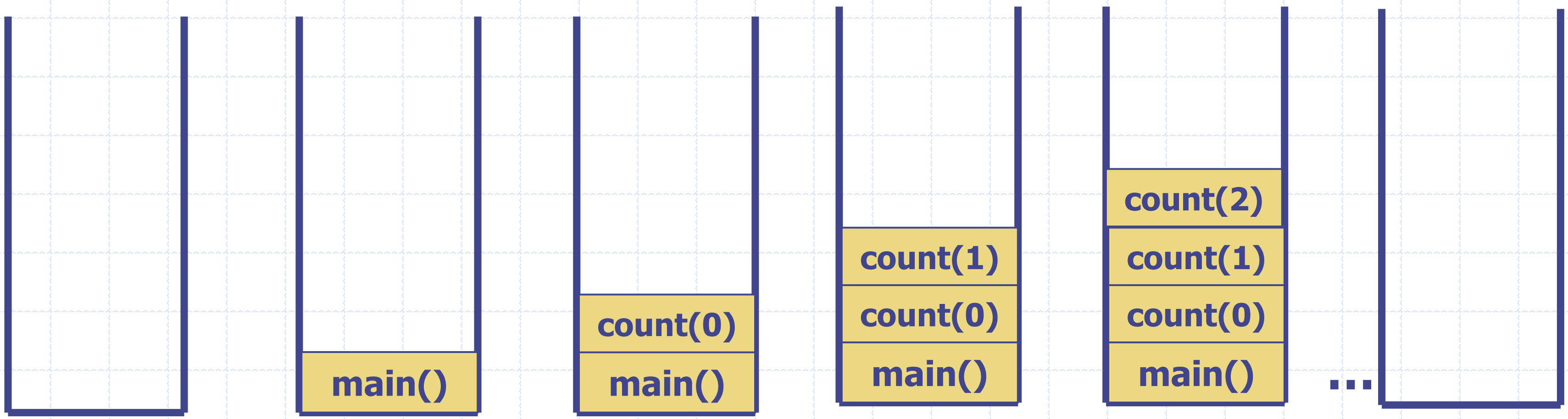
Back to the Simple Recursion Program

- Here's the code again. Now, that we understand stacks, we can visualize the recursion.

```
def count (index):
        print (index)
        if index < 2:
                count(index+1)
        return

if __name__ == '__main__':
        count(0)
```

Stacks and Recursion in Action



Time: 0
Empty Stack

Time 1:
Push: main()

Time 2:
Push:
count(0)

Time 3:
Push:
count(1)

Time 4:
Push:
count(2)

Times 5-8:
Pop everything

Inside count(0):
print (index)
→ 0
if index < 2:
count(index+1)

Inside count(1):
print (index)
→ 1
if index < 2:
count(index+1)

Inside count(2):
print (index)
→ 2
if index < 2:
count(index+1)

This condition now fails!

Hence, recursion stops, and we proceed to pop all functions off the stack.

Recursion, Variation 1

What will the following program do?

```
def count (index):
    print (index)
    if index < 2:
        count(index+1)
    return

if __name__ == '__main__':
    count(3)
```

Recursion, Variation 2

What will the following program do?

```
def count (index):
        if index < 2:
                count(index+1)
        print (index)
        return

if __name__ == '__main__':
        count(0)
```

Note that the print statement has been moved to the end of the method.

Recursion Example #2

Recursion Example #2

```
def upAndDown (n):
        print("level:",n)
        if n < 4:
                upAndDown(n+1)
        print("LEVEL:",n)
        return

upAndDown(1)
```

Recursion occurs here.

Determining the Output

- Suppose you were given this problem on the final exam, and your task is to "determine the output."
- How do you figure out the output?
- Answer: Use Stacks to Help Visualize

Stack Short-Hand

- Rather than draw each stack like we did last time, you can try using a short-hand notation.

time	stack	output
time 0:	empty stack	
time 1:	f(1)	Level: 1
time 2:	f(1), f(2)	Level: 2
time 3:	f(1), f(2), f(3)	Level: 3
time 4:	f(1), f(2), f(3), f(4)	Level: 4
time 5:	f(1), f(2), f(3)	LEVEL: 4
time 6:	f(1), f(2)	LEVEL: 3
time 7:	f(1)	LEVEL: 2
time 8:	empty	LEVEL: 1

Factorials

- Computing factorials are a classic problem for examining recursion.
- A factorial is defined as follows:

  n! = n * (n-1) * (n-2) …. * 1;
- For example:

  1! = 1 (Base Case)

  2! = 2 * 1 = 2

  3! = 3 * 2 * 1 = 6

  4! = 4 * 3 * 2 * 1 = 24

  5! = 5 * 4 * 3 * 2 * 1 = 120

Iterative Approach

```
def findFactorialIterative(n)
        if n<0:
                return 0
        factorial = 1
        while n>0:
                factorial = factorial*n
                n = n-1
        return factorial

print(findFactorialInterative(5))
```

This is an iterative solution to finding a factorial.
It's iterative because we have a simple for loop.
Note that the for loop goes from n-1 to 1.

Factorials

- Computing factorials are a classic problem for examining recursion.
- A factorial is defined as follows:

  n! = n * (n-1) * (n-2) …. * 1;
- For example:

  1! = 1 (Base Case)

  2! = 2 * 1 = 2

  3! = 3 * 2 * 1 = 6

  4! = 4 * 3 * 2 * 1 = 24

  5! = 5 * 4 * 3 * 2 * 1 = 120

If you study this table closely, you will start to see a pattern.
The pattern is as follows:
You can compute the factorial of any number (n) by taking n and multiplying it by the factorial of (n-1).

For example:
5! = 5 * 4!
(which translates to 5! = 5 * 24 = 120)

Seeing the Pattern

- Seeing the pattern in the factorial example is difficult at first.
- But, once you see the pattern, you can apply this pattern to create a recursive solution to the problem.
- Divide a problem up into:
  - What it can do (usually a base case)
  - What it cannot do
    - What it cannot do resembles original problem
    - The function launches a new copy of itself (recursion step) to solve what it cannot do.

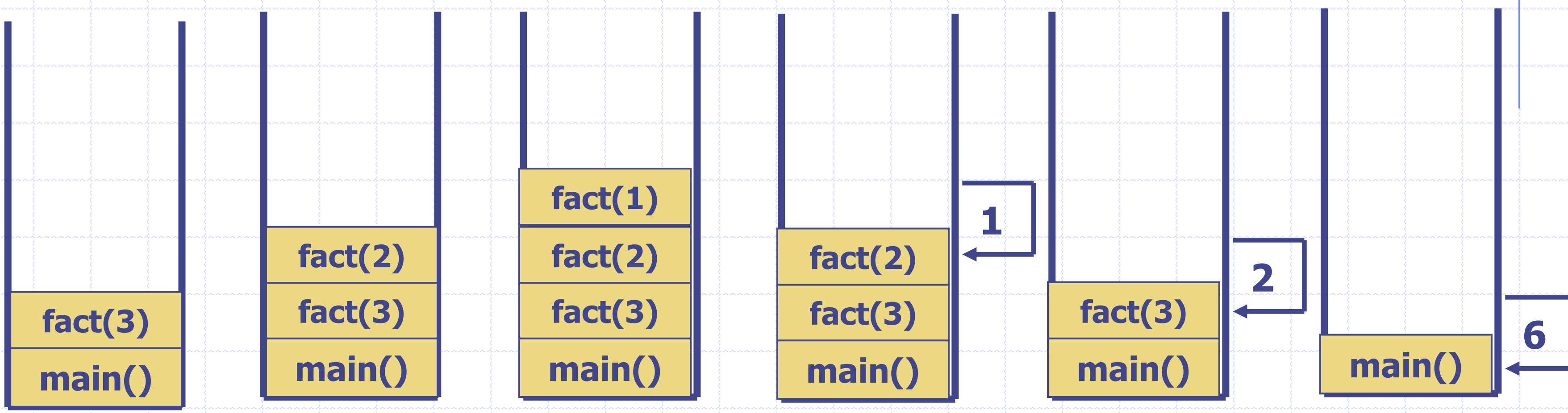
Recursive Solution

```
def findFactorialR(n)
        if n<0:
                return 0
        elif n==0 or n==1:
                return 1

        return n*findFactorialR(n-1)

print(findFactorialR(5))
```

Finding the factorial of 3



Time 2:
Push: fact(3)

Time 3:
Push: fact(2)

Time 4:
Push: fact(1)

Time 5:
Pop: fact(1)
returns 1.

Time 6:
Pop: fact(2)
returns 2.

Time 7:
Pop: fact(3)
returns 6.

Inside findFactorialR(3):

return 3 * factorial (2)

Inside findFactorialR(2):
return 2 * factorial (1)

Inside findFactorialR(1):
elif n==0 or n==1:
return 1;

Recursion vs. Iteration

- Iteration
  - Uses repetition structures (`for`, `while` or `do…while`)
  - Repetition through explicitly use of repetition structure
  - Terminates when loop-continuation condition fails
  - Controls repetition by using a counter
- Recursion
  - Uses selection structures (`if`, `if…else`)
  - Repetition through repeated method calls
  - Terminates when base case is satisfied
  - Controls repetition by dividing problem into simpler one

Recursion vs. Iteration (cont.)

- Recursion
  - More overhead than iteration
  - More memory intensive than iteration
  - Can also be solved iteratively
  - Often can be implemented with only a few lines of code

Characteristics of a Recursive Method

- Calls itself to solve a smaller problem

  Simplifies the initial problem conceptually

- Base case
  - Smallest problem to be solved
  - Result is returned to the calling method **(Terminal condition)**

- Induces overhead
  - Transfer of the control to the beginning of the method
  - Storage of all return points, intermediate arguments, return values

Recursive Binary Search

Binary search can also be a recursion

- Method calls itself with new starting and ending values
- Base case: starting value > end value

Binary Search

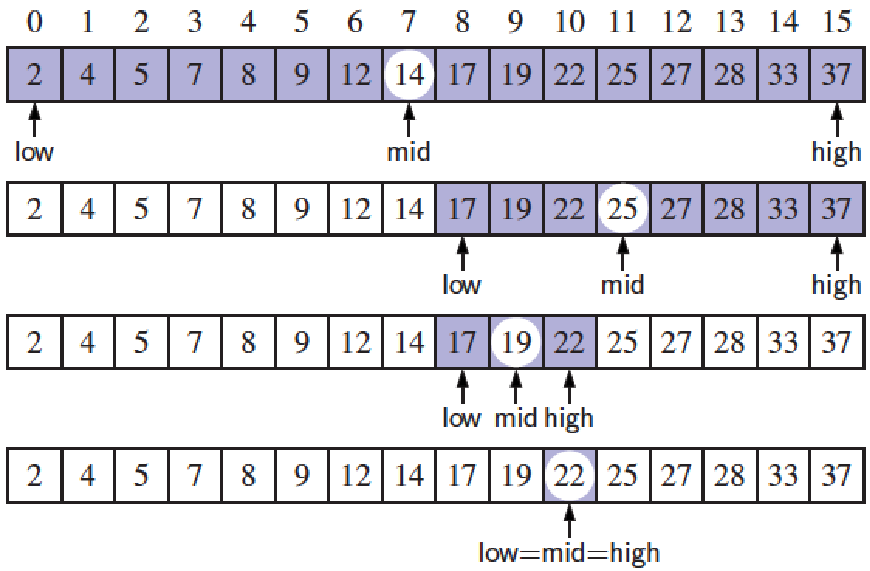
- Search for an integer, target, in an ordered list.

```python
def binary_search(data, target, low, high):
  """Return True if target is found in indicated portion of a Python list.

  The search only considers the portion from data[low] to data[high] inclusive.
  """
  if low > high:
    return False                                    # interval is empty; no match
  else:
    mid = (low + high) // 2
    if target == data[mid]:                         # found a match
      return True
    elif target < data[mid]:
      # recur on the portion left of the middle
      return binary_search(data, target, low, mid − 1)
    else:
      # recur on the portion right of the middle
      return binary_search(data, target, mid + 1, high)
```

Visualizing Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



Analyzing Binary Search

- Runs in O(log n) time.
  - The remaining portion of the list is of size high – low + 1.
  - After one comparison, this becomes one of the following:

$$(\mathsf{mid}-1)-\mathsf{low}+1=\left\lfloor\frac{\mathsf{low}+\mathsf{high}}{2}\right\rfloor-\mathsf{low}\leq\frac{\mathsf{high}-\mathsf{low}+1}{2}$$

$$\mathsf{high}-(\mathsf{mid}+1)+1=\mathsf{high}-\left\lfloor\frac{\mathsf{low}+\mathsf{high}}{2}\right\rfloor\leq\frac{\mathsf{high}-\mathsf{low}+1}{2}.$$

  - Thus, each recursive call divides the search region in half; hence, there can be at most log n levels.

Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

Example of Linear Recursion

Algorithm LinearSum(*A, n*):

Input:

A integer array *A* and an integer *n* = 1, such that *A* has at least *n* elements

Output:

The sum of the first *n* integers in *A*

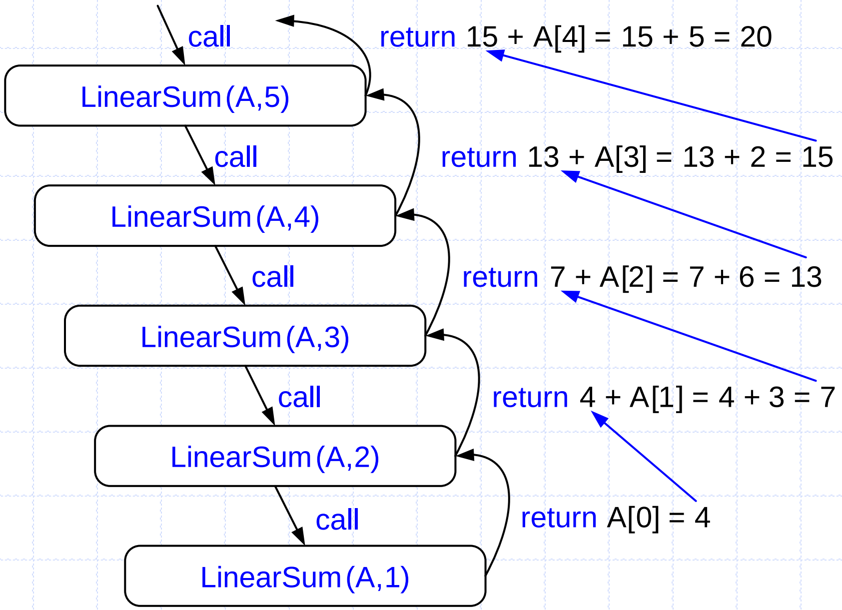
if *n* = 1 **then**

  return *A*[0]

else

  return LinearSum(*A, n* - 1) + *A*[*n* - 1]

Example recursion trace:



Reversing an Array

Algorithm ReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

if $i < j$ **then**

Swap $A[i]$ and $A[j]$

ReverseArray(A, $i + 1$, $j - 1$)

return

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional paramaters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(*A*, *i*, *j*), not ReverseArray(*A*).
- Python version:

```
def reverse(S, start, stop):
  """Reverse elements in implicit slice S[start:stop]."""
  if start < stop - 1:                          # if at least 2 elements:
    S[start], S[stop-1] = S[stop-1], S[start]   # swap first and last
    reverse(S, start+1, stop-1)                 # recur on rest
```

Computing Powers

- The power function, $p(x,n)=x^n$, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- This leads to an power function that runs in O(n) time (for we make n recursive calls).
- We can do better than this, however.

Recursive Squaring

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } x > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } x > 0 \text{ is even} \end{cases}$$

- For example,

$$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$$

$$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$$

$$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$$

$$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$$

Recursive Squaring Method

Algorithm Power(x, n):

Input: A number x and integer $n = 0$

Output: The value x^n

if $n = 0$ **then**

return 1

if n is odd **then**

$y = $ Power(x, $(n - 1)/2$)

return $x \cdot y \cdot y$

else

$y = $ Power(x, $n/2$)

return $y \cdot y$

Analysis

Algorithm Power(x, n):

Input: A number x and integer $n = 0$

Output: The value x^n

if $n = 0$ **then**

  return 1

if n is odd **then**

  $y = \text{Power}(x, (n - 1)/2)$

  return $x \cdot y \cdot y$

else

  $y = \text{Power}(x, n/2)$

  return $y \cdot y$

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we use a variable twice here rather than calling the method twice.

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

Algorithm IterativeReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

while $i < j$ **do**

Swap $A[i]$ and $A[j]$

$i = i + 1$

$j = j - 1$

return

Binary Recursion

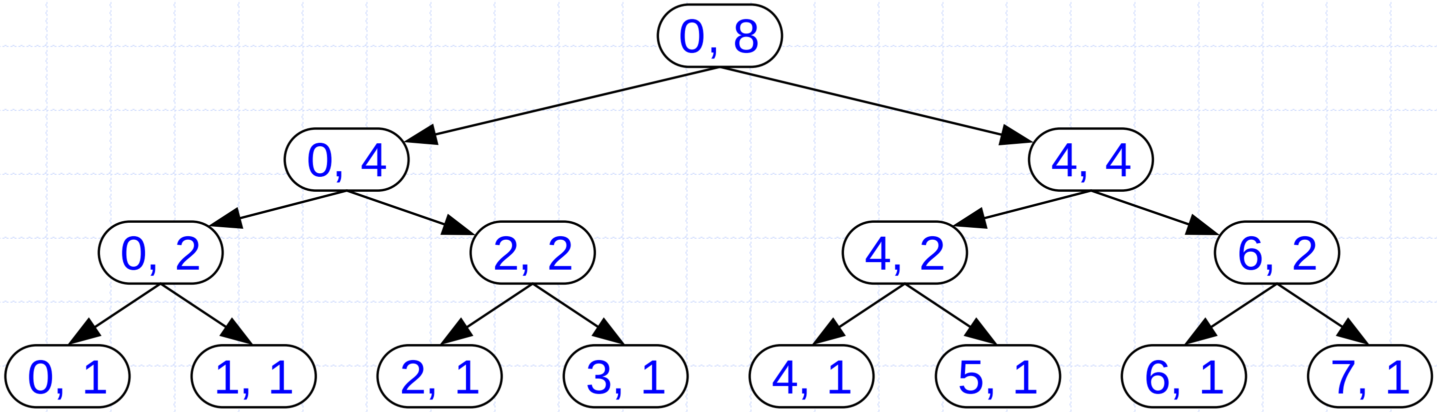
- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.

Binary Recusive Method

- Problem: add all the numbers in an integer array A:

Algorithm BinarySum(A, i, n):
    Input: An array A and integers i and n
    Output: The sum of the n integers in A starting at index i
    if $n = 1$ **then**
    return $A[i]$
    return BinarySum(A, i, $n/2$) + BinarySum(A, $i + n/2$, $n/2$)

- **Example trace:**



Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1.$$

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):

Input: Nonnegative integer k

Output: The kth Fibonacci number F_k

if $k = 0$ **then**

return k

else if k = 1 then

return k

else

return BinaryFib(k - 1) + BinaryFib(k - 2)

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

A Better Fibonacci Algorithm

- Use linear recursion instead

```
Algorithm LinearFibonacci(k):
  Input: A nonnegative integer k
  Output: Pair of Fibonacci numbers (F_k , F_{k−1})
 if k = 1 then
     return (k, 0)
 else
     (i, j) = LinearFibonacci(k − 1)
     return (i +j, i)
```

- LinearFibonacci makes $k-1$ recursive calls

Multiple Recursion

- Multiple recursion:
  - makes potentially many recursive calls
  - not just one or two

Algorithm for Multiple Recursion

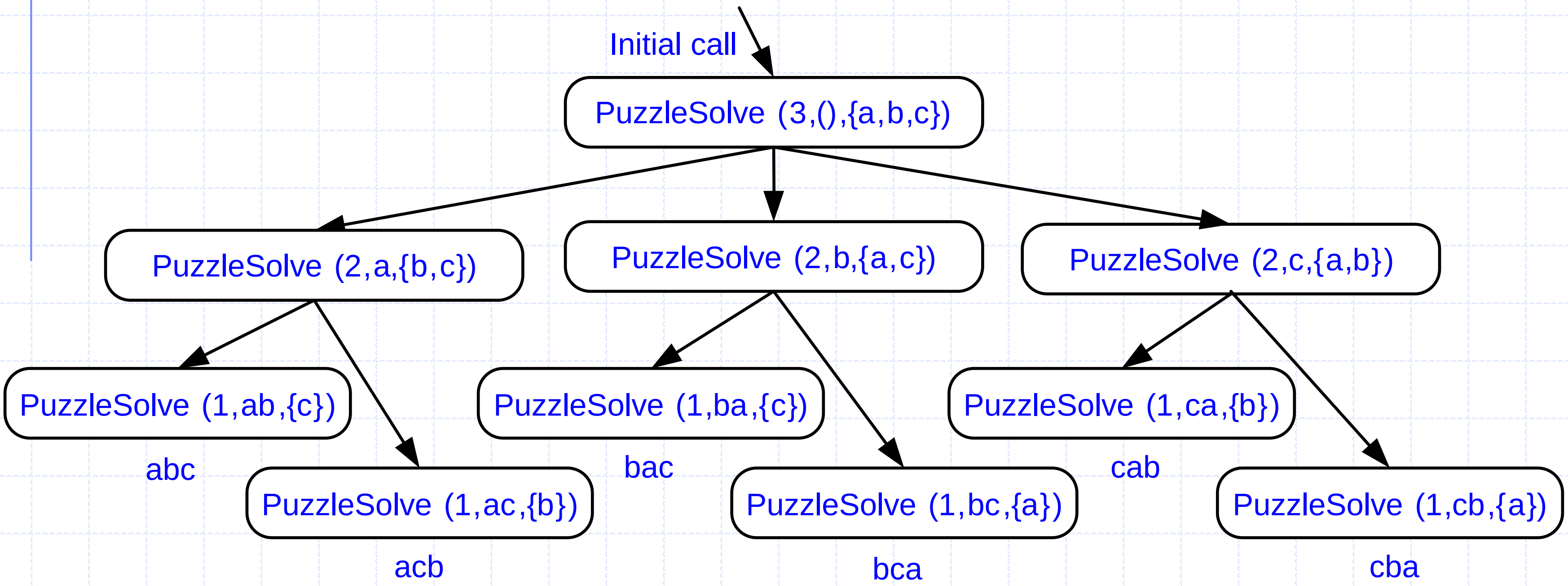
Algorithm PuzzleSolve(k,S,U):

Input: Integer k, sequence S, and set U (universe of elements to test)

Output: Enumeration of all k-length extensions to S using elements in U without repetitions

```
for all e in U do
    Remove e from U        {e is now being used}
    Add e to the end of S
    if k = 1 then
        Test whether S is a configuration that solves the puzzle
        if S solves the puzzle then
            return "Solution found: " S
    else
        PuzzleSolve(k - 1, S,U)
    Add e back to U        {e is now unused}
    Remove e from the end of S
```

Visualizing PuzzleSolve



Divide-and-Conquer

- Recursive binary search is an example of divide-and-conquer
- Idea
  - Divide the bigger problem into smaller problems
  - Solve each smaller problem separately
  - If smaller problem still too big, then solve its divisions
  - Continue process until smaller problem is a base case

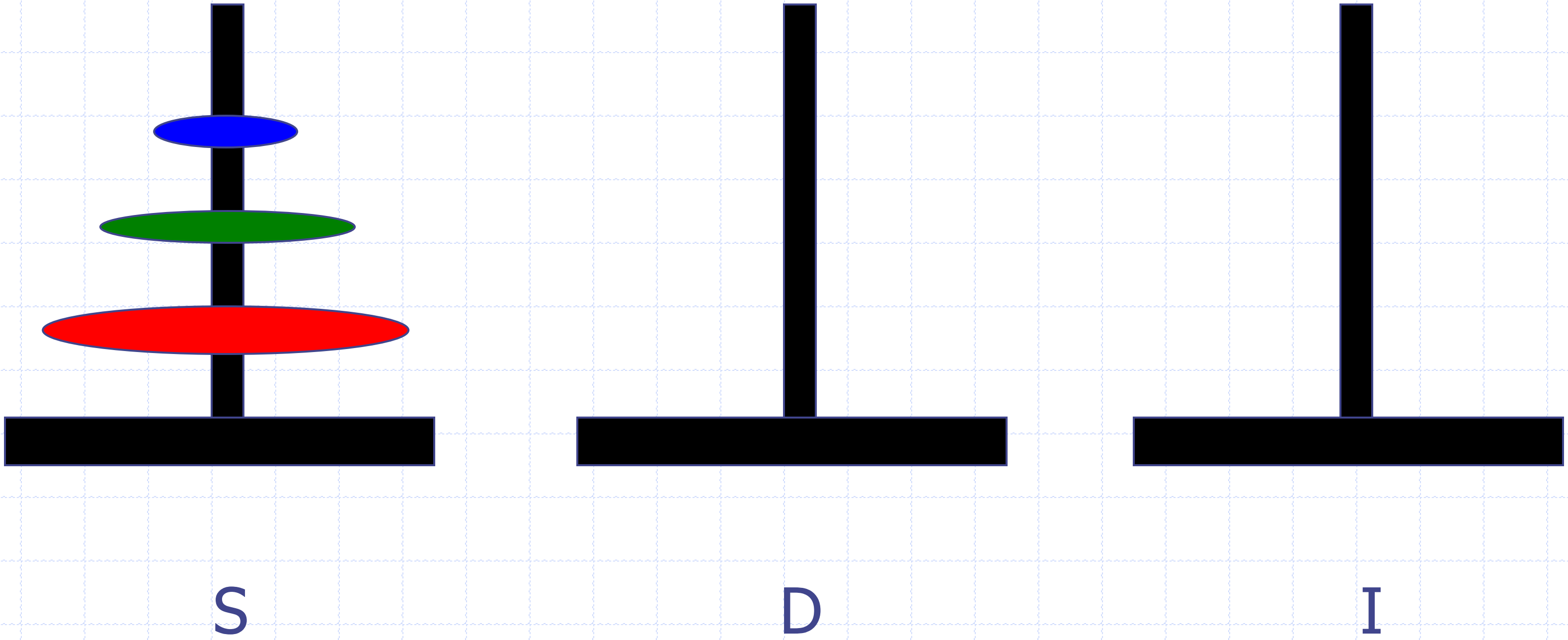
- DaC can be used with recursion as well as non recursion

Towers of Hanoi

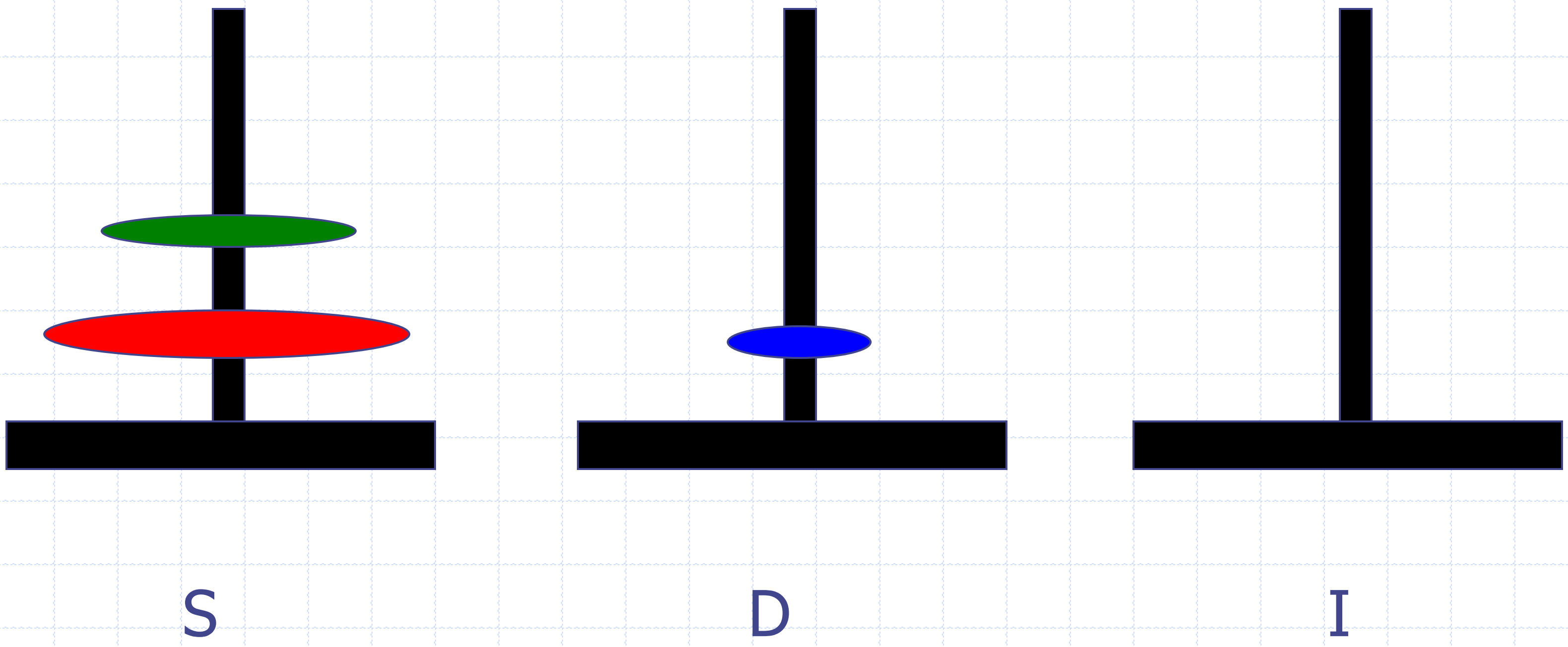
Invented by Edouard Lucas in 1883

- Three towers
- 64 gold disks (decreasing sizes) placed on the first tower
- All disks must be moved from the Source tower to the Destination Tower
- Larger disks can not be placed on top of smaller disks
- The third tower can be used to temporarily hold disks

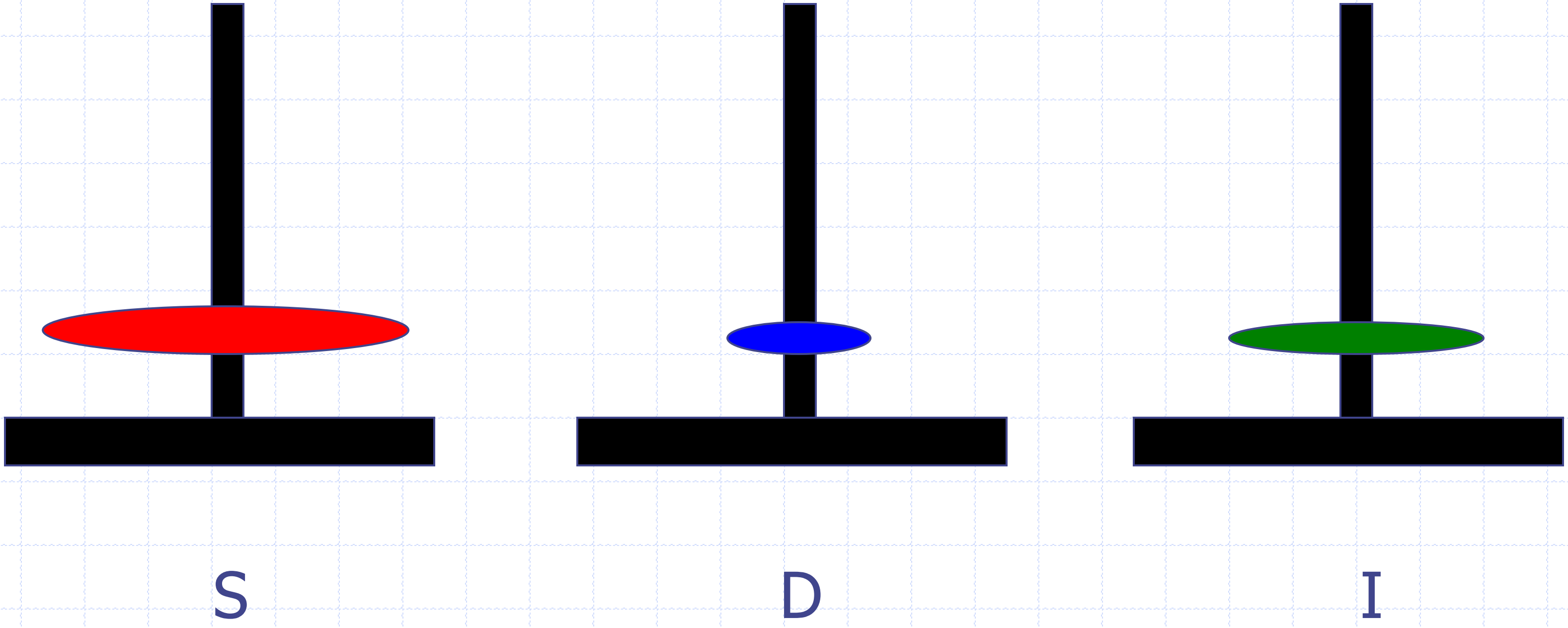
Towers of Hanoi



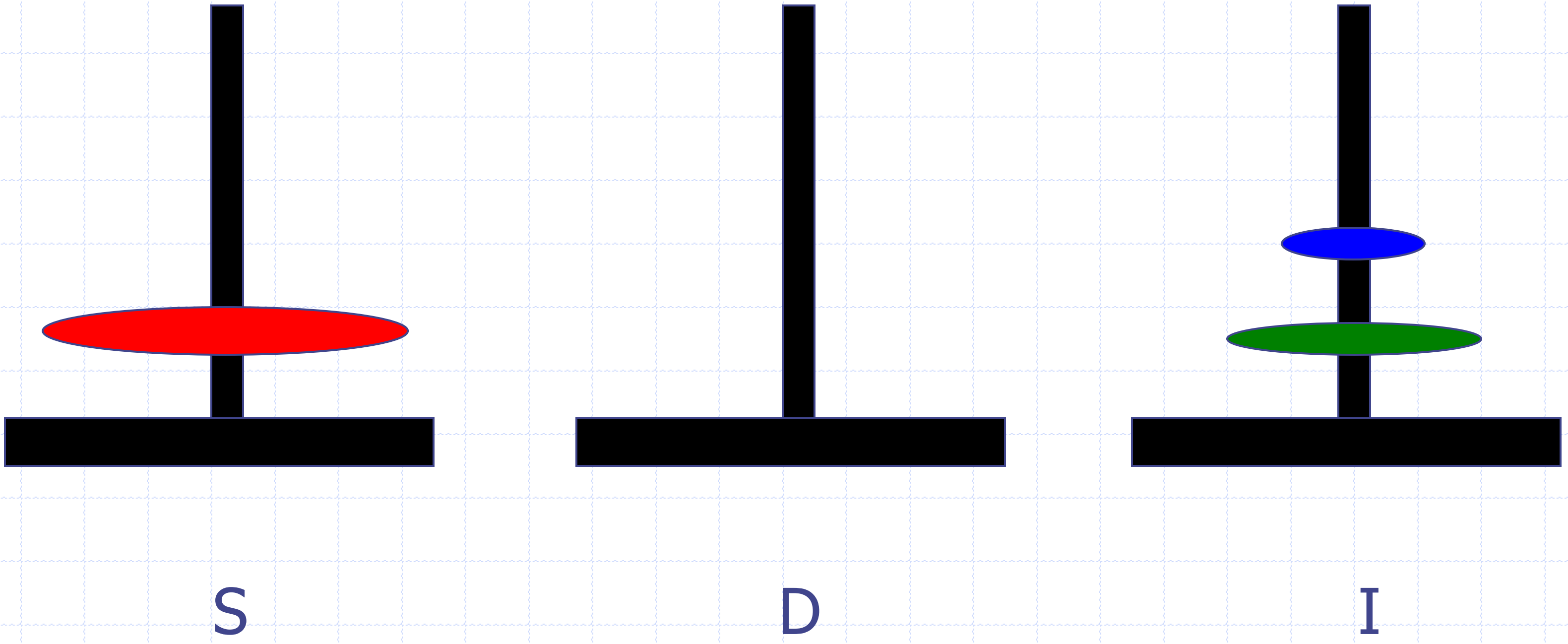
Towers of Hanoi



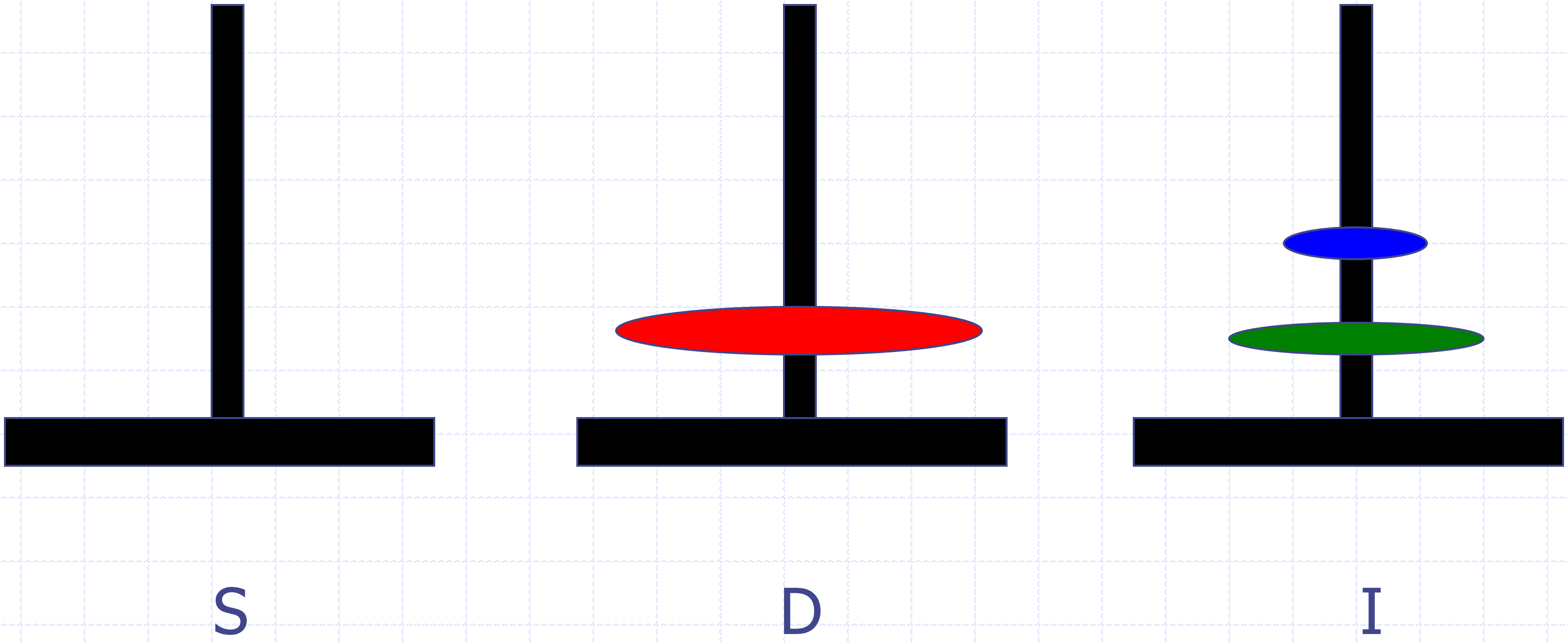
Towers of Hanoi



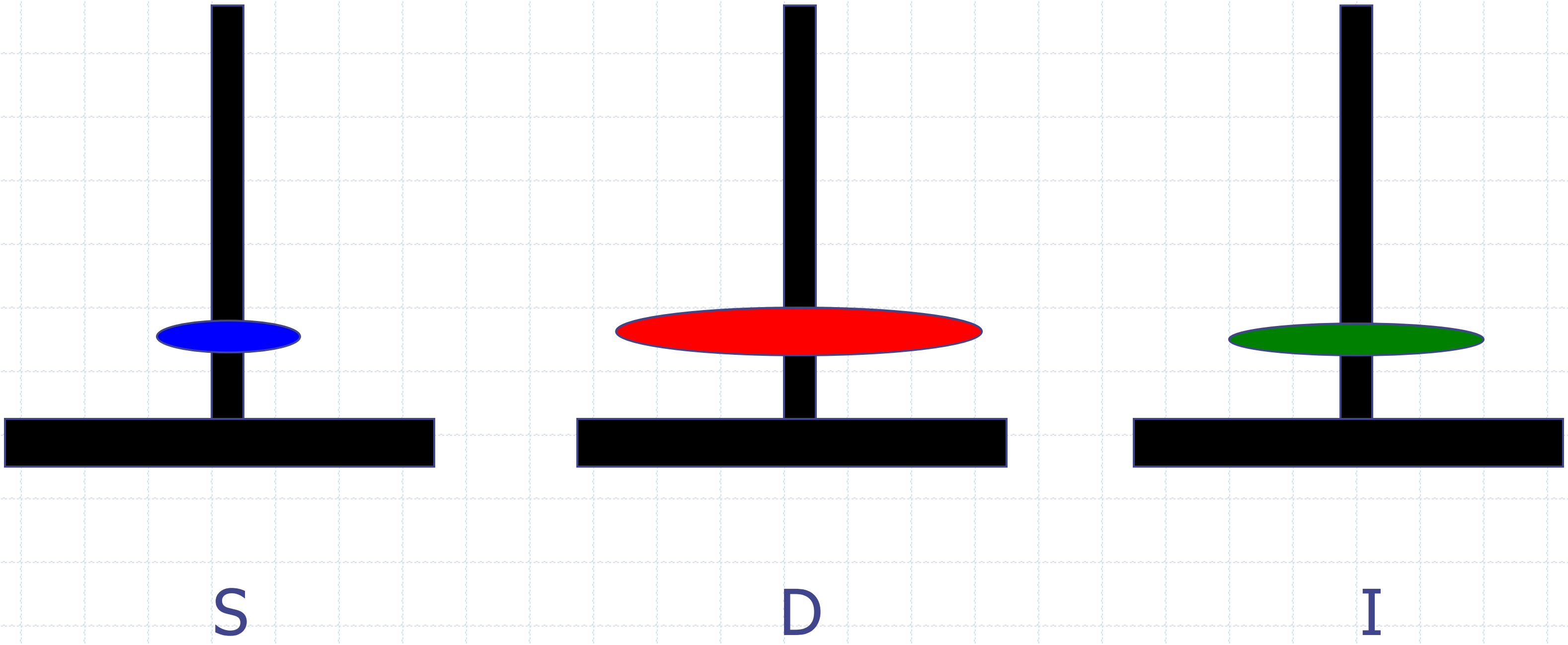
Towers of Hanoi



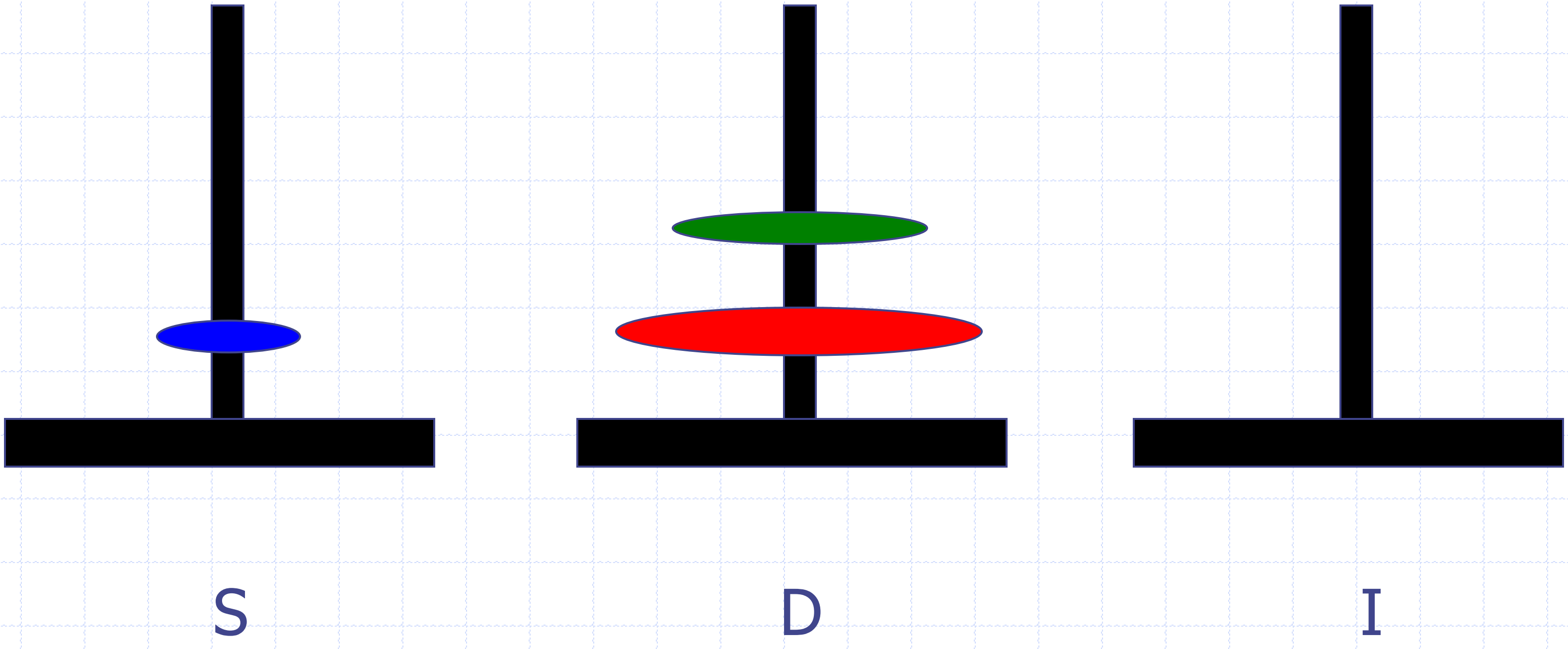
Towers of Hanoi



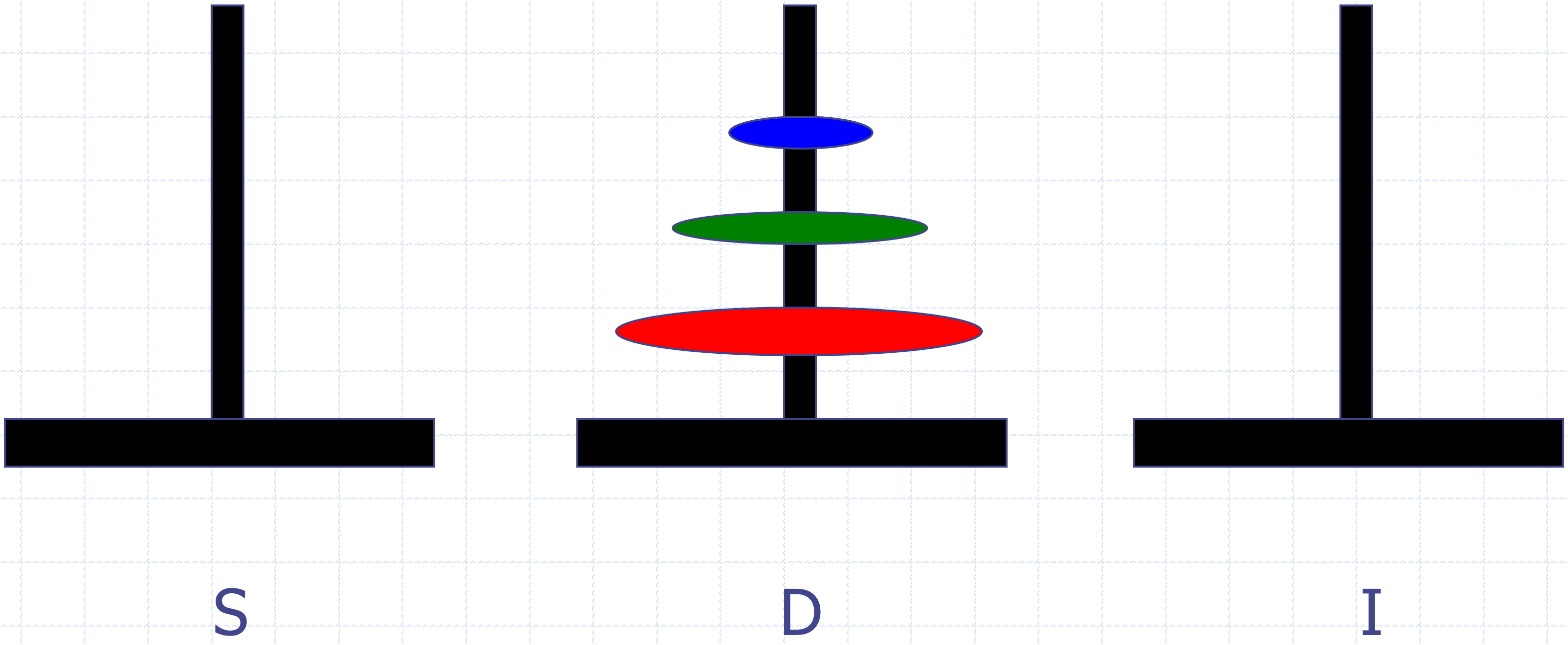
Towers of Hanoi



Towers of Hanoi



Towers of Hanoi



ToH – Recursive Solution

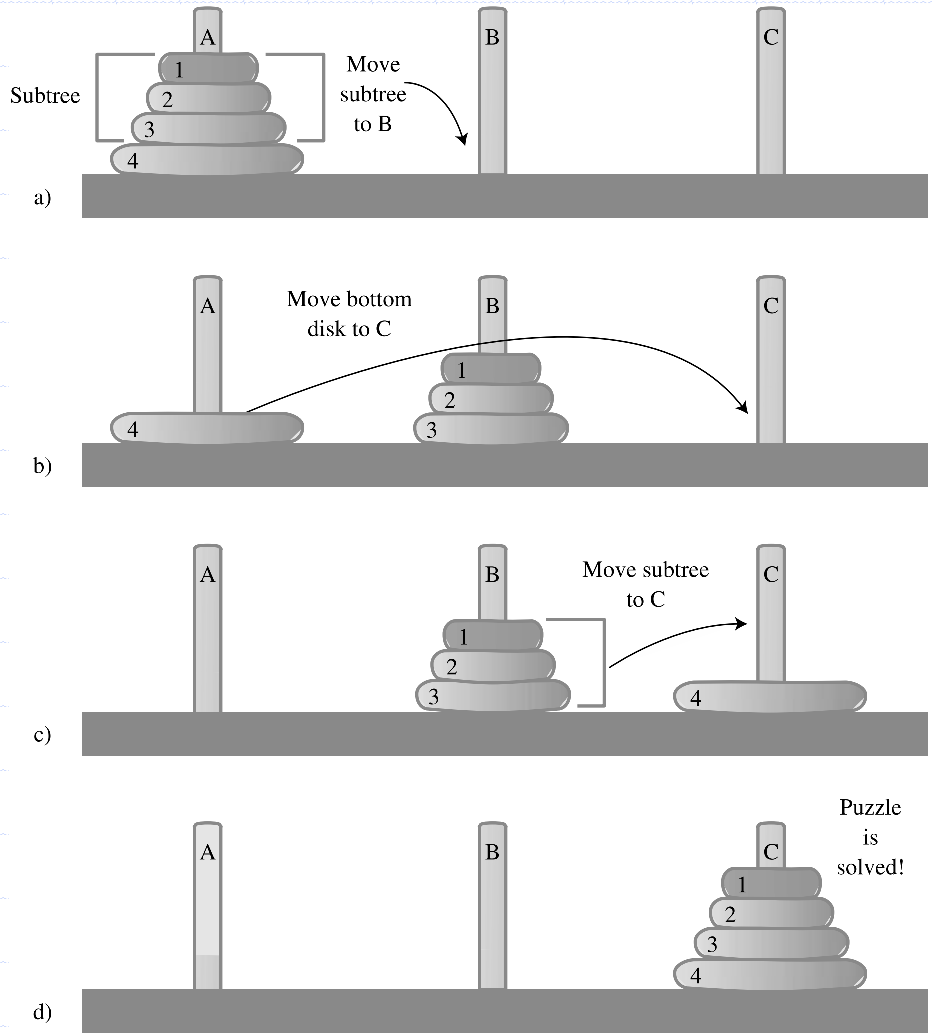
Model

- Source tower S
- Intermediate tower I
- Destination tower D

Assume n disks on S

1. Move subtree (top n-1) disks from S to I
2. Move the remaining (largest) disk from S to D
3. Move the subtree from I to D.

ToH – Recursive Solution (Moving disks from Tower A (Source) to C (Destination))



ToH – Recursive Algorithm

```
def Hanoi(topN, from, inter, to):

    if topN == 1:  // base case
      Move(1, from, to);
    else:  // recursion
      Hanoi(topN-1, from, to, inter); // from → inter
      Move(topN, from, to);
      Hanoi(topN-1, inter, from, to); // inter → to
```

Runtime of Recursive Functions

```
def recursiveFun1(n):
    if n<=0:
        return 1
    else:
        return 1+recursiveFun1(n-1)
```

Runtime of Recursive Functions

```
def recursiveFun2(n):
        if n<=0:
                return 1
        else:
                return 1+recursiveFun2(n-5)
```

Runtime of Recursive Functions

```
def recursiveFun3(n):
    if n<=0:
        return 1
    else:
        return 1+recursiveFun3(n//5)
```

Runtime of Recursive Functions

```
def recursiveFun4(n,m,o):
    if n<=0:
        print(m,",",o)
    else:
        recursiveFun4(n-1,m+1,o)
        recursiveFun4(n-1,m,o+1)
```

Runtime of Recursive Functions

```
def recursiveFun5(n):
    for i in range(0,n,2):
        print("hello")
    if n<=0:
        return 1
    else:
        return 1+recursiveFun5(n-5)
```

Divide and Conquer Master Method

$$f(n) = af(n/b) + g(n).$$

This is called a **divide-and-conquer recurrence relation.**

THEOREM 2 MASTER THEOREM Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + cn^d$$

whenever $n = b^k$, where k is a positive integer, $a \geq 1$, b is an integer greater than 1, and c and d are real numbers with c positive and d nonnegative. Then

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d, \\ O(n^d \log n) & \text{if } a = b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Complexity of Merge Sort In Example 3 we explained that the number of comparisons used by the merge sort to sort a list of n elements is less than $M(n)$, where $M(n) = 2M(n/2) + n$. By the master theorem (Theorem 2) we find that $M(n)$ is $O(n \log n)$, which agrees with the estimate found in Section 5.4.

Practice Master Method

1. $T(n) = 3T(n/2) + n^2$

2. $T(n) = 4T(n/2) + n^2$

3. $T(n) = T(n/2) + 2^n$

1. $T(n) = 3T(n/2) + n^2 \implies T(n) = \Theta(n^2)$
2. $T(n) = 4T(n/2) + n^2 \implies T(n) = \Theta(n^2 \log n)$
3. $T(n) = T(n/2) + 2^n \implies \Theta(2^n)$

Practice Master Method

5. $T(n) = 16T(n/4) + n$

6. $T(n) = 2T(n/2) + n \log n$

15. $T(n) = 3T(n/4) + n \log n$

16. $T(n) = 3T(n/3) + n/2$

5. $T(n) = 16T(n/4) + n \Longrightarrow T(n) = \Theta(n^2)$

6. $T(n) = 2T(n/2) + n \log n \Longrightarrow T(n) = n \log^2 n$

15. $T(n) = 3T(n/4) + n \log n \Longrightarrow T(n) = \Theta(n \log n)$

16. $T(n) = 3T(n/3) + n/2 \Longrightarrow T(n) = \Theta(n \log n)$

Practice Master Method (Can't apply always)

$$9.\ T(n) = 0.5T(n/2) + 1/n$$

$$4.\ T(n) = 2^n T(n/2) + n^n$$

$$9.\ T(n) = 0.5T(n/2) + 1/n \Longrightarrow \text{Does not apply } (a < 1)$$

$$4.\ T(n) = 2^n T(n/2) + n^n \Longrightarrow \text{Does not apply } (a \text{ is not constant})$$