# Recursion

Lecture 10

# We will learn

1. How to ask the recursion fairy to solve the problem for us.

# We will learn

1. How to ask the recursion fairy to solve the problem for us.
2. How to analyze the running time of a recursive algorithm.

# We will learn

1. How to ask the recursion fairy to solve the problem for us.
2. How to analyze the running time of a recursive algorithm.
3. Recursion in action
    1. Tower of Hanoi puzzle
    2. Merge sort
    3. Quick sort

# Recursion

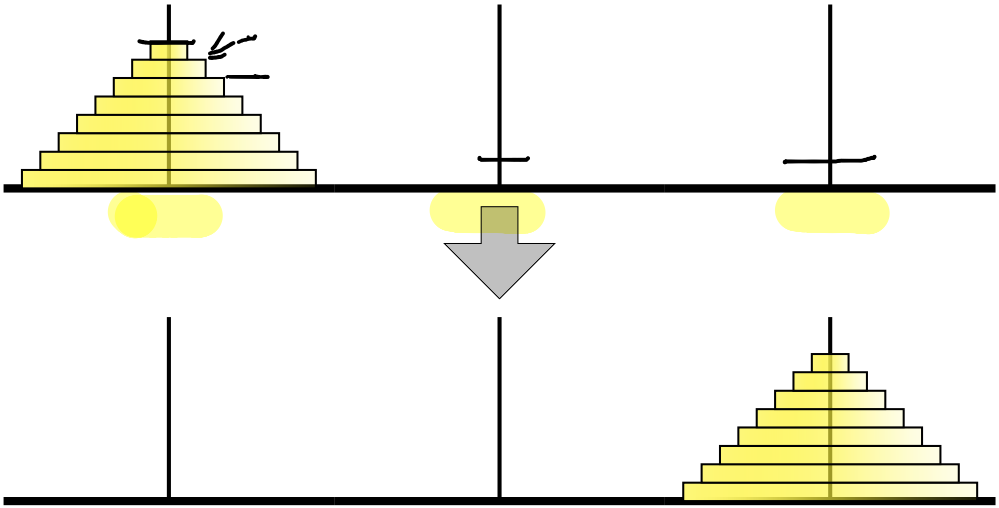
## How to think about it

Recursion = Induction

# Part I

# Tower of Hanoi

# Tower of Hanoi



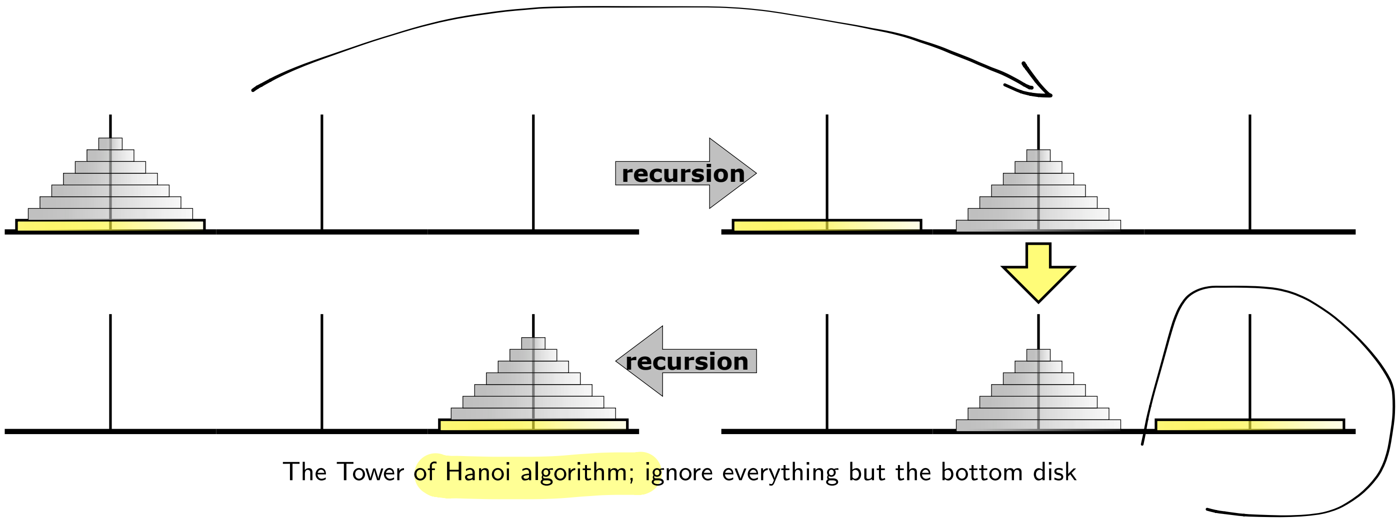
The Tower of Hanoi puzzle

Move stack of $n$ disks from peg **0** to peg **2**, one disk at a time.

Rule: cannot put a larger disk on a smaller disk.

Question: what is a strategy and how many moves does it take?

# Tower of Hanoi via Recursion




The Tower of Hanoi algorithm; ignore everything but the bottom disk

# Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

# Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

# Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

Proof of correctness.

# Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

Running time analysis.
$T(n)$: time to move $n$ disks via recursive strategy

# Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

$+1$

$f_n = 2f_{n-1}$

Running time analysis.

$T(n)$: time to move $n$ disks via recursive strategy

$$T(n) = 2T(n-1) + 1 \quad n > 1 \quad \text{and } T(1) = 1$$

$$T(n) - c = 2(T(n-1) - c) = \cdots = 2^{n-1}(T(1) - c)$$

$c - 2c = 1 \quad c = -1$

$$T(n) + 1 = 2^{n-1}(T(1) + 1) = 2^n$$

# Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2^2T(n-2)+2+1 \\
&= \ldots \\
&= 2^iT(n-i)+2^{i-1}+2^{i-2}+\ldots+1 \\
&= \ldots \\
&= 2^{n-1}T(1)+2^{n-2}+\ldots+1 \\
&= 2^{n-1}+2^{n-2}+\ldots+1 \\
&= (2^n-1)/(2-1)=2^n-1
\end{aligned}
$$

# Part II

## Merge Sort

# Sorting

Input Given an array of $n$ elements

Goal Rearrange them in ascending order

1. Input: Array $A[1 \ldots n]$

$$A\ L\ G\ O\ R\ I\ T\ H\ M\ S$$

# Merge Sort [von Neumann]

## MergeSort

1. Input: Array $A[1 \dots n]$

   $$A \; L \; G \; O \; R \; I \; T \; H \; M \; S$$

2. Divide into subarrays $A[1 \dots m]$ and $A[m+1 \dots n]$, where $m = \lfloor n/2 \rfloor$

   $$A \; L \; G \; O \; R \qquad I \; T \; H \; M \; S$$

1. Input: Array $A[1 \dots n]$

$$A\ L\ G\ O\ R\ I\ T\ H\ M\ S$$

2. Divide into subarrays $A[1 \dots m]$ and $A[m + 1 \dots n]$, where $m = \lfloor n/2 \rfloor$

$$A\ L\ G\ O\ R \quad I\ T\ H\ M\ S$$

3. Recursively **MergeSort** $A[1 \dots m]$ and $A[m + 1 \dots n]$

$$A\ G\ L\ O\ R \quad H\ I\ M\ S\ T$$

# Merge Sort [von Neumann]

## MergeSort

1. Input: Array $A[1 \ldots n]$

$$A\ L\ G\ O\ R\ I\ T\ H\ M\ S$$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m+1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

$$A\ L\ G\ O\ R \quad I\ T\ H\ M\ S$$

3. Recursively **MergeSort** $A[1 \ldots m]$ and $A[m+1 \ldots n]$

$$A\ G\ L\ O\ R \quad H\ I\ M\ S\ T$$

4. Merge the sorted arrays

$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

# Merging Sorted Arrays

1. Use a new array $B$ to store the merged array
2. Scan $A[1 \ldots m]$ and $A[m+1 \ldots n]$ from left-to-right, storing elements in $B$ in order

$$A \; G \; L \; O \; R \qquad H \; I \; M \; S \; T$$

$$A$$

# Merging Sorted Arrays

1. Use a new array $B$ to store the merged array
2. Scan $A[1 \ldots m]$ and $A[m+1 \ldots n]$ from left-to-right, storing elements in $B$ in order

$$A \; G \; L \; O \; R \qquad H \; I \; M \; S \; T$$
$$A \; G$$

# Merging Sorted Arrays

1. Use a new array $B$ to store the merged array
2. Scan $A[1 \dots m]$ and $A[m+1 \dots n]$ from left-to-right, storing elements in $B$ in order

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$
$$A\ G\ H$$

# Merging Sorted Arrays

1. Use a new array $B$ to store the merged array
2. Scan $A[1 \ldots m]$ and $A[m+1 \ldots n]$ from left-to-right, storing elements in $B$ in order

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$

$$A\ G\ H\ I$$

# Merging Sorted Arrays

1. Use a new array $B$ to store the merged array
2. Scan $A[1 \ldots m]$ and $A[m+1 \ldots n]$ from left-to-right, storing elements in $B$ in order

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

# Formal Code

```
MergeSort(A[1..n]):
  if n > 1
    m ← ⌊n/2⌋
    MergeSort(A[1..m])
    MergeSort(A[m+1..n])
    Merge(A[1..n], m)
```

```
Merge(A[1..n], m):
  i ← 1;  j ← m + 1
  for k ← 1 to n
    if j > n
      B[k] ← A[i];  i ← i + 1
    else if i > m
      B[k] ← A[j];  j ← j + 1
    else if A[i] < A[j]
      B[k] ← A[i];  i ← i + 1
    else
      B[k] ← A[j];  j ← j + 1

  for k ← 1 to n
    A[k] ← B[k]
```

# Proving Correctness

Obvious way to prove correctness of recursive algorithm:

# Proving Correctness

Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on $n$ that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct?

# Proving Correctness

Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on $n$ that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct? Also by induction!
- One way is to rewrite Merge into a recursive version.
- For algorithms with loops one comes up with a natural *loop invariant* that captures all the essential properties and then we prove the loop invariant by induction on the index of the loop.

# Proving Correctness

Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on $n$ that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct? Also by induction!
- One way is to rewrite Merge into a recursive version.
- For algorithms with loops one comes up with a natural *loop invariant* that captures all the essential properties and then we prove the loop invariant by induction on the index of the loop.

At the start of iteration $k$ the following hold:

- $B[1..k]$ contains the smallest $k$ elements of $A$ correctly sorted.
- $B[1..k]$ contains the elements of $A[1..(i-1)]$ and $A[(m+1)..(j-1)]$.
- No element of $A$ is modified.

# Running Time

$T(n)$: time for merge sort to sort an $n$ element array

# Running Time

$T(n)$: time for merge sort to sort an $n$ element array

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

# Running Time

$T(n)$: time for merge sort to sort an $n$ element array

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

What do we want as a solution to the recurrence?

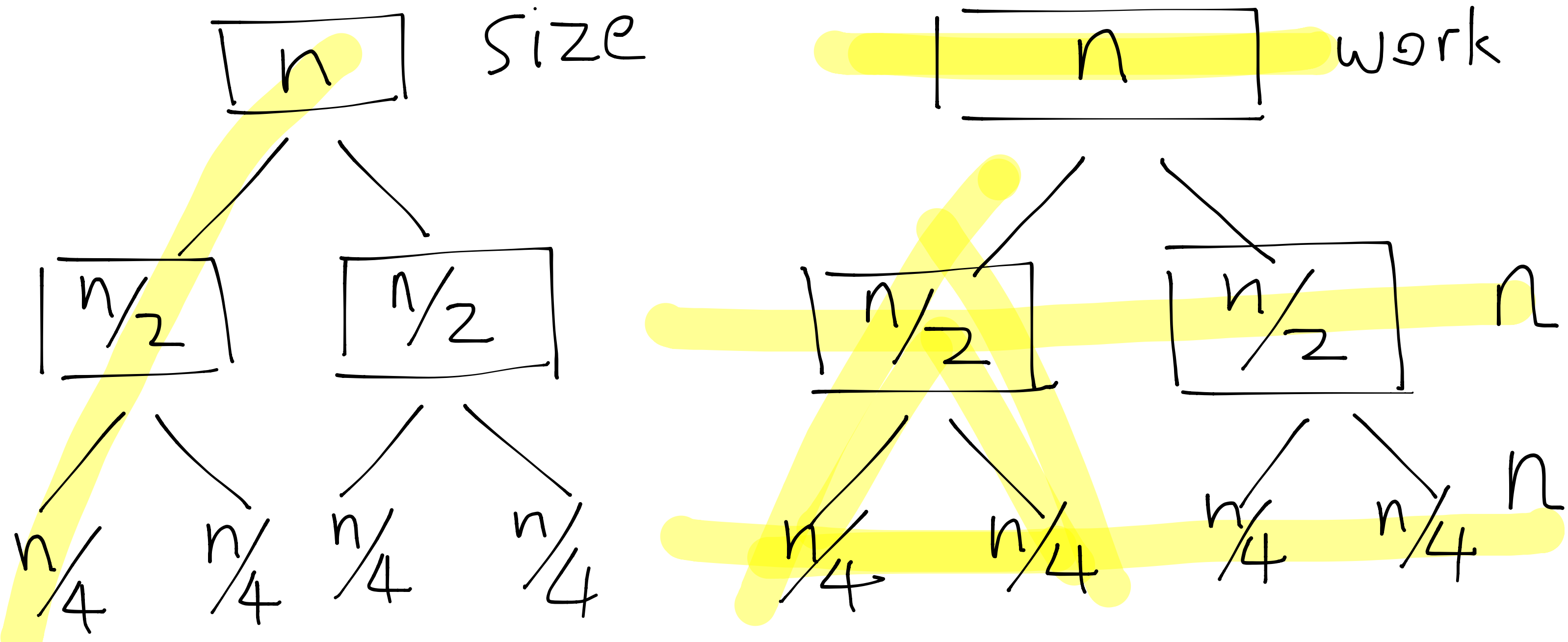
Almost always only an *asymptotically* tight bound. That is we want to know $f(n)$ such that $T(n) = \Theta(f(n))$.

1. $T(n) = O(f(n))$ - upper bound
2. $T(n) = \Omega(f(n))$ - lower bound

# Solving Recurrences: Some Techniques

1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus
2. Expand the recurrence and spot a pattern and use simple math
3. Recursion tree method — imagine the computation as a tree
4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds

# Recursion Trees




⋮

1

$$2^k = n$$

$$k = \log n$$

$$O(n \log n)$$

# Part III

## Quick Sort

# Quick Sort

## Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

# Quick Sort

## Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

# Quick Sort

## Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.

## Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.

# Quick Sort: Example

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: ~~16~~ 1

{ }     1     everything else

$\boxed{n}$ size

$\boxed{n-1}$

$\boxed{n-2}$

$\vdots$

$\boxed{1}$

work

$\boxed{n}$ $\quad n$

$\boxed{n-1}$ $\quad \frac{n(n+1)}{2}$

$\boxed{n-2}$ $\quad O(n^2)$

$\vdots$

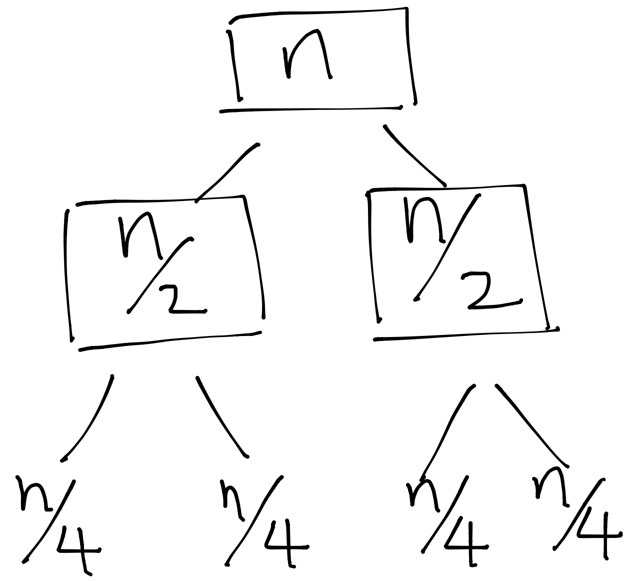
$1$

# Time Analysis

1. Let $k$ be the rank of the chosen pivot. Then,
$T(n) = T(k-1) + T(n-k) + O(n)$

# Time Analysis

1. Let $k$ be the rank of the chosen pivot. Then,
   $T(n) = T(k-1) + T(n-k) + O(n)$
2. If $k = \lceil n/2 \rceil$ then $T(n) =$
   $T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$.
   Then, $T(n) = O(n \log n)$.



# Time Analysis

1. Let $k$ be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$
2. If $k = \lceil n/2 \rceil$ then $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$. Then, $T(n) = O(n \log n)$.
   1. Theoretically, median can be found in linear time.

# Time Analysis

1. Let $k$ be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$
2. If $k = \lceil n/2 \rceil$ then $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$. Then, $T(n) = O(n \log n)$.
   1. Theoretically, median can be found in linear time.
3. Typically, pivot is the first or last element of array. Then,

$$T(n) = \max_{1 \leq k \leq n} (\underbrace{T(k-1)}_{0} + \underbrace{T(n-k)}_{0} + O(n))$$

In the worst case $T(n) = T(n-1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.

# Recursion Trees