MAKE
SCHOOL

RECURSIVE ALGORITHM ANALYSIS

Wait, so... to find the answer... I have to know the answer?!

ASYMPTOTIC NOTATION

Worst case – upper bound

Algorithm is $\mathbf{O}(f(n))$ – "big oh of $f(n)$"

Best case – lower bound

Algorithm is $\mathbf{\Omega}(f(n))$ – "omega of $f(n)$"

If both bounds are the same, then

Algorithm is $\mathbf{\Theta}(f(n))$ – "theta of $f(n)$"

MAKE SCHOOL

BINARY SEARCH

Return index of `item` in sorted `array`, or `None` if not found

```
def binary_search(array, item):
    left, right = 0, len(array)
    while left <= right:
        middle = (left + right) / 2
        if item == array[middle]:
            return middle  # found
        elif item < array[middle]:
            right = middle - 1  # search left half
        else:
            left = middle + 1  # search right half
    return None  # not found
```

MAKE SCHOOL

RECURSIVE BINARY SEARCH

Return index of `item` in sorted `array`, or `None` if not found

```
def binary_search(array, item, left, right):
    if left > right:
        return None  # not found
    middle = (left + right) / 2
    if item == array[middle]:
        return middle  # found
    elif item < array[middle]:  # search left half
        return binary_search(..., left, middle-1)
    else:  # search right half
        return binary_search(..., middle+1, right)
```

MAKE SCHOOL

RECURSIVE ALGORITHMS

Recursive algorithms call themselves with different input values until reaching a base case solution or terminating condition

To account for recursive calls, we can write a ***recurrence relation*** to describe running time

e.g., Binary Search: $T(n) = 4 + T(n/2)$

MAKE SCHOOL

BINARY SEARCH ANALYSIS

Let's unwrap the recurrence for Binary Search:

$T(n) = 4 + T(n/2)$	after 1 iteration
$T(n) = 4 + 4 + T(n/4)$	after 2 iterations
$T(n) = 4 + 4 + 4 + T(n/8)$	after 3 iterations
$T(n) = 4i + T(n/2^i)$	after i iterations

2^i can be at most n (terminates with sublist size 1)

→ i can be at most $\log_2 n$ (max depth of recursion)

∴ Binary Search is $O(\log_2 n)$ and $\Omega(1)$ (early exit)

MAKE SCHOOL

MERGE SORT

Merge Sort steps at a high level:

1. If list size is **1**, return the list (it's trivially sorted)
2. Recursively sort each **1/2** of the list (**2** calls)
3. Merge the **2** sorted **1/2**-lists into a size n list

Running time recurrence relation:

$T(1) = 1$ base case (list size 1)

$T(n) = 2T(n/2) + n$ each recursive iteration

MAKE SCHOOL

MERGE SORT ANALYSIS

Let's unwrap the recurrence for Merge Sort:

$T(n) = 2T(n/2) + n$ after 1 iteration

$T(n) = 4T(n/4) + 2n$ after 2 iterations

$T(n) = 8T(n/8) + 3n$ after 3 iterations

$T(n) = 2^iT(n/2^i) + in$ after i iterations

2^i must reach n (base case with sublist size 1)

→ i must reach $\log_2 n$ (depth of recursion tree)

∴ Merge Sort is $\Theta(n\log_2 n)$

MAKE SCHOOL

MASTER THEOREM

Given a general recurrence relation:

$T(n) = aT(n/b) + cn^d$

$T(1) = c$

where $a \geq 1$, $b > 1$, $c > 0$, $d \geq 0$

If $a < b^d$ then $T(n)$ is $\Theta(n^d)$

If $a = b^d$ then $T(n)$ is $\Theta(n^d \log_b n)$

If $a > b^d$ then $T(n)$ is $\Theta(n^e)$ where $e = \log_b a$

MAKE SCHOOL

RESOURCES

Introduction to Algorithms by Cormen, Leiserson, Rivest, and Stein – widely considered *the Bible of Algorithms*

Algorithms Unlocked by Thomas Cormen – introductory and more accessible, less technical detail than CLRS

Recursion and Recurrences by U. of Dartmouth Math

Sorting-Algorithms.com – animations of common sorting algorithms running in parallel on specific datasets

MAKE SCHOOL