Recursion

The Pattern

- : when a method calls itself
- Classic example--the factorial function:
  - n! = 1 · 2 · 3 · · · · · (n-1) · n
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \textit{else} \end{cases}$$

- As a Python method:

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n−1)
```

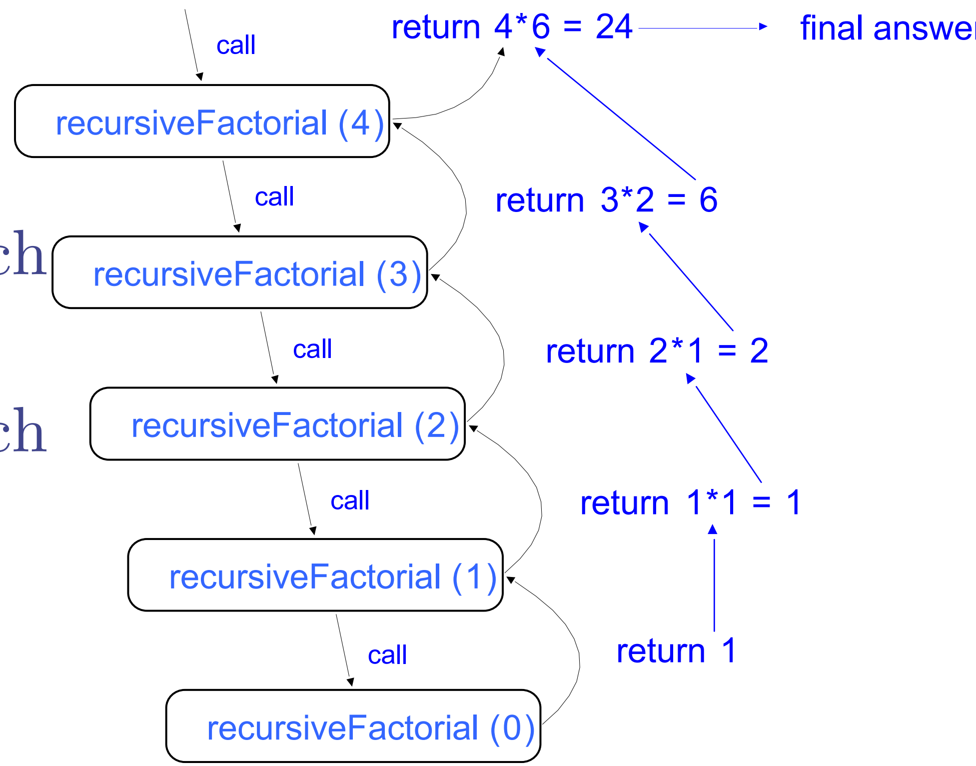
Content of a Recursive Method

- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Visualizing

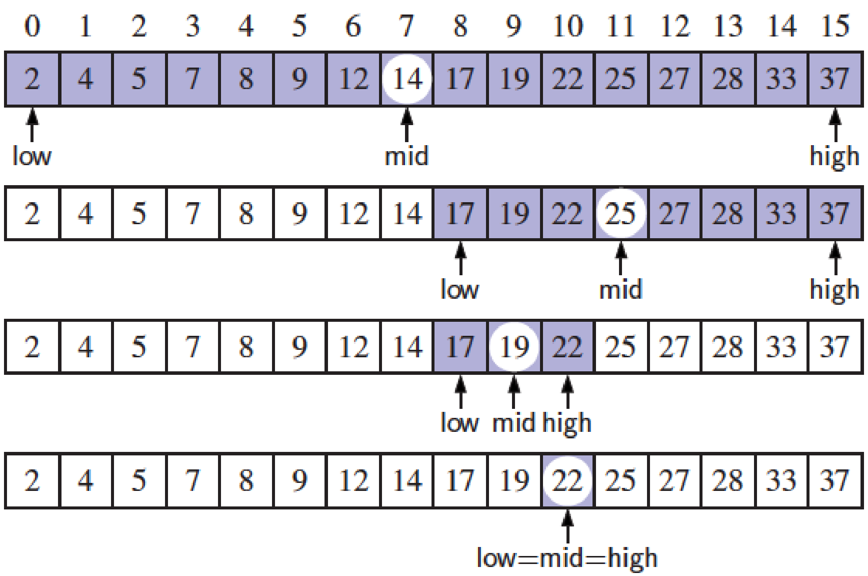
- trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value

- Example



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.



Example of Linear

Algorithm LinearSum(A, n):

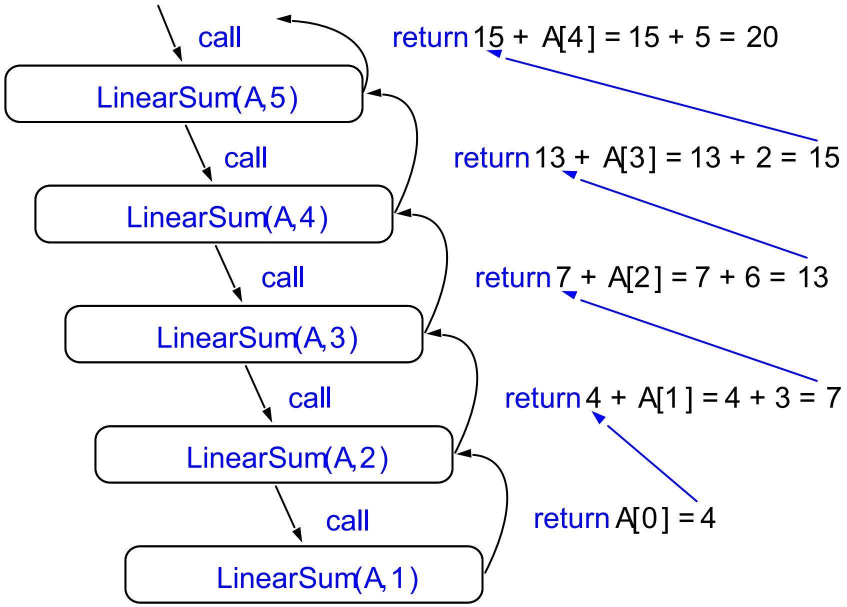
Input:

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

Example 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

Defining Arguments for

- In creating recursive methods, it is important to define the methods in ways that facilitate .
- 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:

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

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 $y \cdot y$

Tail Recursion

- Tail 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

Another 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

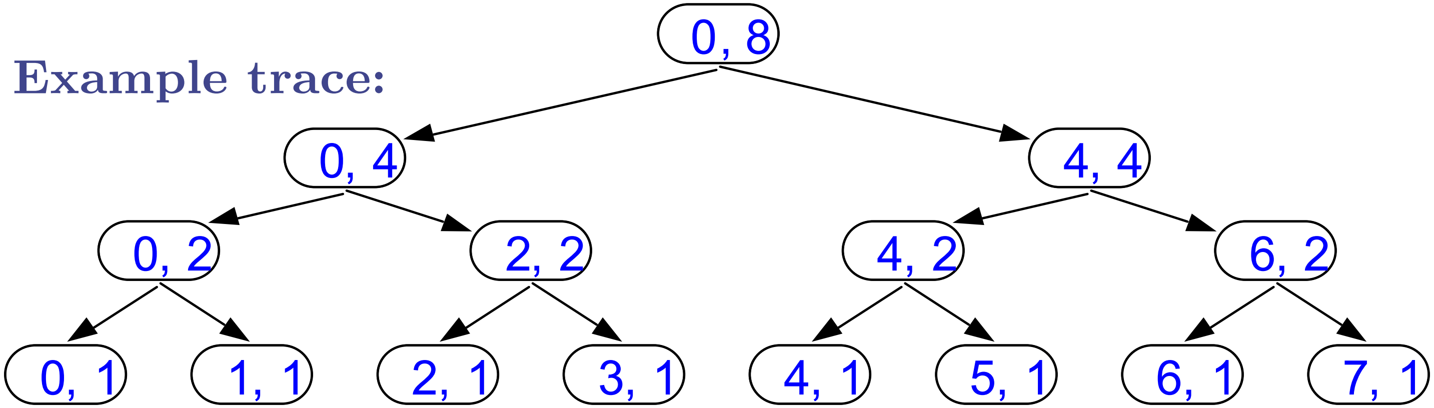
Output: The sum of the n integers in A starting at index i

if $n = 1$ **then**

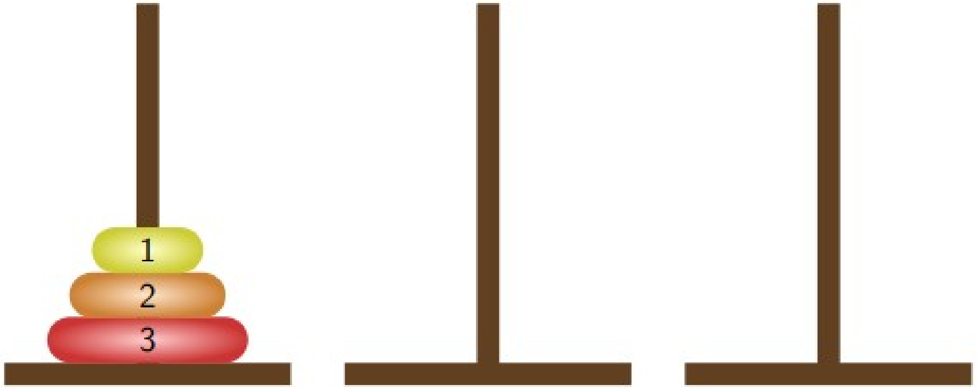
return $A[i\,]$

return BinarySum(A, i, $n/\,2$) + BinarySum(A, $i + n/\,2$, $n/\,2$)

- **Example trace:**



Tower of Hanoi



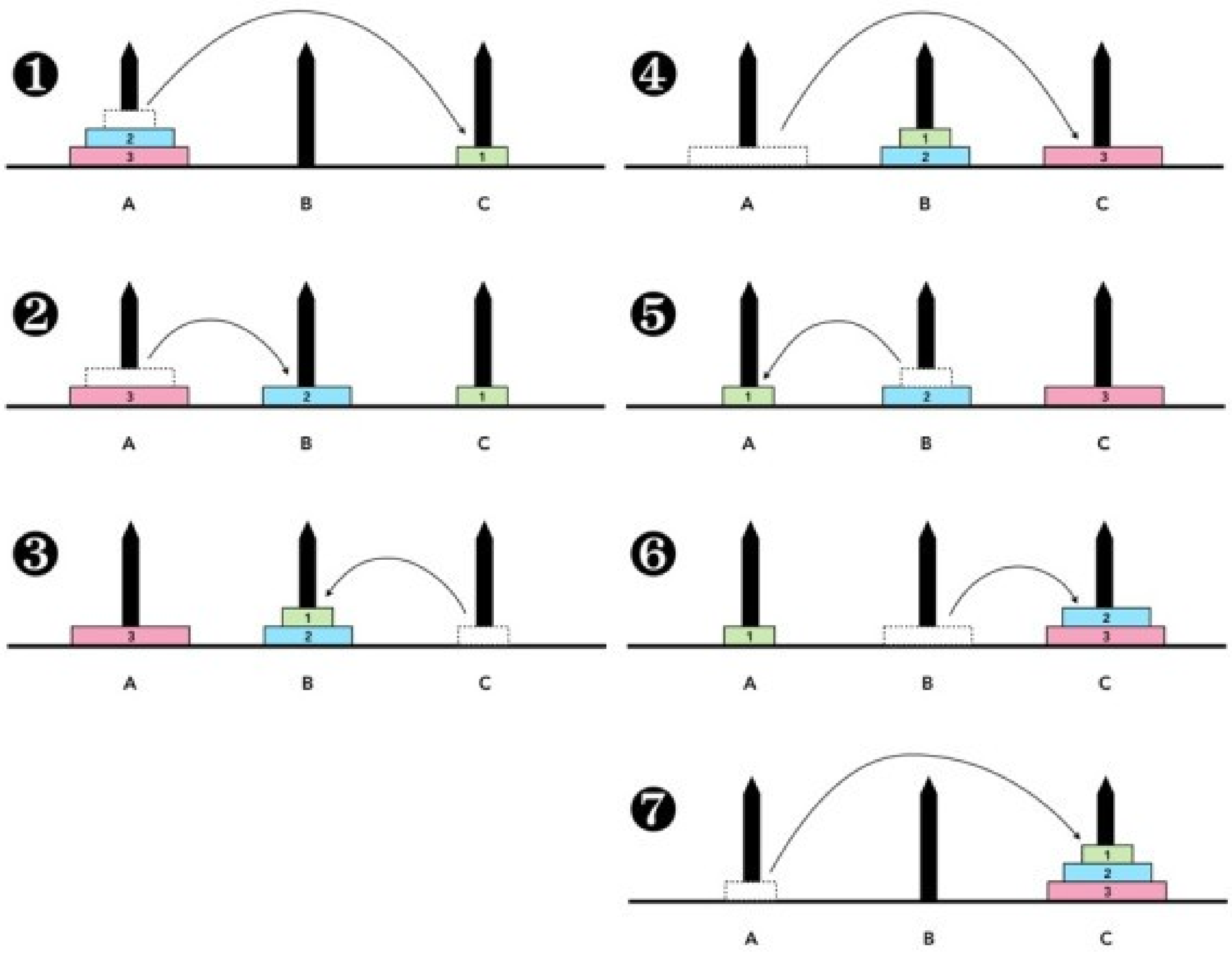
Procedure for moving a tower of n disks from a peg A onto a peg C, with B serving as an auxiliary peg:

1) If $n > 1$, then first use this procedure to move the $n - 1$ smaller disks from peg A to peg B.

2) Now the largest disk, i.e. disk n can be moved from peg A to peg C.

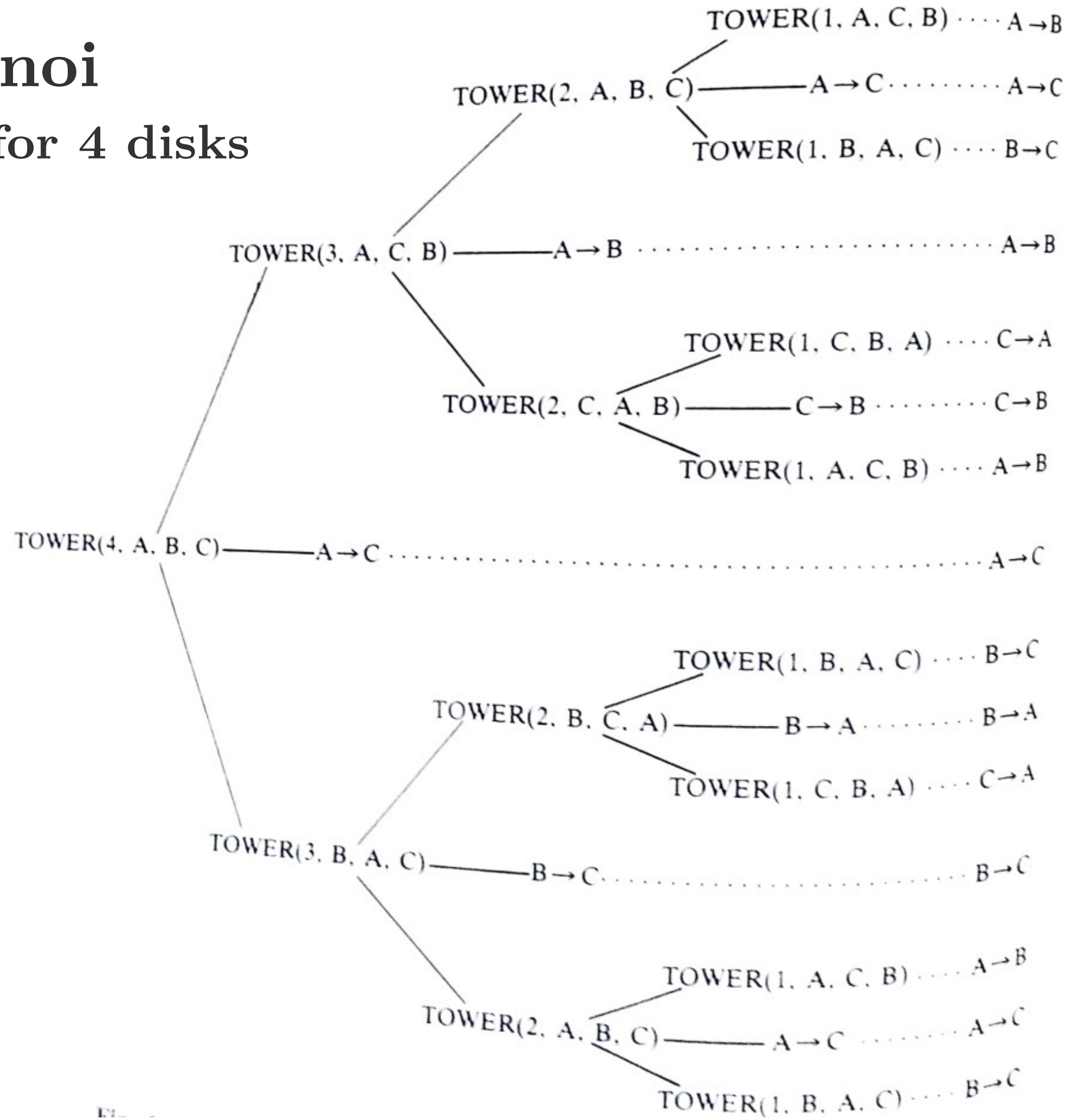
3) If $n > 1$, then again use this procedure to move the $n - 1$ smaller disks from peg B to peg C.

Tower of Hanoi for 3 disks



Tower of Hanoi

Recursion trace for 4 disks



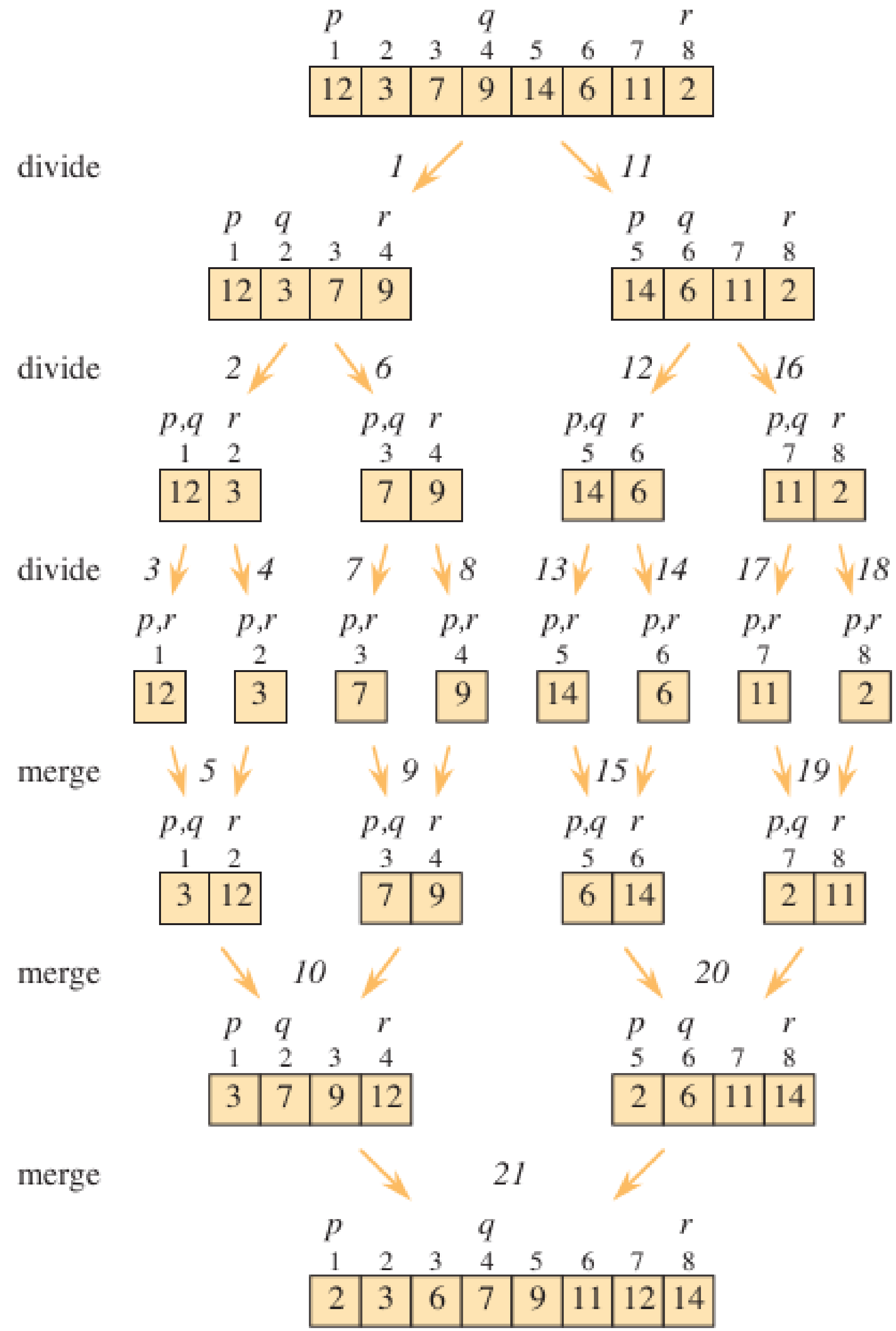
Divide & Conquer methods

- **Divide** the problem into one or more subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the subproblem solutions to form a solution to the original problem.

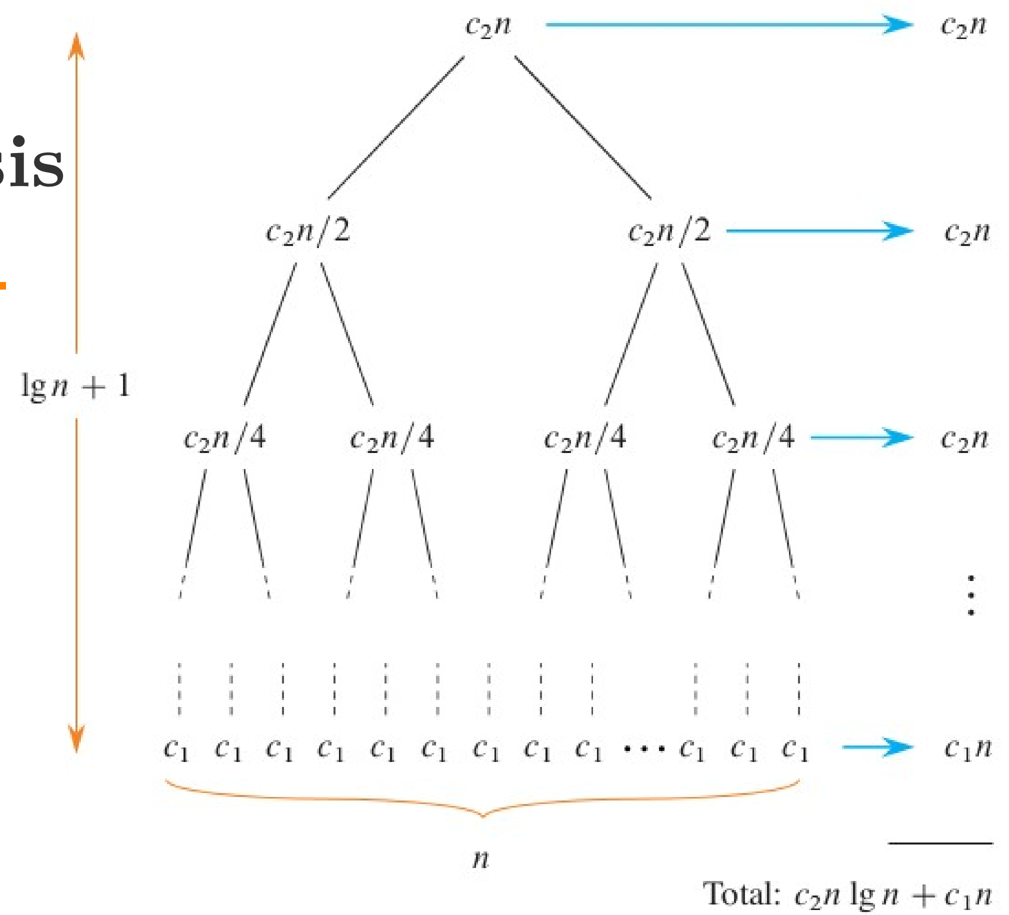
Merge Sort Analysis

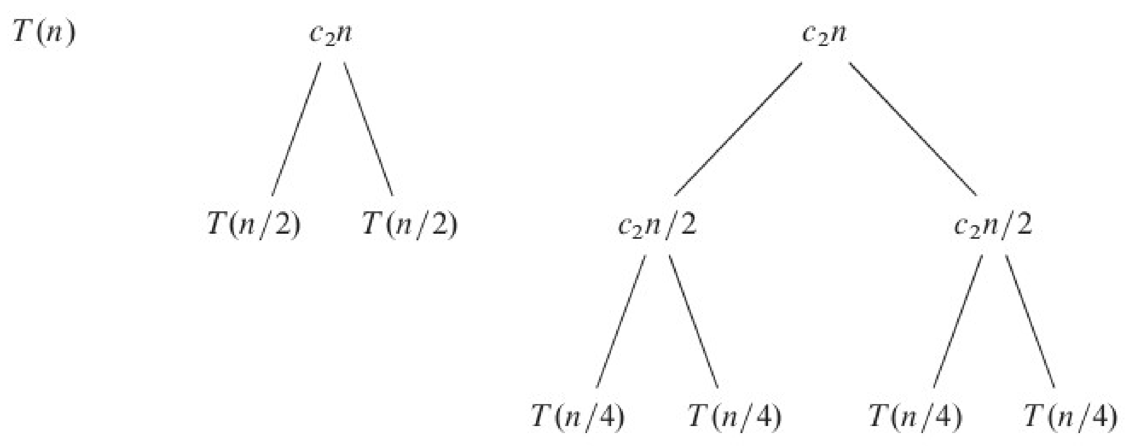
$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 , \\ D(n) + aT(n/b) + C(n) & \text{otherwise} . \end{cases}$$

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 , \\ 2T(n/2) + c_2 n & \text{if } n > 1 , \end{cases}$$



Merge Sort Analysis





Iterative Substitution

In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$T(n) = 2T(n/2) + c_2 n$$

$$2(2T(n/2^2)) + c_2(n/2)) + c_2 n$$

$$T(n) = 2^2 T(n/2^2) + 2c_2 n$$

$$T(n) = 2^3 T(n/2^3) + 3c_2 n$$

$$T(n) = 2^4 T(n/2^4) + 4c_2 n$$

$$\ldots$$

$$T(n) = 2^i T(n/2^i) + ic_2 n$$

Note that base, $T(1)=c_1$, case occurs when $2^i=n$. That is, $i = \log n$.

So,

$$T(n) = c_1 n + c_2 n \log n$$

Thus, $T(n)$ is $O(n \log n)$.

The Master Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Master Method, Example 1

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Example: $T(n) = 4T(n/2) + n$

Solution: $\log_b a = 2$, so case 1 says $T(n)$ is $O(n^2)$.

Master Method, Example 2

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Example: $T(n) = 2T(n/2) + n \log n$

Solution: $\log_b a=1$, so case 2 says T(n) is $O(n \log^2 n)$.

Master Method, Example 3

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Example: $T(n) = T(n/3) + n \log n$

Solution: $\log_b a = 0$, so case 3 says T(n) is O(n log n).

Master Method, Example 4

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Example: $$T(n) = 8T(n/2) + n^2$$

Solution: $\log_b a = 3$, so case 1 says T(n) is $O(n^3)$.

Master Method, Example 5

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Example: $$T(n) = 9T(n/3) + n^3$$

Solution: $\log_b a=2$, so case 3 says T(n) is O(n^3).

Master Method, Example 6

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ∎

Example: $T(n) = T(n/2) + 1$ (binary search)

Solution: $\log_b a = 0$, so case 2 says T(n) is O(log n).

Master Method, Example 7

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Example: $T(n) = 2T(n/2) + \log n$ (heap construction)

Solution: $\log_b a = 1$, so case 1 says T(n) is O(n).

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 $k <= 1$ **then**

  return k

else

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

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1 \quad n_1 = 1$
  - $n_2 = n_1 + n_0 = 2$
  - $n_3 = n_2 + n_1 = 3$
  - $n_4 = n_3 + n_2 = 5$
  - $n_5 = n_4 + n_3 = 8$
  - $n_6 = n_5 + n_4 = 13$
  - $n_7 = n_6 + n_5 = 21$
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

Integer Multiplication

Algorithm: Multiply two n-digit integers I and J.

Divide step: Split I and J into high-order and low-order halves

$$I = I_h r^{n/2} + I_l$$
$$J = J_h r^{n/2} + J_l$$

We can then define I*J by multiplying the parts and adding:

$$I * J = \left(I_h r^{n/2} + I_l \right) * \left(J_h r^{n/2} + J_l \right)$$
$$I_h J_h r^n + I_h J_l r^{n/2} + I_l J_h r^{n/2} + I_l J_l$$

So, $T(n) = 4T(n/2) + O(n)$, which implies $T(n)$ is $O(n^2)$.

But that is no better than the algorithm we learned in grade school.

An Improved Integer Multiplication Algorithm: Karatsuba Algorithm

Algorithm: Multiply two n-digit integers I and J.

Divide step: Split I and J into high-order and low-order halves

$$I = I_h 2^{n/2} + I_l$$

$$J = J_h 2^{n/2} + J_l$$

Observe that there is a different way to multiply parts:

$$a = I_h J_h$$

$$d = I_l J_l$$

$$e = \left(I_h J_l + I_l J_h\right) - a - d$$

$$I * J = ar^n + er^{n/2} + d$$

So, $T(n) = 3T(n/2) + O(n)$, which implies $T(n)$ is $O(n^{\log_2 3}) = O(n^{1.584})$, by the Master Theorem.

References

Data Structures and Algorithms in Python
Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

Introduction to Algorithms
Leiserson,Stein,Rivest,Cormen

Algorithms, 4th Edition
Robert Sedgewick and Kevin Wayne

Few Images from the internet

A Better Fibonacci Algorithm

- Use linear 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