Recursive Problem Solving

Elyse Cornwall

July 12th, 2023

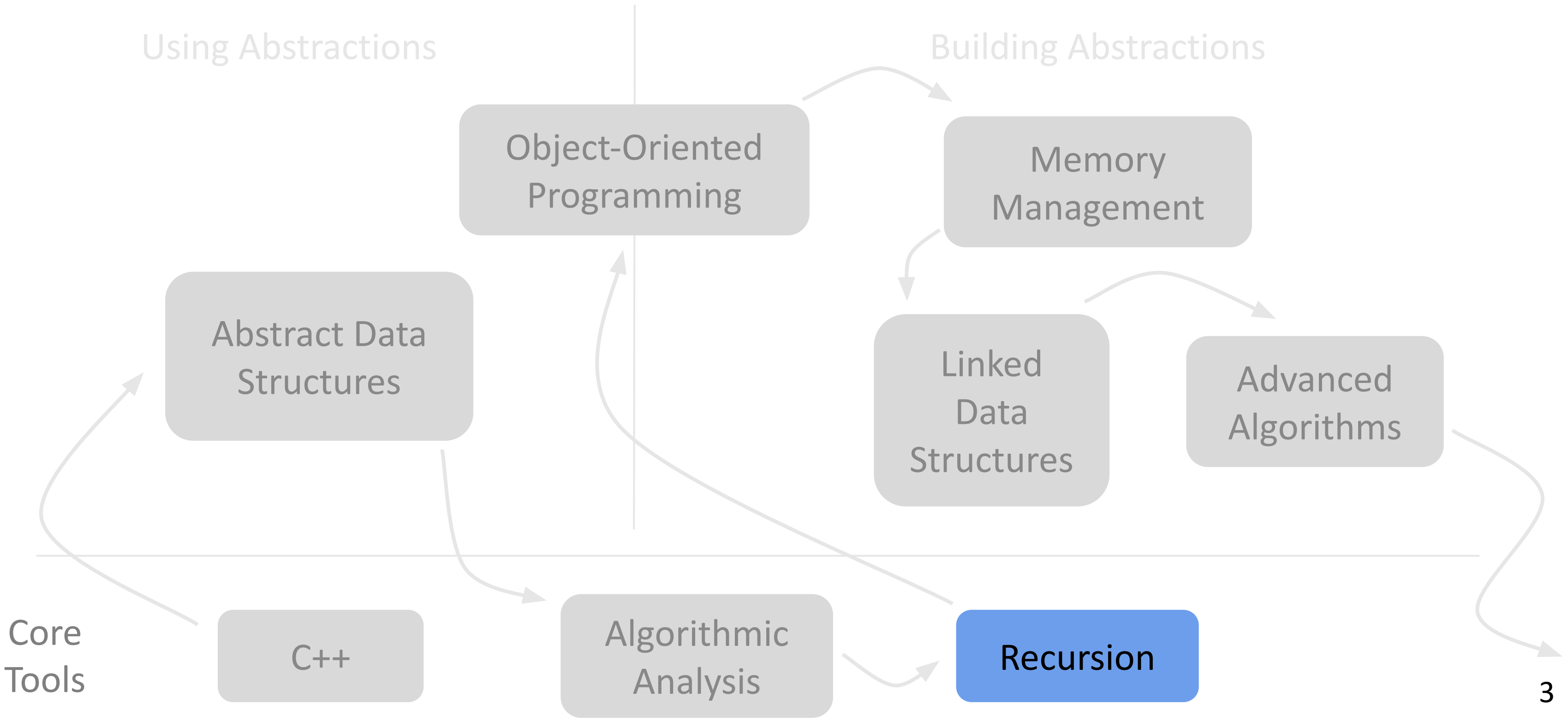
Contributions made from previous CS106B Instructors

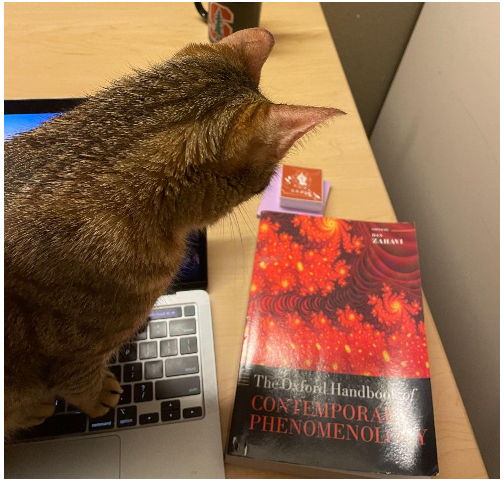
Announcements

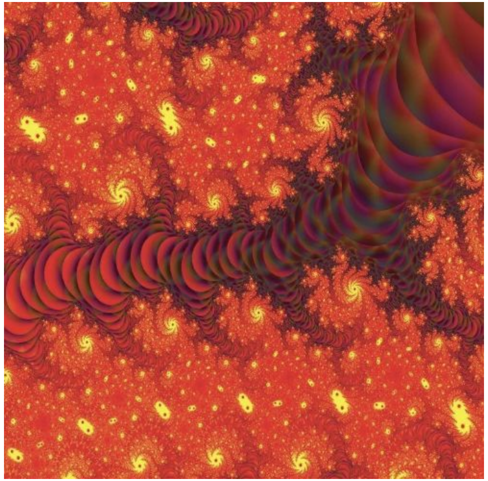
- HW1 IGs this week
- Midterm exam next Monday
  - Find all logistics and practice material here
  - Today is the last lecture covered on the midterm
  - Thursday and Friday optional review
  - No class on Monday
- First part of Assignment 3 released Friday (helpful recursion practice)
  - Rest of assignment comes out after the midterm

Roadmap

Wow, we've made lots of progress!





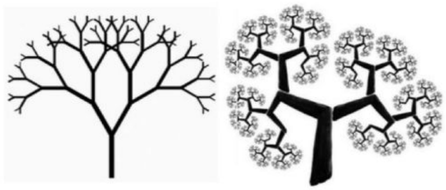


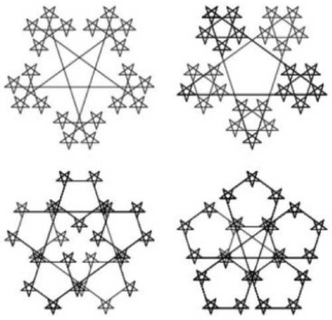
(Sidibou discovers fractals)

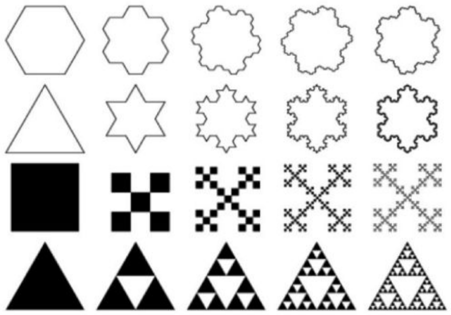
Fractals Recap

Fractal

- Any repeated, graphical pattern
- Composed of repeated instances of the same shape or pattern, arranged in a structured way

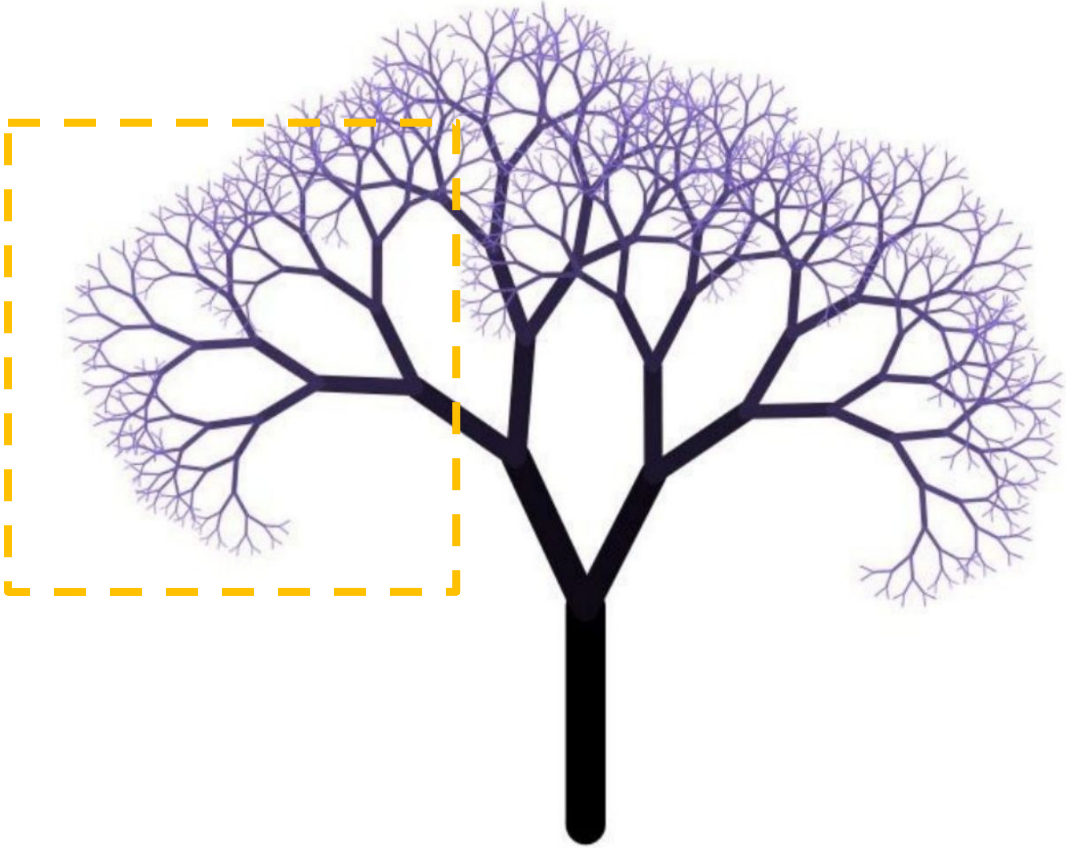






What differentiates the smaller tree from the bigger one?

1. It's at a different **position**
2. It has a different **size**
3. It has a different **orientation**
4. It has a different order



Fractals and self-similar structures are often defined in terms of some parameter called the order, which indicates the complexity of the overall structure.

Iteration + Recursion

- It’s completely reasonable to mix iteration and recursion in the same function.
- Recursion doesn’t mean “the absence of iteration.” It just means “solving a problem by solving smaller copies of that same problem.”
- Iteration and recursion can be very powerful in combination!

Why do we use recursion?

Why do we use recursion?

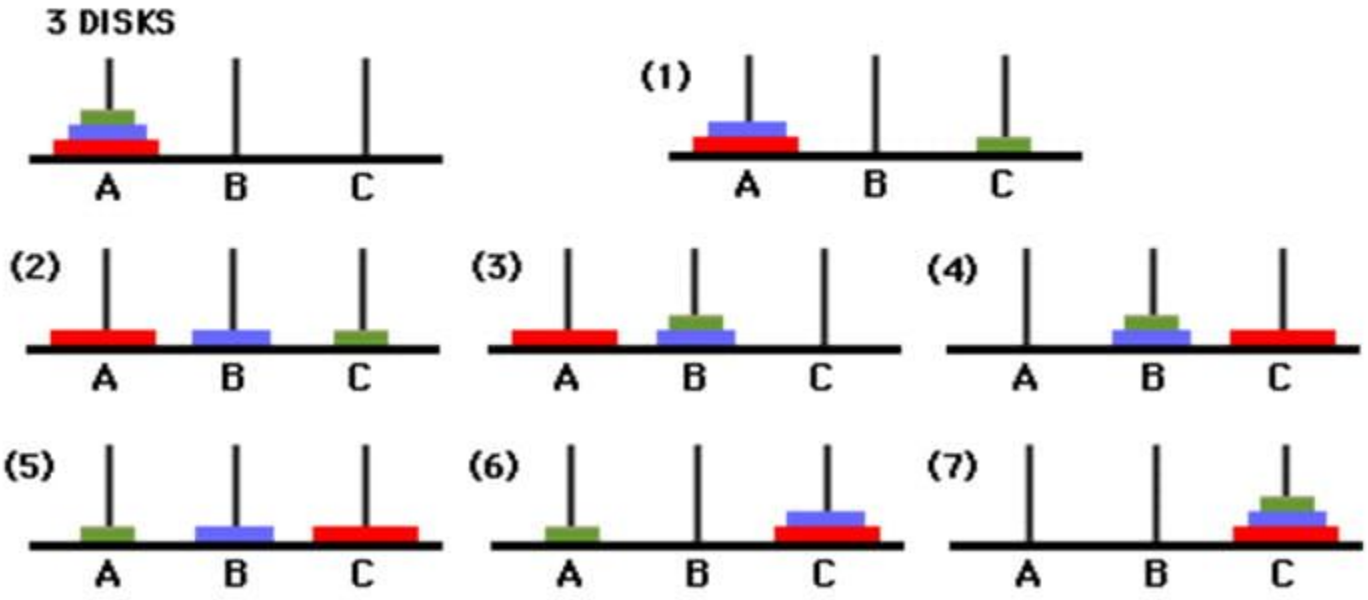
- Elegant
  - Some problems have beautiful, concise recursive solutions
- Efficient
  - Recursive solutions can have faster runtimes
- Dynamic
  - We’ll explore **recursive backtracking** next week :)

An *elegant* solution: Tower of Hanoi

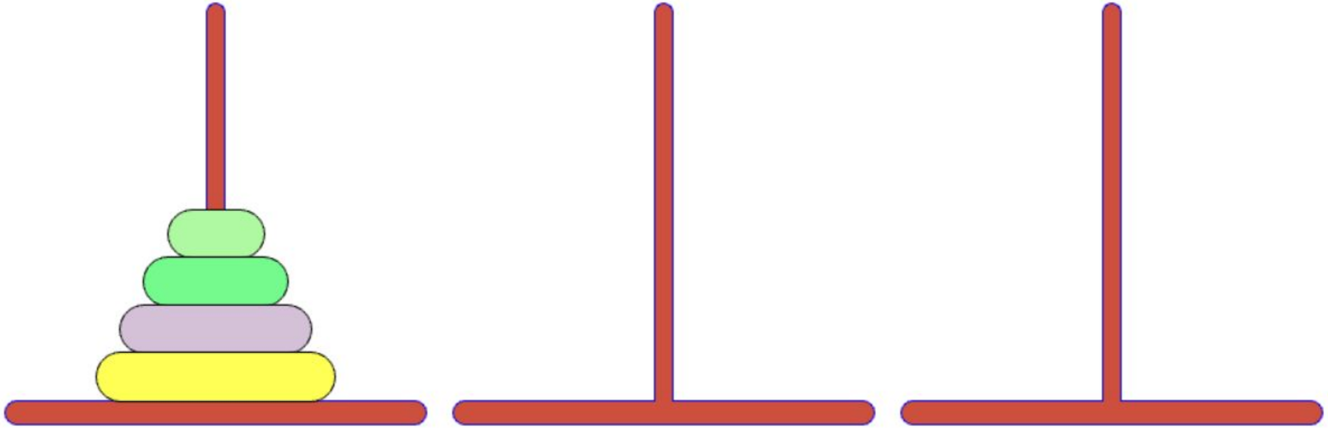
Tower of Hanoi

- Try playing online!
  - https://www.mathsisfun.com/games/towerofhanoi.html
- What strategies do you use? Try getting to 5 disks.

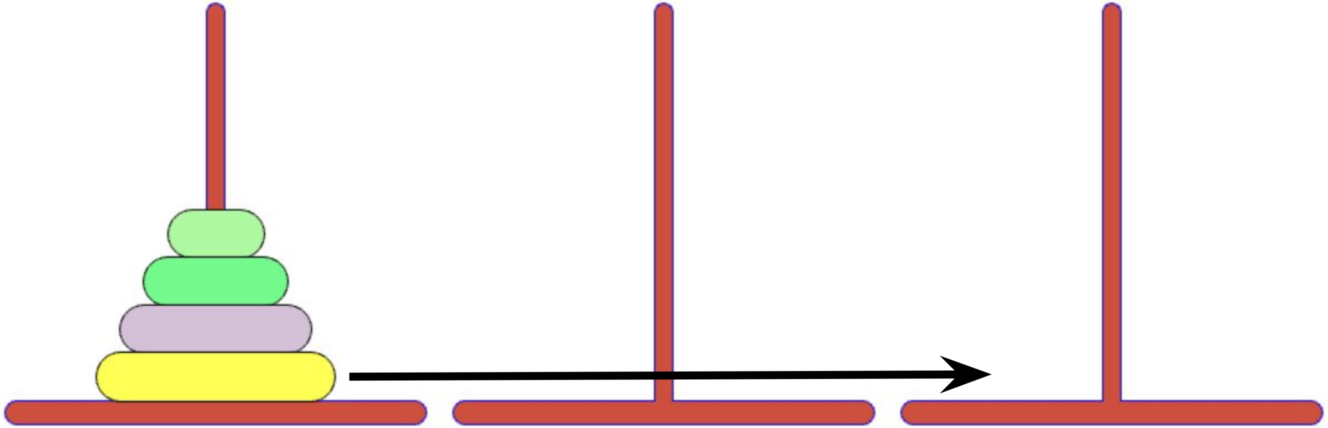
Solving with 3 Disks



Solving with 4 Disks

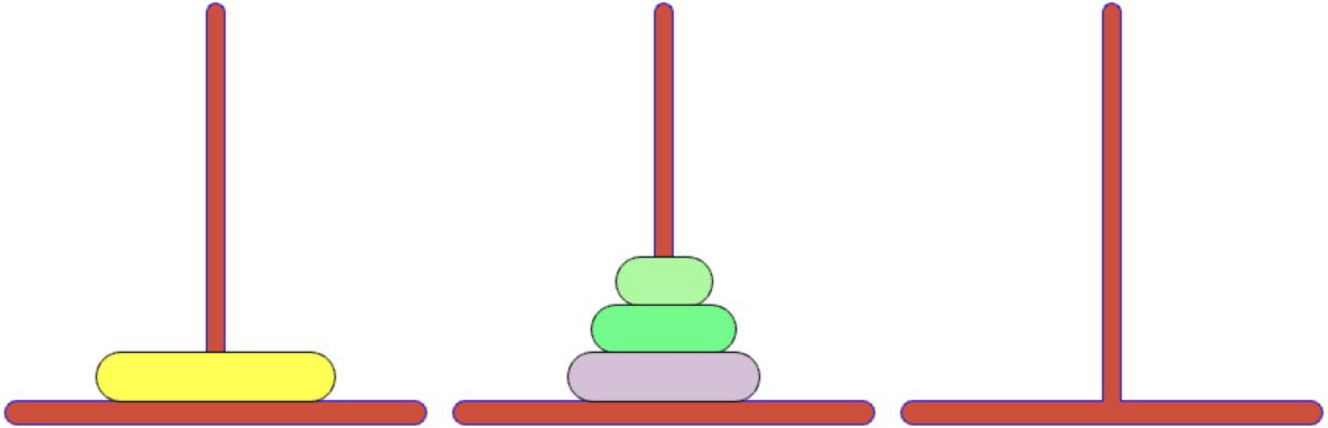


Solving with 4 Disks



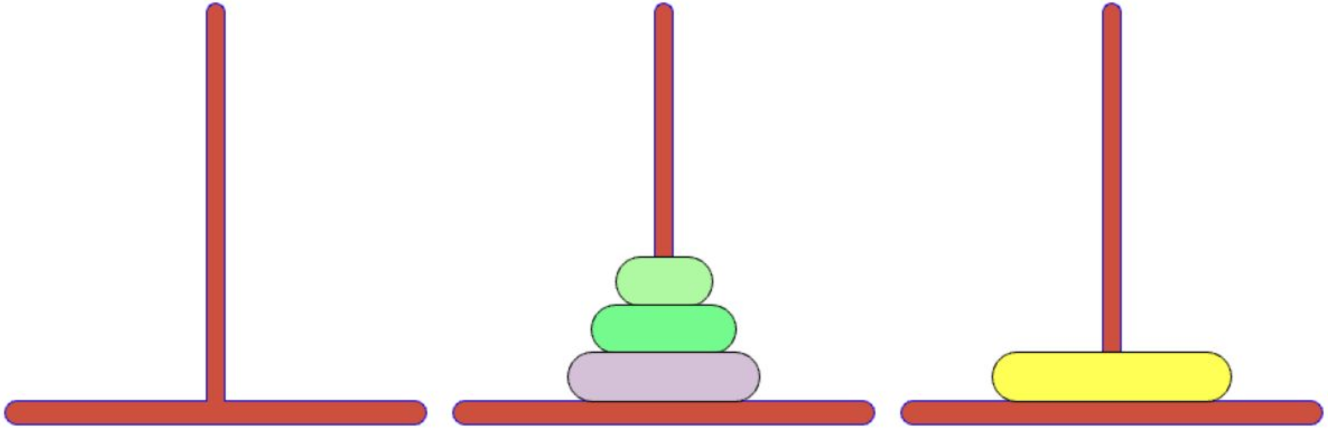
Eventually, we need to get this bottom disk over here.

Solving with 4 Disks



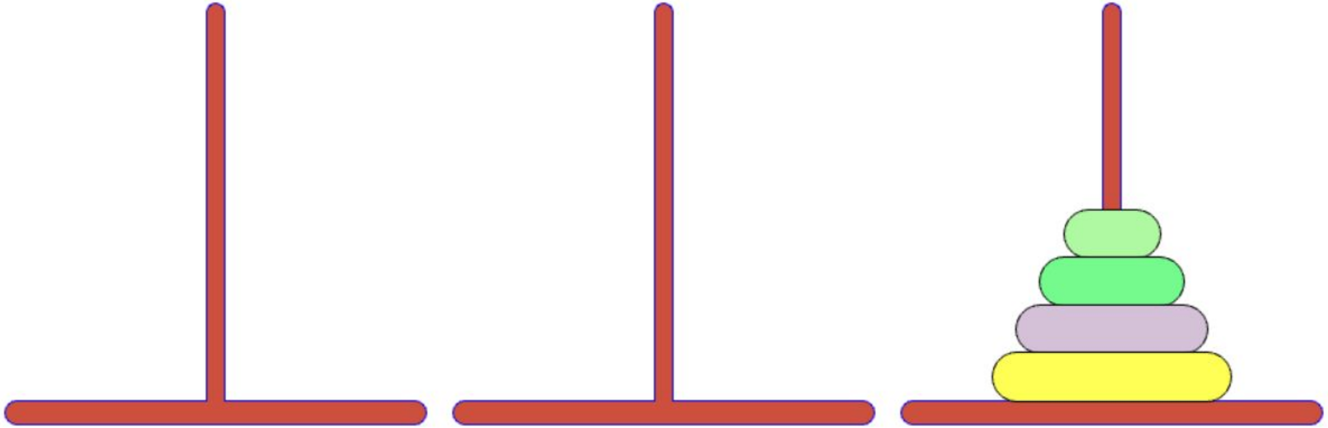
We'll need to get the smaller 3 disks out of the way,

Solving with 4 Disks



Move the bottom piece over…

Solving with 4 Disks



Then stack the 3 smaller disks on top.

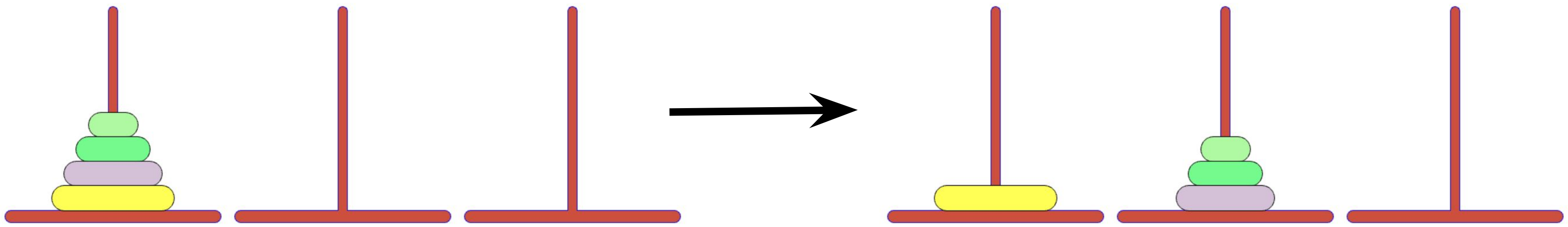
Solving with 4 Disks

1. Move tower of 3 disks onto middle peg
2. Move 4th disk over
3. Move tower of 3 disks onto end peg

Solving with 4 Disks

1. Move tower of 3 disks onto middle peg
2. Move 4th disk over
3. Move tower of 3 disks onto end peg

We know how to do steps 1 and 3 - same as solving with 3 disks.



Solving with 5 Disks

1. Move tower of 4 disks onto middle peg
2. Move 5th disk over
3. Move tower of 4 disks onto end peg

We know how to do steps 1 and 3 - same as solving with 4 disks.

Solving with N Disks

1. Move tower of N-1 disks onto middle peg
2. Move Nth disk over
3. Move tower of N-1 disks onto end peg

Tower of Hanoi as a Recursive Process

To **solve** Tower of Hanoi for N disks:

Solve for N-1 disks (but place on the middle peg)

Move Nth disk over to the end peg

Solve for N-1 disks (but move from middle peg to end peg)

Tower of Hanoi as a Recursive Process

To **solve** Tower of Hanoi for N disks:

Solve for N-1 disks (but place on the middle peg)

Move Nth disk over to the end peg

Solve for N-1 disks (but move from middle peg to end peg)

Is that really it? Let's code it up! 💻

Solution

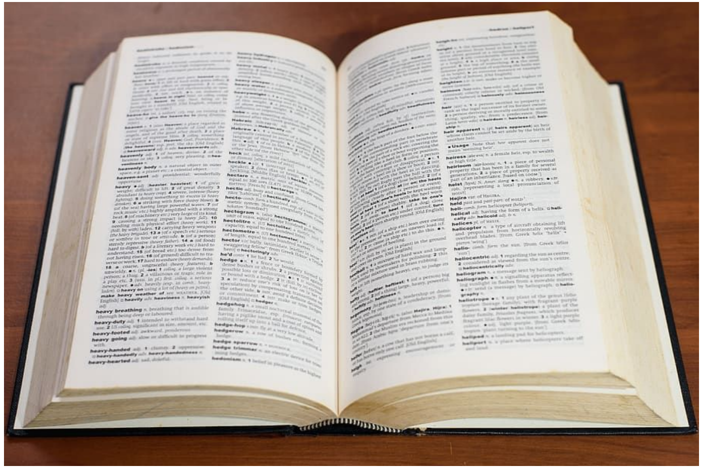
```
void solveTowers(int n, char start, char end, char aux) {
    if (n == 0) {
        return;
    }
    solveTowers(n-1, start, aux, end);
    moveSingleDisk(start, end);
    solveTowers(n-1, aux, end, start);
}
```

An *efficient* solution:
Binary Search

Dictionary Lookups

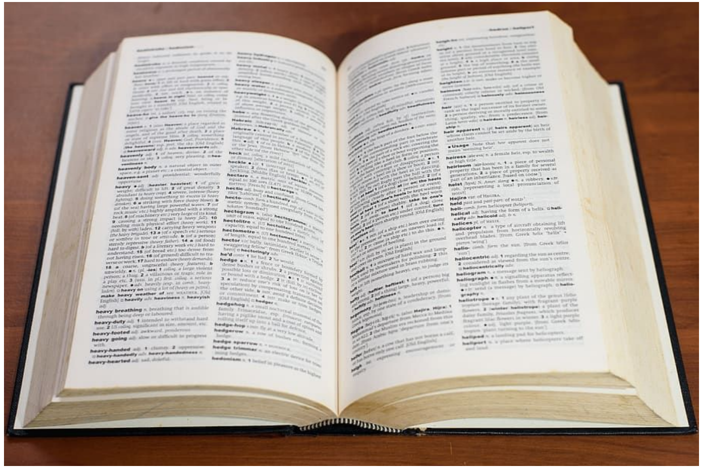
👥 What's your algorithm for finding a word in a dictionary?

- Where do you start?
- If the first page you look at doesn't have the word, how do you proceed?



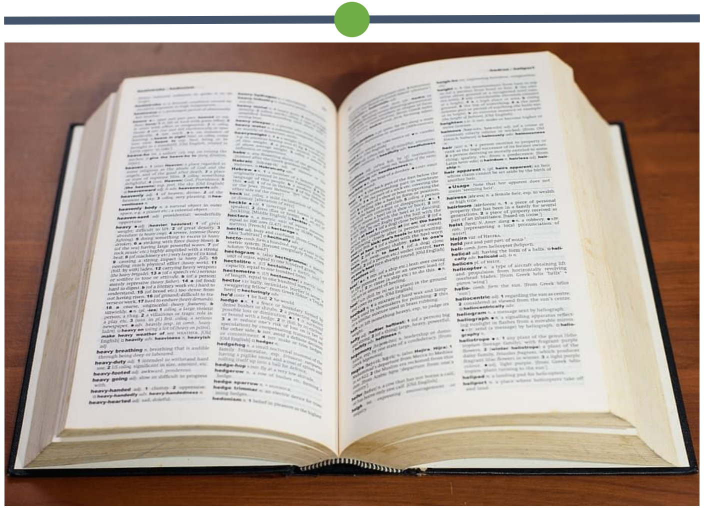
Dictionary Lookups

1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
    a. Repeat step 2 until the word is found



Dictionary Lookups

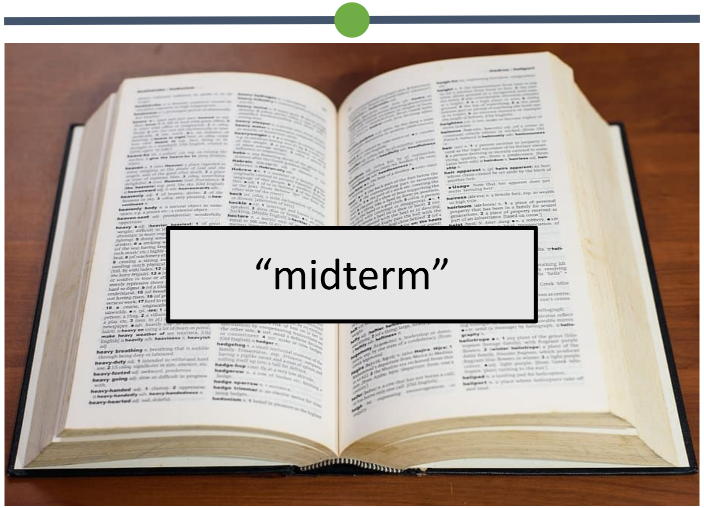
1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
    a. Repeat step 2 until the word is found



Total pages: 1000
Current page: 500
Looking up: "toaster"

Dictionary Lookups

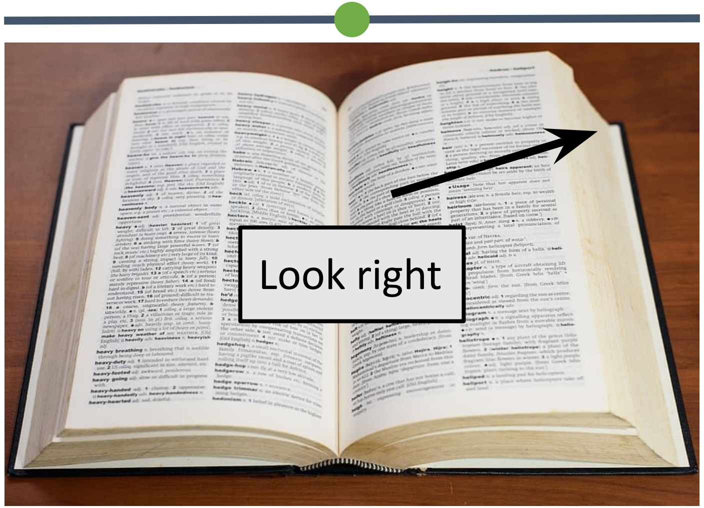
1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
    a. Repeat step 2 until the word is found



Total pages: 1000
Current page: 500
Looking up: "toaster"

Dictionary Lookups

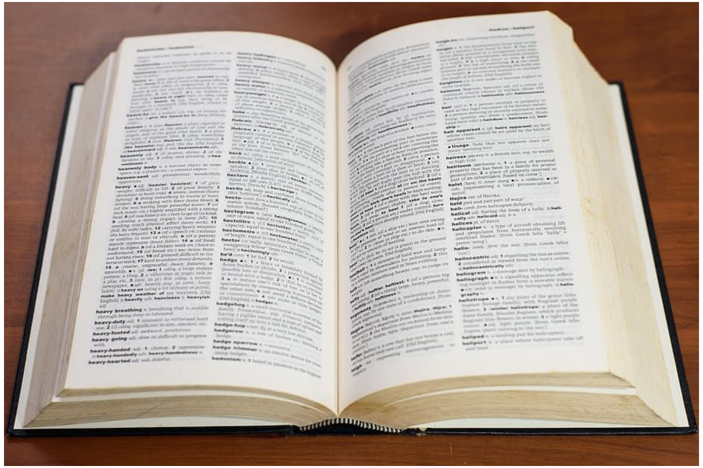
1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
    a. Repeat step 2 until the word is found



Total pages: 1000
Current page: 500
Looking up: "toaster"

Dictionary Lookups

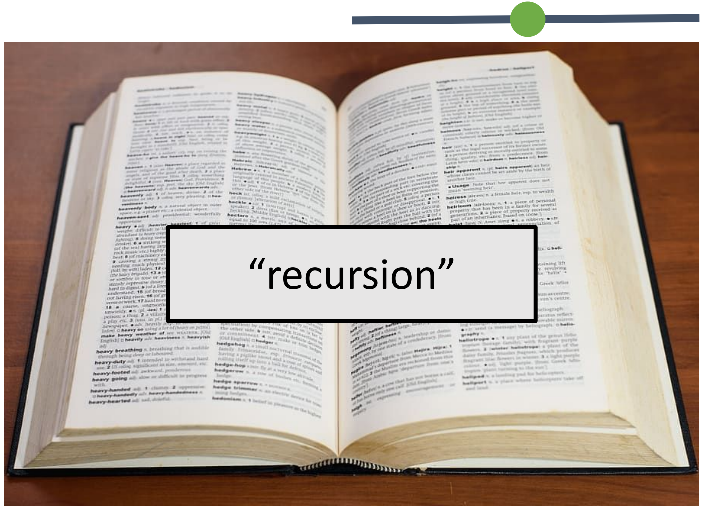
1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
   a. Repeat step 2 until the word is found



Total pages: 1000
Current page: **750**
Looking up: "toaster"

Dictionary Lookups

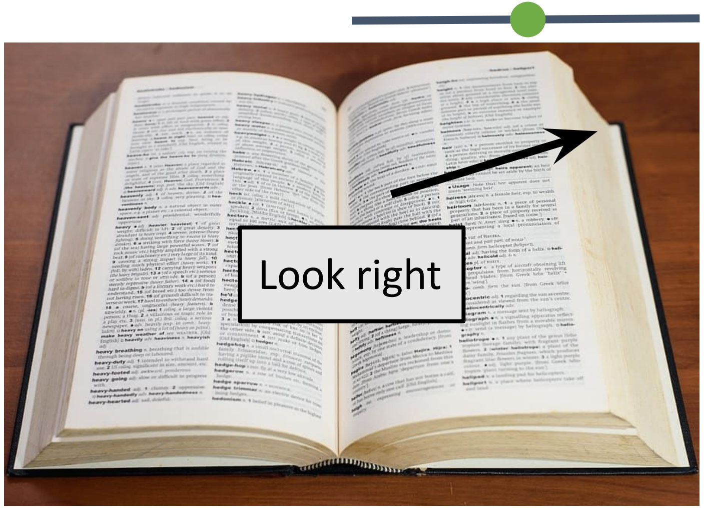
1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
    a. Repeat step 2 until the word is found



Total pages: 1000
Current page: 750
Looking up: "toaster"

Dictionary Lookups

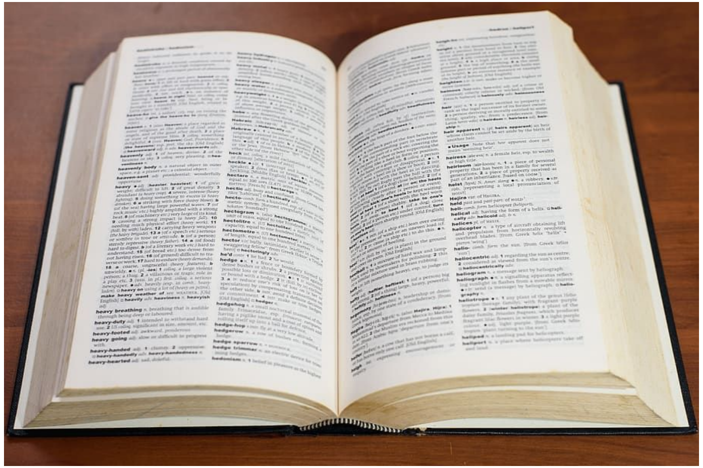
1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
    a. Repeat step 2 until the word is found



Total pages: 1000
Current page: 750
Looking up: "toaster"

Dictionary Lookups

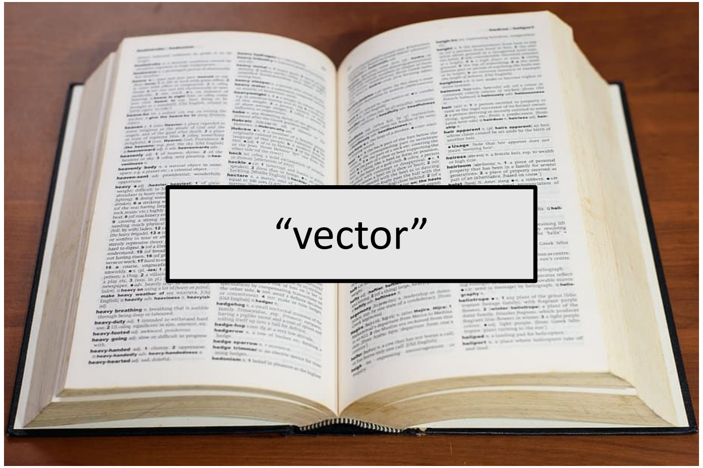
1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
   a. Repeat step 2 until the word is found



Total pages: 1000
Current page: **875**
Looking up: "toaster"

Dictionary Lookups

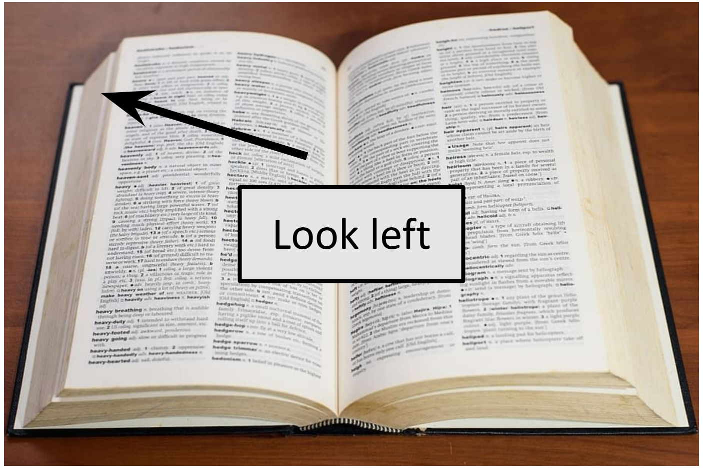
1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
    a. Repeat step 2 until the word is found



Total pages: 1000
Current page: 875
Looking up: "toaster"

Dictionary Lookups

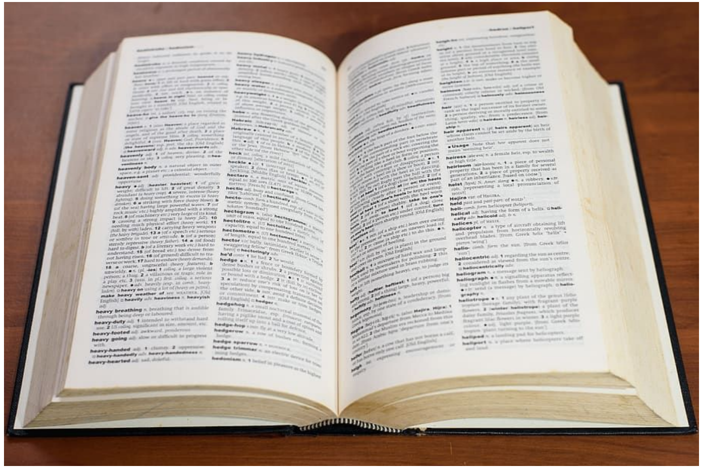
1. Open the dictionary to somewhere in the middle
2. If the word isn’t on this page, look left or right
    a. Repeat step 2 until the word is found



Total pages: 1000
Current page: 875
Looking up: “toaster”

Dictionary Lookups

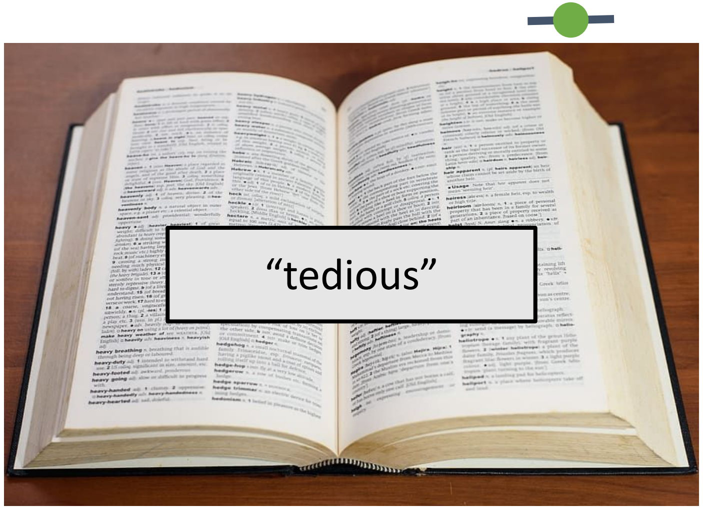
1. Open the dictionary to somewhere in the middle
2. If the word isn’t on this page, look left or right
   a. Repeat step 2 until the word is found



Total pages: 1000
Current page: **812**
Looking up: “toaster”

Dictionary Lookups

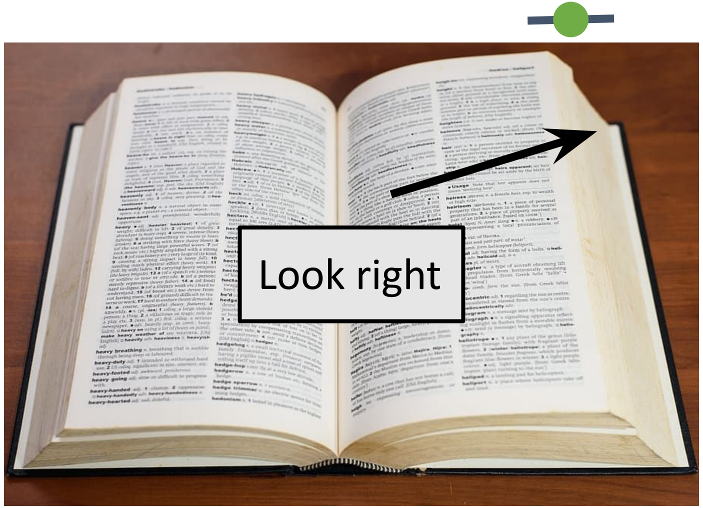
1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
    a. Repeat step 2 until the word is found



Total pages: 1000
Current page: 812
Looking up: "toaster"

Dictionary Lookups

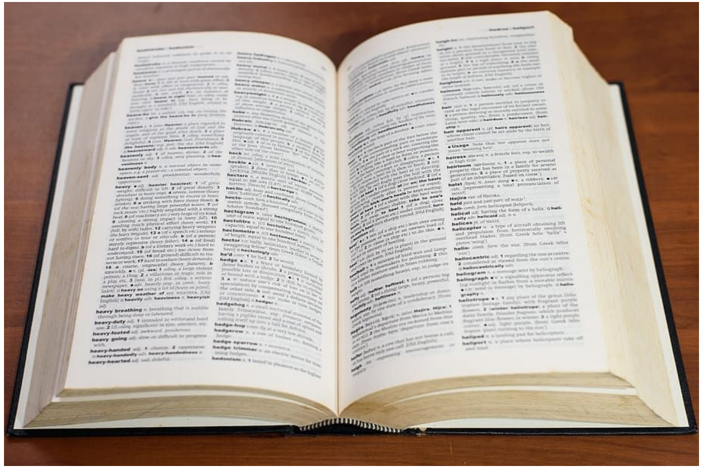
1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
   a. Repeat step 2 until the word is found



Total pages: 1000
Current page: 812
Looking up: "toaster"

Dictionary Lookups

1. Open the dictionary to somewhere in the middle
2. If the word isn’t on this page, look left or right
   a. Repeat step 2 until the word is found



Total pages: 1000
Current page: **844**
Looking up: “toaster”

Dictionary Lookups

1. Open the dictionary to somewhere in the middle
2. If the word isn't on this page, look left or right
   a. Repeat step 2 until the word is found

Total pages: 1000
Current page: **844**
Looking up: "toaster"

👥 Was This Efficient?

- How many pages did we have to read to find the answer?
- How many pages would we have to read if we did a linear search (scanning from beginning to end)?

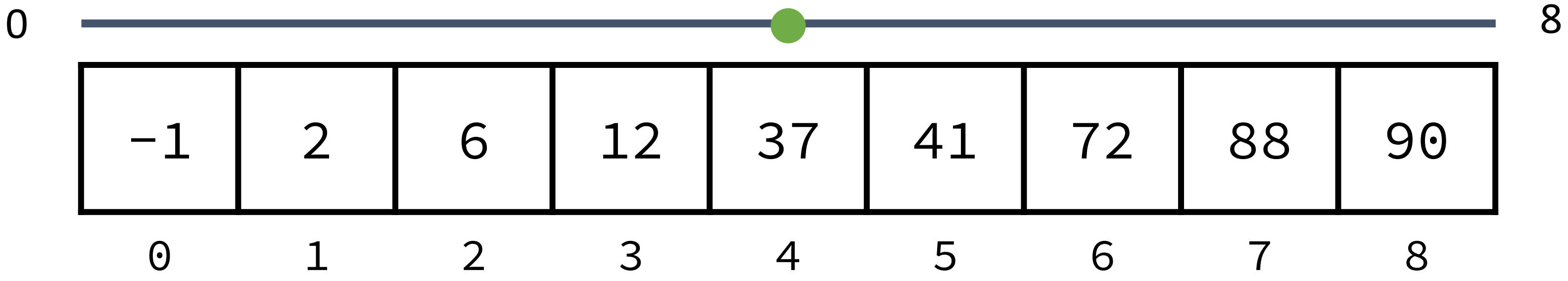
Binary Search

- Let’s say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?

-1	2	6	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

Binary Search

- Let’s say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?



Let’s try to find the number 6 in our Vector

Binary Search

Let's try to find the number 6

- Let's say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?

0 8

-1	2	6	12	**37**	41	72	88	90
0	1	2	3	4	5	6	7	8

Too big, look left

Binary Search

Let's try to find the number 6

- Let's say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?

0 3

-1	2	6	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

When there are two middles, we'll just choose the first one

Binary Search

Let's try to find the number 6

- Let's say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?

0

3

-1	**2**	6	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

Too small, look right

Binary Search

Let’s try to find the number 6

👥 *What’s going to happen to our search range?*

What is our new range of indices?

- Let’
- Can … ber?

0 ——●—— 3

-1	**2**	6	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

Too small, look right

Binary Search

Let's try to find the number 6

- Let's say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?

-1	2	6	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

Binary Search

- Let’s say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?

2 3

-1	2	**6**	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

Found it! 🎉🎉🎉

Binary Search as a Recursive Process

Binary search over some range of sorted elements:

1. Choose element in the middle of the range
2. If this element is our target, success!
3. If element is less than our target, do **binary search** to the right
4. If element is greater than our target, do **binary search** to the left

DEMO: Binary Search

Solution

```
int binarySearchHelper(Vector<int>& v, int target, int start, int end) {
    if (start > end) {
        return -1;
    }
    int mid = (start + end) / 2;
    int elem = v[mid];
    if (elem == target) {
        return mid;
    } else if (elem < target) {
        return binarySearchHelper(v, target, mid + 1, end);
    } else {
        return binarySearchHelper(v, target, start, mid - 1);
    }
}
```

👥 Was This Efficient?

- How many elements did we have to check to find the answer?
- How many elements would we have to look at if we did a linear search (scanning from beginning to end)?

-1	2	6	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

👥 Was This Efficient?

- How many elements did we have to check to find the answer?
- How many elements would we have to look at if we did a linear search (scanning from beginning to end)?

-1	2	6	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

🎟️ *So, is binary search more efficient than linear search?*

Runtime of Binary Search

- Must be faster than O(n), since it's faster than a linear search
- Must be slower than O(1), since it takes longer for larger input sizes

Constant	Logarithmic	Linear	n log n	Quadratic	Polynomial	Exponential
$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^k)$ $k \geq 1$	$O(a^n)$ $a > 1$

Runtime of Binary Search

- Must be faster than O(n), since it's faster than a linear search
- Must be slower than O(1), since it takes longer for larger input sizes

Constant	Logarithmic	Linear	n log n	Quadratic	Polynomial	Exponential
O(1)	**O(log n)**	O(n)	O(n log n)	$O(n^2)$	$O(n^k)$ $k \geq 1$	$O(a^n)$ $a > 1$

What does a logarithmic runtime look like? 🪵

Runtime of Binary Search

- We're searching through N elements and eliminating half until only one element remains:
  - 1000 pages → 500 pages → 250 pages → … → 1 page
  - $N \rightarrow N/2 \rightarrow N/4 \rightarrow \ldots \rightarrow 2 \rightarrow 1$

Runtime of Binary Search

- We're searching through N elements and eliminating half until only one element remains:
  - 1000 pages $\rightarrow$ 500 pages $\rightarrow$ 250 pages $\rightarrow$ … $\rightarrow$ 1 page
  - N $\rightarrow$ N/2 $\rightarrow$ N/4 $\rightarrow$ … $\rightarrow$ 2 $\rightarrow$ 1
- How many steps does it take to get from N down to 1?
  - Let's think of it this way: 1 $\rightarrow$ 2 $\rightarrow$ … $\rightarrow$ N/4 $\rightarrow$ N/2 $\rightarrow$ N
  - Number of times we multiply by 2 to get to N

Runtime of Binary Search

- We're searching through N elements and eliminating half until only one element remains:

If x is the number of times we multiply by 2 to get N…

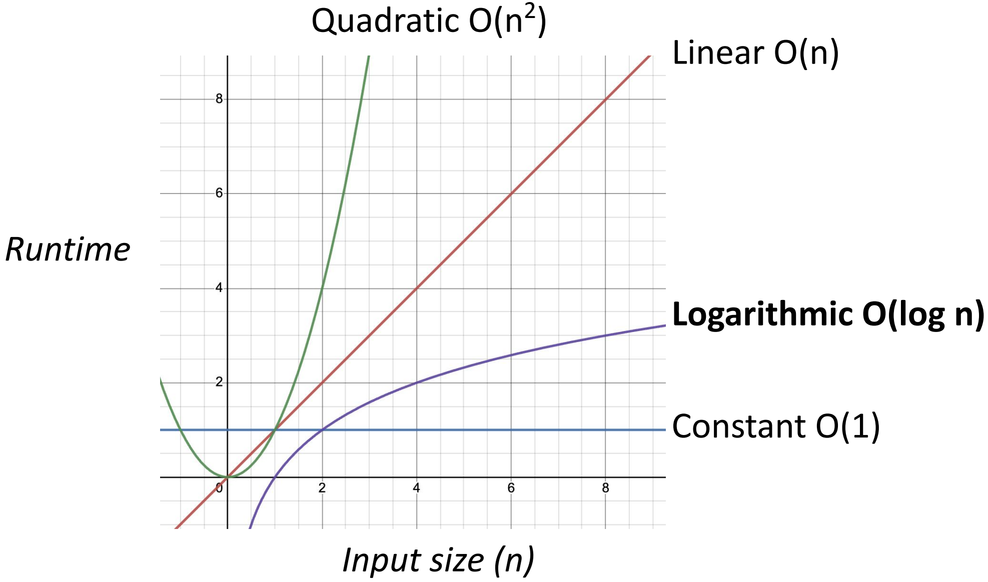
$$2^x = N$$

$$x = \log_2 N$$

-
  - Let's think of it this way: 1 → 2 → … → N/4 → N/2 → N
  - Number of times we multiply by 2 to get to N

Runtime of Binary Search

- Binary search has runtime O(log n)
  - Common runtime for algorithms that halve search space at every step



🎟️ *Binary search is more efficient than linear search, in terms of its Big-O runtime.*

DEMO: Binary Search Time Trials

Big-O of ADT Operations

Vectors

- `.size() - O(1)`
- `.add() - O(1)`
- `v[i] - O(1)`
- `.insert() - O(n)`
- `.remove() - O(n)`
- `.sublist() - O(n)`
- `traversal - O(n)`

Grids

- `.numRows() - O(1)`
- `.numCols() - O(1)`
- `grid[i][j] - O(1)`
- `.inBounds() - O(1)`
- `traversal` - $O(n^2)$

Queues

- `.size() - O(1)`
- `.peek() - O(1)`
- `.enqueue() - O(1)`
- `.dequeue() - O(1)`
- `.isEmpty() - O(1)`
- `traversal - O(n)`

Stacks

- `.size() - O(1)`
- `.peek() - O(1)`
- `.push() - O(1)`
- `.pop() - O(1)`
- `.isEmpty() - O(1)`
- `traversal - O(n)`

Sets

- `.size() - O(1)`
- `.isEmpty() - O(1)`
- **`.add() - ???`**
- **`.remove() - ???`**
- **`.contains() ???`**
- `traversal - O(n)`

Maps

- `.size() - O(1)`
- `.isEmpty() - O(1)`
- **`m[key] - ???`**
- **`.contains() - ???`**
- `traversal - O(n)`

Big-O of ADT Operations

Vectors

- `.size()` - O(1)
- `.add()` - O(1)
- `v[i]` - O(1)
- `.insert()` - O(n)
- `.remove()` - O(n)
- `.sublist()` - O(n)
- `traversal` - O(n)

Grids

- `.numRows()` - O(1)
- `.numCols()` - O(1)
- `grid[i][j]` - O(1)
- `.inBounds()` - O(1)
- `traversal` - O(n^2)

Queues

- `.size()` - O(1)
- `.peek()` - O(1)
- `.enqueue()` - O(1)
- `.dequeue()` - O(1)
- `.isEmpty()` - O(1)
- `traversal` - O(n)

Stacks

- `.size()` - O(1)
- `.peek()` - O(1)
- `.push()` - O(1)
- `.pop()` - O(1)
- `.isEmpty()` - O(1)
- `traversal` - O(n)

Sets

- `.size()` - O(1)
- `.isEmpty()` - O(1)
- **`.add()` - O(log n)**
- **`.remove()` - O(log n)**
- **`.contains()` - O(log n)**
- `traversal` - O(n)

Maps

- `.size()` - O(1)
- `.isEmpty()` - O(1)
- **`m[key]` - O(log n)**
- **`.contains()` - O(log n)**
- `traversal` - O(n)

Big-O of ADT Operations

Vectors

- `.size() – O(1)`
- `.add() – O(1)`
- `v[i] – O(1)`
- `.i`
- `.r`
- `.s`
- `tr`

Grids

- `.n`
- `.n`
- `gr`
- `.inBounds() – O(1)`
- `traversal – O(n²)`

Queues

- `.size() – O(1)`
- `.peek() – O(1)`
- `.enqueue() – O(1)`

- `.push() – O(1)`
- `.pop() – O(1)`
- `.isEmpty() – O(1)`
- `traversal – O(n)`

Sets

- `.size() – O(1)`
- `.isEmpty() – O(1)`
- **`.add() – O(log n)`**
- **`.remove() – O(log n)`**
- **`.contains() – O(log n)`**
- `traversal – O(n)`

Maps

- `.size() – O(1)`
- `.isEmpty() – O(1)`
- **`m[key] – O(log n)`**
- **`.contains() – O(log n)`**
- `traversal – O(n)`

Behind the scenes, Sets and Maps use binary search to find elements!

More on that later in the course…

Big-O of Recursive Functions

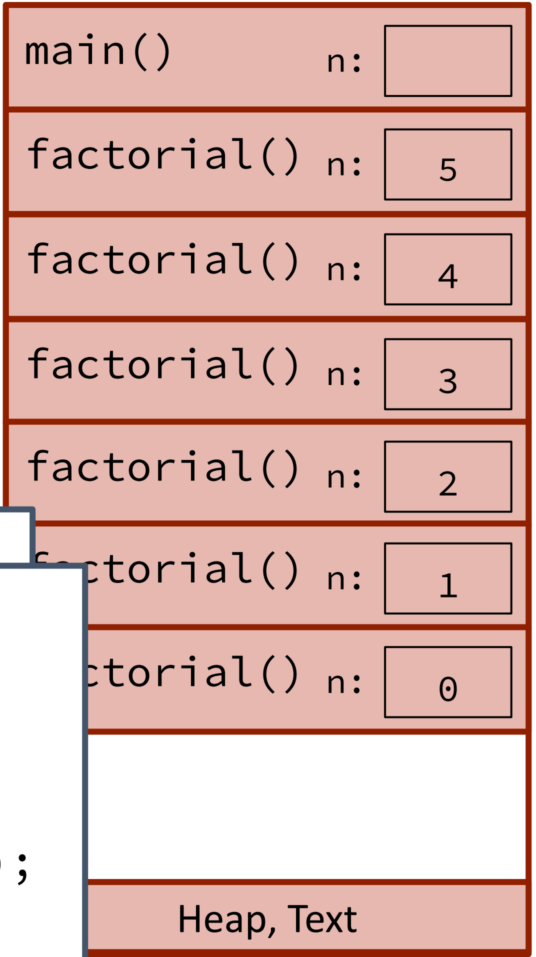
Big-O of Recursive Functions

Depends on:

- Big-O of each execution of the function
- Number of recursive calls

Factorial, Revisited

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



Factorial, Revisited

```
int main () {
```

```
int factorial (int n) {
```

What's the runtime of one call to `factorial`?

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

main()	n:	
factorial()	n:	5
factorial()	n:	4
factorial()	n:	3
factorial()	n:	2
factorial()	n:	1
factorial()	n:	0

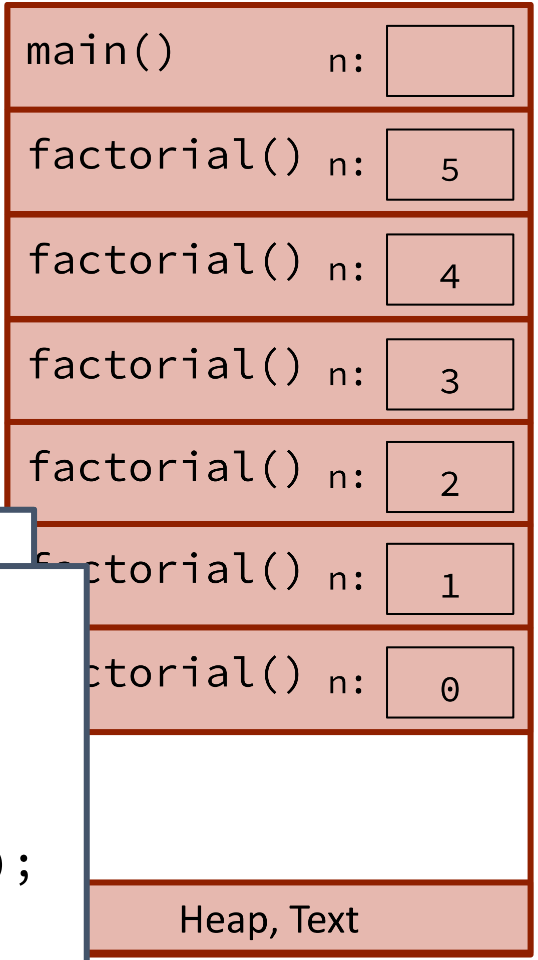
Heap, Text

Factorial, Revisited

O(1)

All of these operations (comparison, multiplication) are constant time.

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



Factorial, Revisited

How many times does `factorial` get called?

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

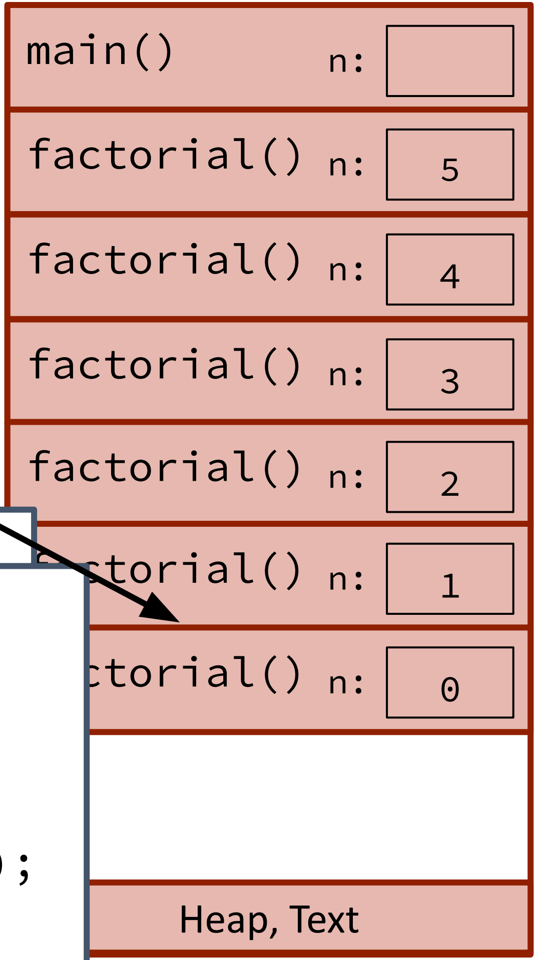
main()	n:	
factorial()	n:	5
factorial()	n:	4
factorial()	n:	3
factorial()	n:	2
factorial()	n:	1
factorial()	n:	0
Heap, Text		

Factorial, Revisited

O(n) times

We decrease n by 1 each recursive call, so it takes n recursive calls to get down to the base case.

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



Factorial, Revisited

What's the runtime of `factorial`?

`O(1) * O(n) = ` **`O(n)`**

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

main()	n:
factorial()	n: 5
factorial()	n: 4
factorial()	n: 3
factorial()	n: 2
factorial()	n: 1
factorial()	n: 0

Heap, Text

Recap

- Tower of Hanoi: Elegant
  - Recursive approach is much cleaner than the iterative one
- Binary search: Efficient
  - Allows us to find elements from a sorted collection in O(log n) time
- Calculating Big-O of recursive functions
  - Think about the function's runtime and the number of recursive calls

See you tomorrow (optionally)!

📚💻🖨️☕📖📓📚📝✍️📜