CSI 2103: Data Structures

Recursion (Ch 4)

Yonsei University

Spring 2022

Seong Jae Hwang

Aims

- Various operations operate repeatedly on data structures
  - Loops
- Discuss recursion
- Examples
- Time complexity

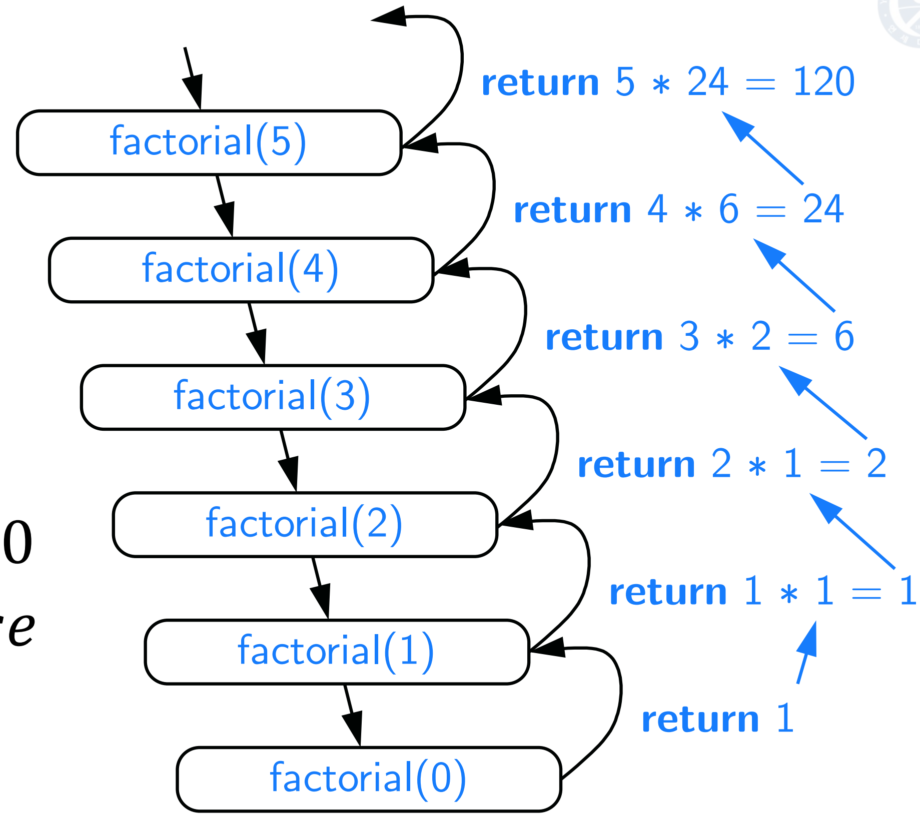
Recursion

- A recursive method which calls itself recursively
  - Base case: when to stop the recursion
  - Recursive calls: Calls itself to reach the base case
- Factorial $f(n) = n!$:

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

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

Factorial



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

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

Recursion

- A recursive method which calls itself recursively
- Factorial $f(n) = n!$:

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

- Fibonacci sequence:

$$f(n) = \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f(n-1) + f(n-2) & else \end{cases}$$

Binary Search

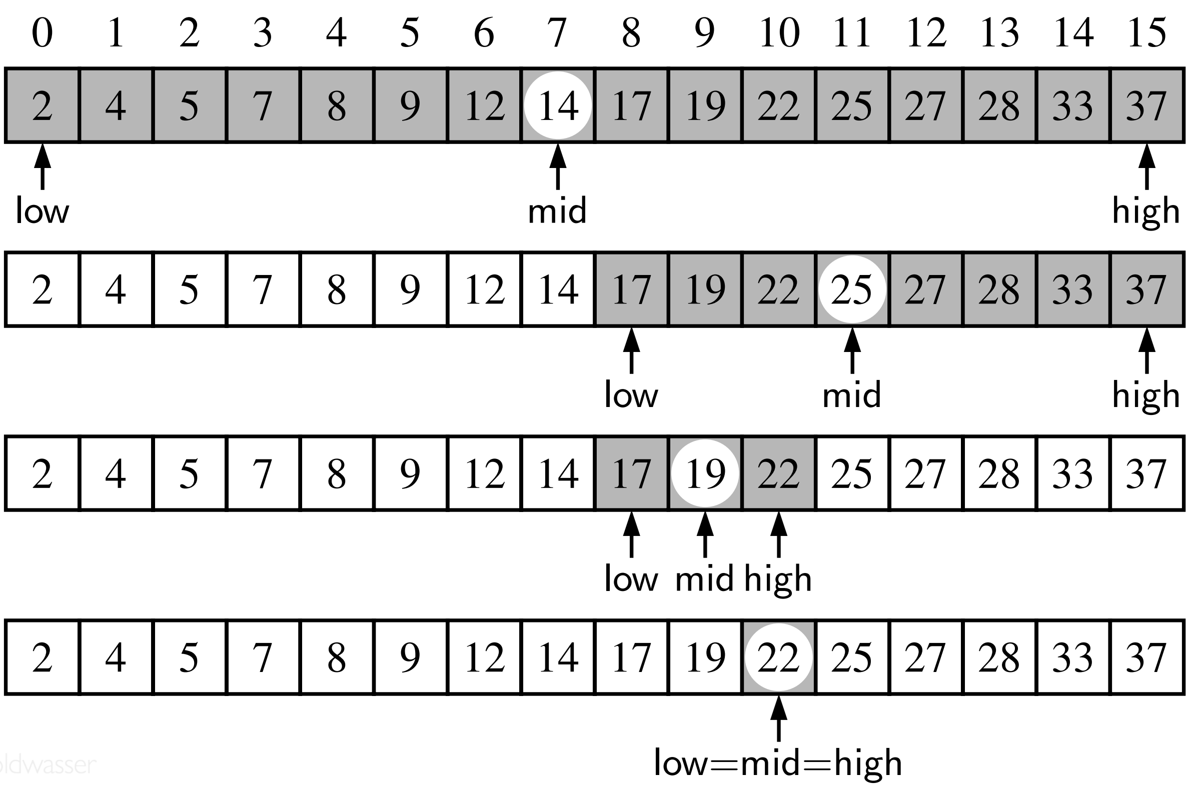
- Search for an integer in a sorted list
  - If unsorted, we need the sequential search $O(n)$
- When the sequence is sorted and indexable
  - $i < j \Rightarrow A[i] \leq A[j]$
  - Which also means for any index i,
    - All values at indices $0, \ldots, i-1 \leq$ the value at i
    - All values at indices $i+1, \ldots, n-1 \geq$ the value at i

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Binary Search

$$\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor$$

- Three cases:
  - If target = data[mid], then we have found the target
  - If target < data[mid], then we recur on the first half (low to mid-1)
  - If target > data[mid], then we recur on the second half (mid+1 to high)



Binary Search

$$\text{mid} = \lfloor(\text{low} + \text{high})/2\rfloor$$

- Three cases:
  - If target = data[mid], then we have found the target
  - If target < data[mid], then we recur on the first half (low to mid-1)
  - If target > data[mid], then we recur on the second half (mid+1 to high)

```
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)
```

Binary Search

$$\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor$$

- Why this complicated way of searching?
  - Because it's super fast!
- Each recursive call divides the search space in half

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

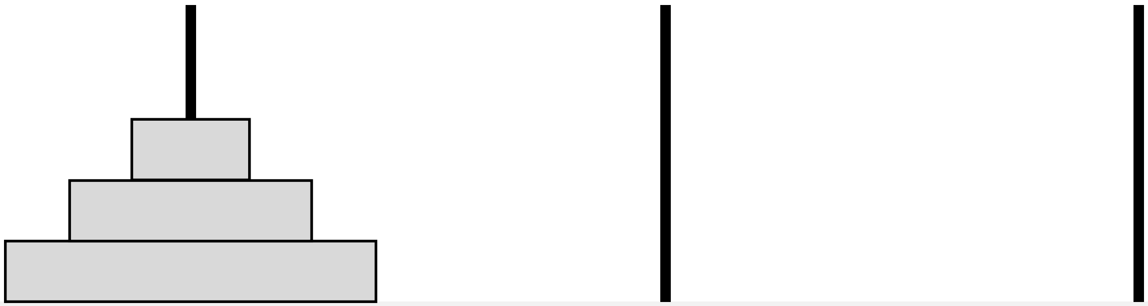
or

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

  - The maximum number of recursive calls is the smallest integer r such that $\frac{n}{2^r} < 1$ so $r = \lfloor \log n \rfloor + 1$
- Time complexity of binary search: $O(\log n)$ (base 2)
  - If $n = 1{,}000{,}000{,}000$, then $\log n = 30$

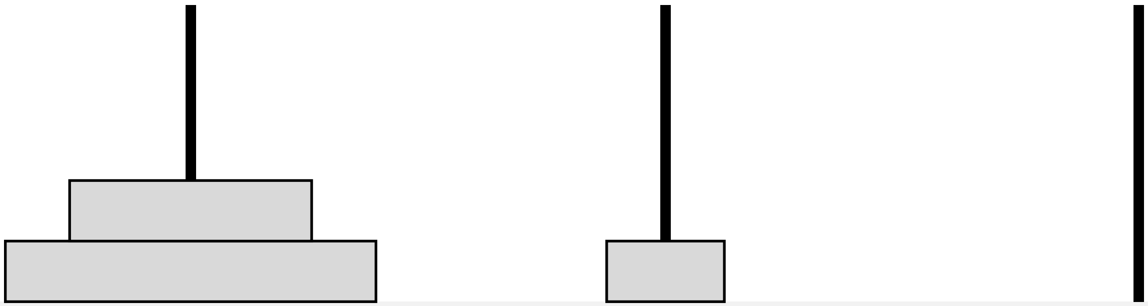
Tower of Hanoi

- *n* discs, 3 rods
  - Move one disc at a time to move all *n* discs to the second rod
  - No disc can be placed on a smaller disc



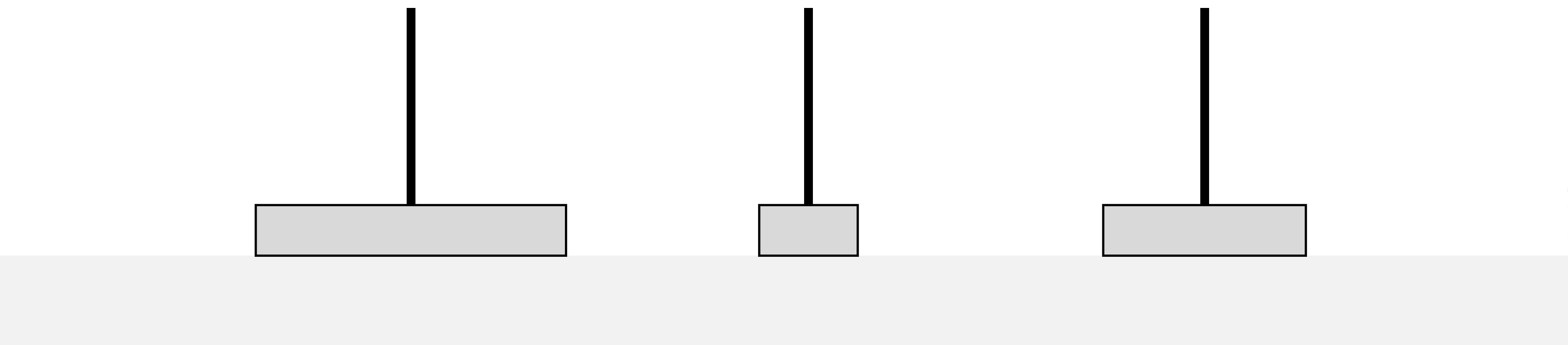
Tower of Hanoi

- *n* discs, 3 rods
  - Move one disc at a time to move all *n* discs to the second rod
  - No disc can be placed on a smaller disc



Tower of Hanoi

- *n* discs, 3 rods
    - Move one disc at a time to move all *n* discs to the second rod
    - No disc can be placed on a smaller disc

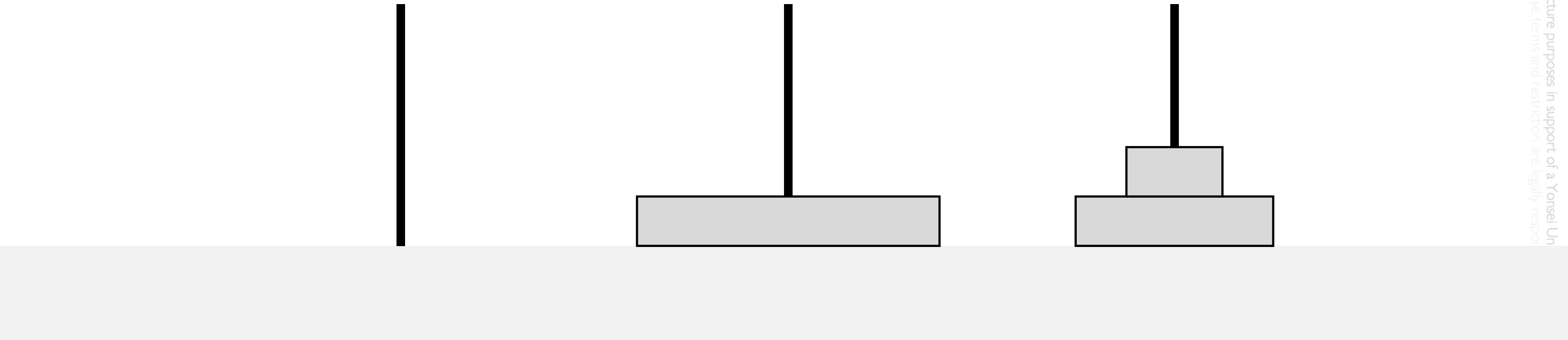


Tower of Hanoi

- *n* discs, 3 rods
  - Move one disc at a time to move all *n* discs to the second rod
  - No disc can be placed on a smaller disc

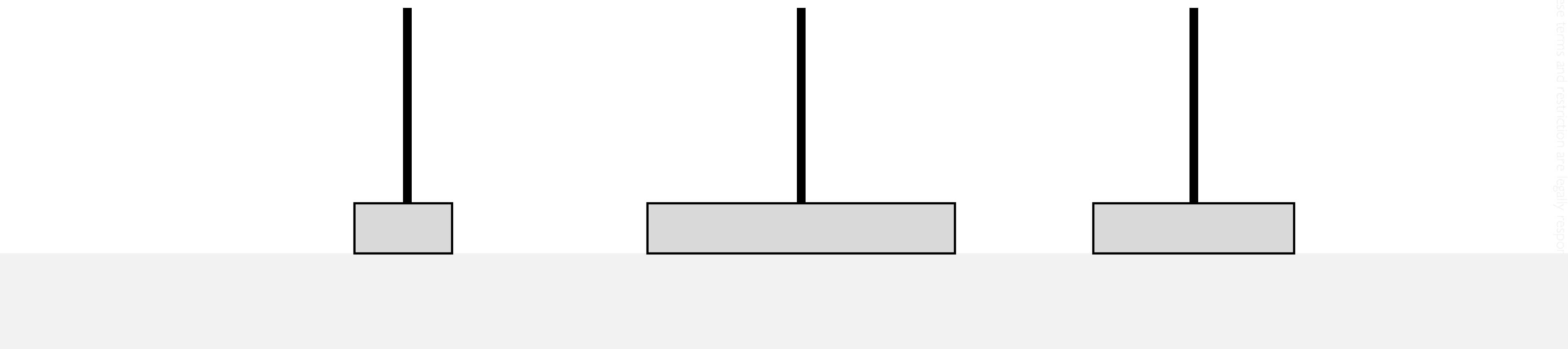
Tower of Hanoi

- n discs, 3 rods
  - Move one disc at a time to move all n discs to the second rod
  - No disc can be placed on a smaller disc



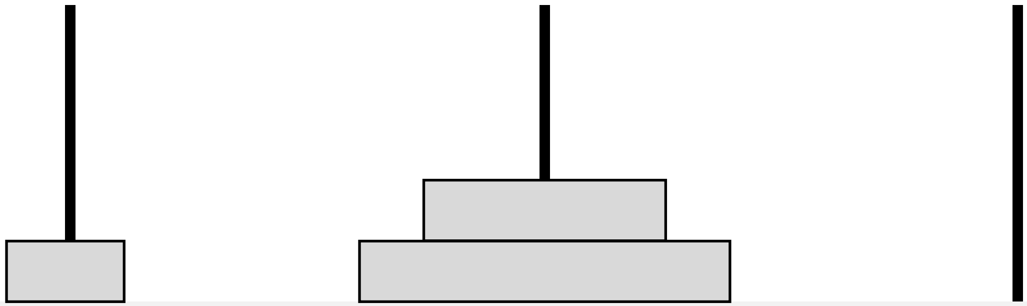
Tower of Hanoi

- *n* discs, 3 rods
  - Move one disc at a time to move all *n* discs to the second rod
  - No disc can be placed on a smaller disc



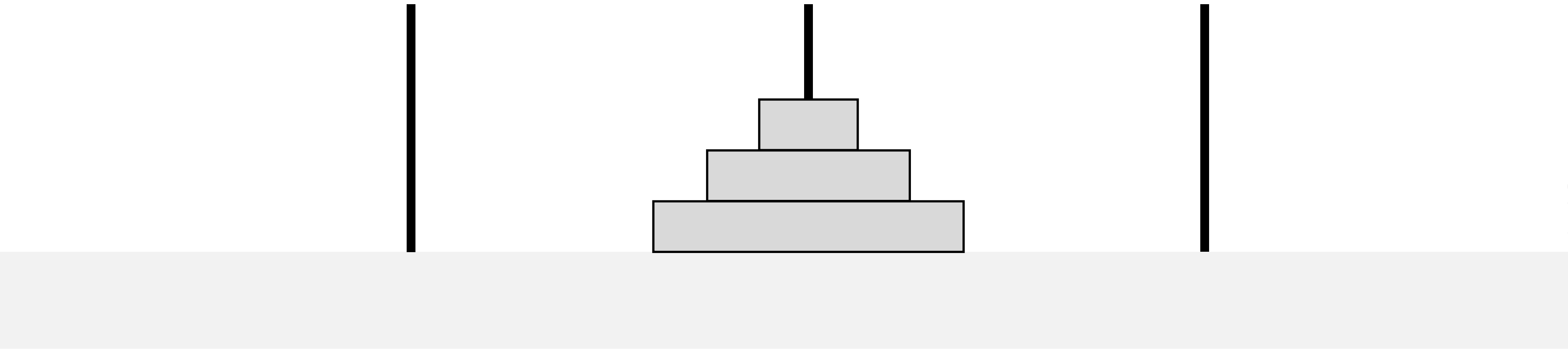
Tower of Hanoi

- *n* discs, 3 rods
  - Move one disc at a time to move all *n* discs to the second rod
  - No disc can be placed on a smaller disc



Tower of Hanoi

- *n* discs, 3 rods
  - Move one disc at a time to move all *n* discs to the second rod
  - No disc can be placed on a smaller disc



Tower of Hanoi

- Move 3 disks from rod 1 to rod 2
  - Base case?
  - To move n disks from 1 to 2, we need to move n-1 disks from 1 to 3, move the bottom-most disk from 1 to 2, then move the n-1 disks from 3 to 2
- Time complexity? $O(2^n)$

```
def Hanoi(n, loc_from, loc_to):
    if n <= 0:
        return
    loc_aux = 6 - loc_from - loc_to
    Hanoi(n-1, loc_from, loc_aux)
    print("Move a disk from rod " + str(loc_from) + " to rod " + str(loc_to))
    Hanoi(n-1, loc_aux, loc_to)
```

```
Hanoi(3, 1, 2)
```

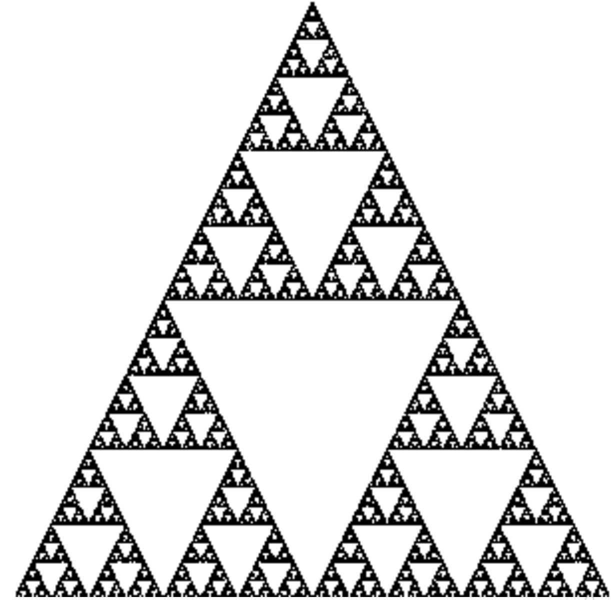
```
Move a disk from rod 1 to rod 2
Move a disk from rod 1 to rod 3
Move a disk from rod 2 to rod 3
Move a disk from rod 1 to rod 2
Move a disk from rod 3 to rod 1
Move a disk from rod 3 to rod 2
Move a disk from rod 1 to rod 2
```

Tower of Hanoi

- Wikipedia illustration: https://upload.wikimedia.org/wikipedia/commons/2/20/Tower_of_Hanoi_recursion_SMIL.svg
- Game: https://www.mathsisfun.com/games/towerofhanoi.html

Fractal

- Endlessly recursive pattern
  - ex: Sierpinski triangle
  - triangles inside triangles inside triangles…

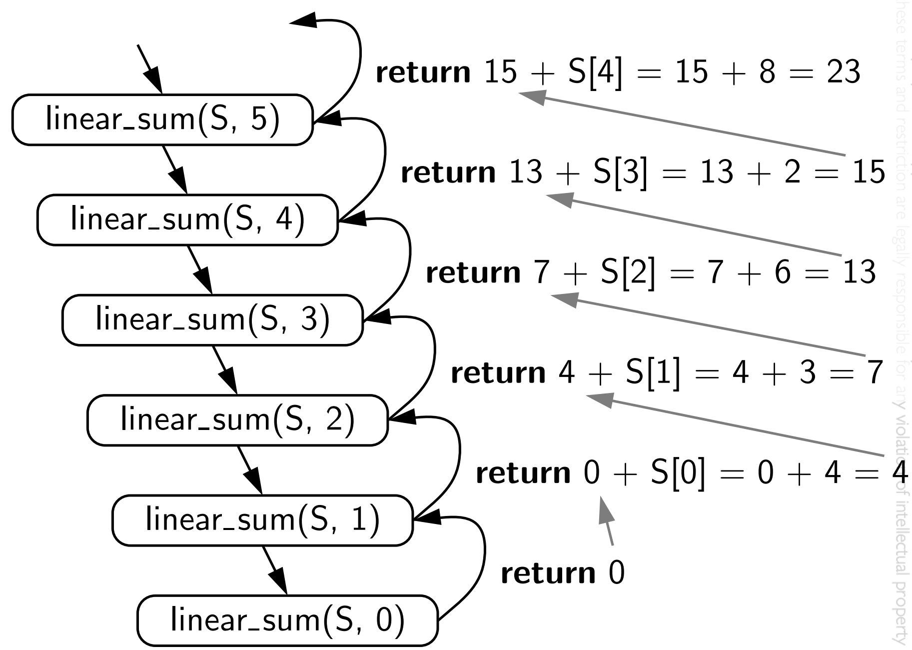


Linear Recursion

- Sum all elements in an array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	3	6	2	8	9	3	2	8	5	1	7	2	8	3	7

```python
def linear_sum(S, n):
  """Return the sum of the first n numbers of sequence S."""
  if n == 0:
    return 0
  else:
    return linear_sum(S, n−1) + S[n−1]
```



Binary Recursion

- Two recursive calls at a time

```
def binary_sum(S, start, stop):
  """Return the sum of the numbers in implicit slice S[start:stop]."""
  if start >= stop:                          # zero elements in slice
    return 0
  elif start == stop−1:                      # one element in slice
    return S[start]
  else:                                      # two or more elements in slice
    mid = (start + stop) // 2
    return binary_sum(S, start, mid) + binary_sum(S, mid, stop)
```

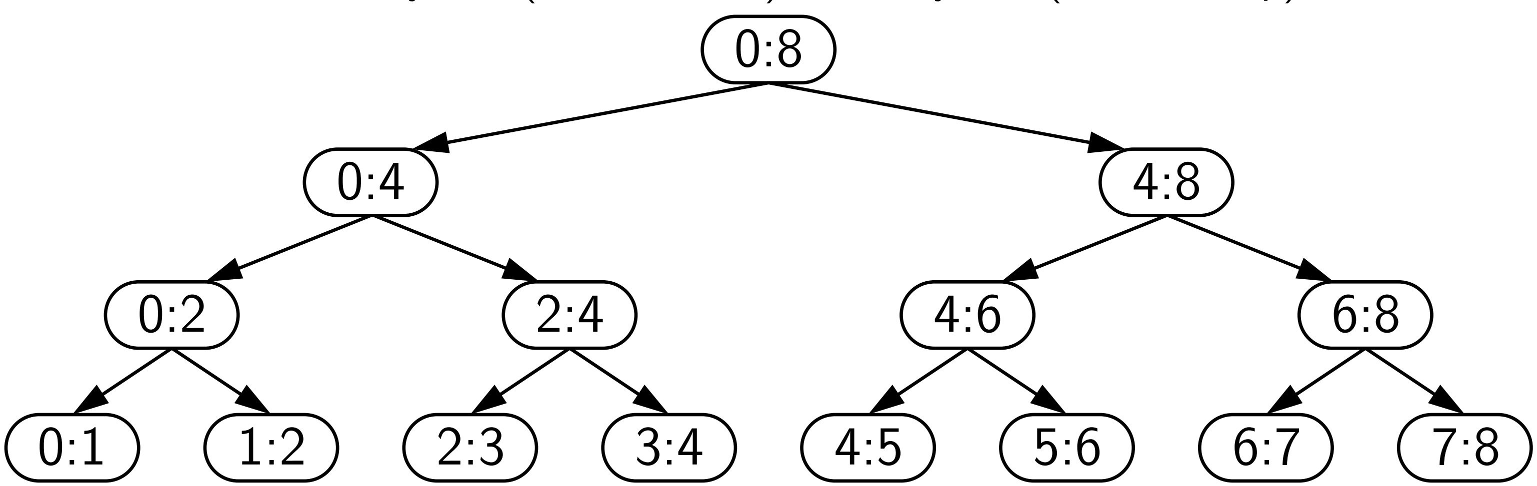


Figure 4.13: Recursion trace for the execution of binary_sum(0, 8).

Bad Recursion?

- Fibonacci sequence:

$$f(n) = \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f(n-1) + f(n-2) & else \end{cases}$$

- Technically correct, but terribly inefficient
  - Each recursion makes 2 calls
  - Similar to the Tower of Hanoi problem

```python
def bad_fibonacci(n):
    """Return the nth Fibonacci number."""
    if n <= 1:
        return n
    else:
        return bad_fibonacci(n-2) + bad_fibonacci(n-1)
```

Bad Recursion!

- c_n = number of calls performed in bad_fibonacci(n)
- When n = n + 2, the number of calls more than doubles
- $c_n > 2^{n/2}$

$$
\begin{aligned}
c_0 &= 1 \\
c_1 &= 1 \\
c_2 &= 1 + c_0 + c_1 = 1 + 1 + 1 = 3 \\
c_3 &= 1 + c_1 + c_2 = 1 + 1 + 3 = 5 \\
c_4 &= 1 + c_2 + c_3 = 1 + 3 + 5 = 9 \\
c_5 &= 1 + c_3 + c_4 = 1 + 5 + 9 = 15 \\
c_6 &= 1 + c_4 + c_5 = 1 + 9 + 15 = 25 \\
c_7 &= 1 + c_5 + c_6 = 1 + 15 + 25 = 41 \\
c_8 &= 1 + c_6 + c_7 = 1 + 25 + 41 = 67
\end{aligned}
$$

Good Recursion

- Why was bad_fibonacci so inefficient?
  - Each iteration requires 2 recursive calls: F_{n-1} and F_{n-2}
  - But computing F_{n-1} makes its own call of F_{n-2}
  - Can we reuse the information?
- Have the function to return two values: F_n and F_{n-1}
  - Thus, the next recursion has both F_{n-1} and F_{n-2} without recomputing them
  - Turns in to a linear recursion

```
def good_fibonacci(n):
  """Return pair of Fibonacci numbers, F(n) and F(n-1)."""
  if n <= 1:
    return (n,0)
  else:
    (a, b) = good_fibonacci(n−1)
    return (a+b, a)
```

Good Recursion 2

- Power function: $power(x, n) = x^n$

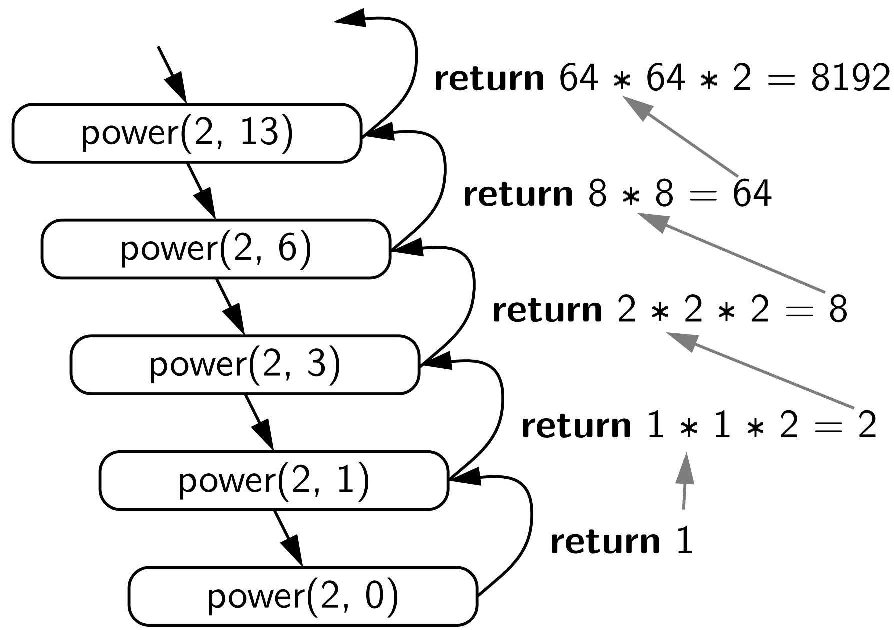
$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot power(x, n-1) & \text{otherwise.} \end{cases}$$

```
def power(x, n):
  """Compute the value x**n for integer n."""
  if n == 0:
    return 1
  else:
    return x * power(x, n-1)
```

Good Recursion 2

- (Better) Power function: $power(x, n) = x^n$

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 & \text{if } n > 0 \text{ is odd} \\ \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$




```
def power(x, n):
  """Compute the value x**n for integer n."""
  if n == 0:
    return 1
  else:
    partial = power(x, n // 2)          # rely on truncated division
    result = partial * partial
    if n % 2 == 1:                      # if n odd, include extra factor of x
      result *= x
    return result
```

Summary

- Often simple and intuitive
- Useful when dealing with “less linearly structured” data
  - e.g., trees, graphs
- In practice: need to be careful as it may not be so efficient
  - Unintendedly large time complexity
  - Good time complexity, but slow in practice due to many costly overhead operations
    - Not so well optimized
- For this course, we will see various recursive algorithms