

# Big O Notation

# Objectives

- Motivate the need for something like Big O Notation
- Describe what Big O Notation is
- Simplify Big O Expressions
- Define "time complexity" and "space complexity"
- Evaluate the time complexity and space complexity of different algorithms using Big O Notation
- Describe what a logarithm is

# What's the idea here?

Imagine we have multiple implementations of the same function.

How can we determine which one is the "best?"

Great!

Pretty Good

Only OK

Ehhhhh

Awful

# Who Cares?

- It's important to have a precise vocabulary to talk about how our code performs
- Useful for discussing trade-offs between different approaches
- When your code slows down or crashes, identifying parts of the code that are inefficient can help us find pain points in our applications
- Less important: it comes up in interviews!

# An Example

Suppose we want to write a function that calculates the sum of all numbers from 1 up to (and including) some number  $n$ .

```
function addUpTo(n) {  
    let total = 0;  
    for (let i = 1; i <= n; i++) {  
        total += i;  
    }  
    return total;  
}
```

```
function addUpTo(n) {  
    return n * (n + 1) / 2;  
}
```

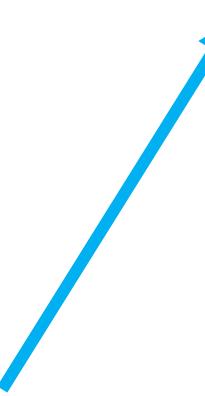
Which one is better?

# What does better mean?

- Faster?
- Less memory-intensive?
- More readable?

# What does better mean?

- Faster?
- Less memory-intensive?
- More readable?



Let's focus here first

# Why not use timers?

```
function addUpTo(n) {
  let total = 0;
  for (let i = 1; i <= n; i++) {
    total += i;
  }
  return total;
}

let t1 = performance.now();
addUpTo(1000000000);
let t2 = performance.now();
console.log(`Time Elapsed: ${ (t2 - t1) / 1000} seconds.`)
```

# The Problem with Time

- Different machines will record different times
- The *same* machine will record different times!
- For fast algorithms, speed measurements may not be precise enough?

# If not time, then what?

Rather than counting *seconds*, which are so variable...

Let's count the *number* of simple operations the computer has to perform!

# Counting Operations

```
function addUpTo(n) {  
    let total = 0;  
    for (let i = 1; i <= n; i++) {  
        total += i;  
    }  
    return total;  
}
```

1 assignment

$n$  additions

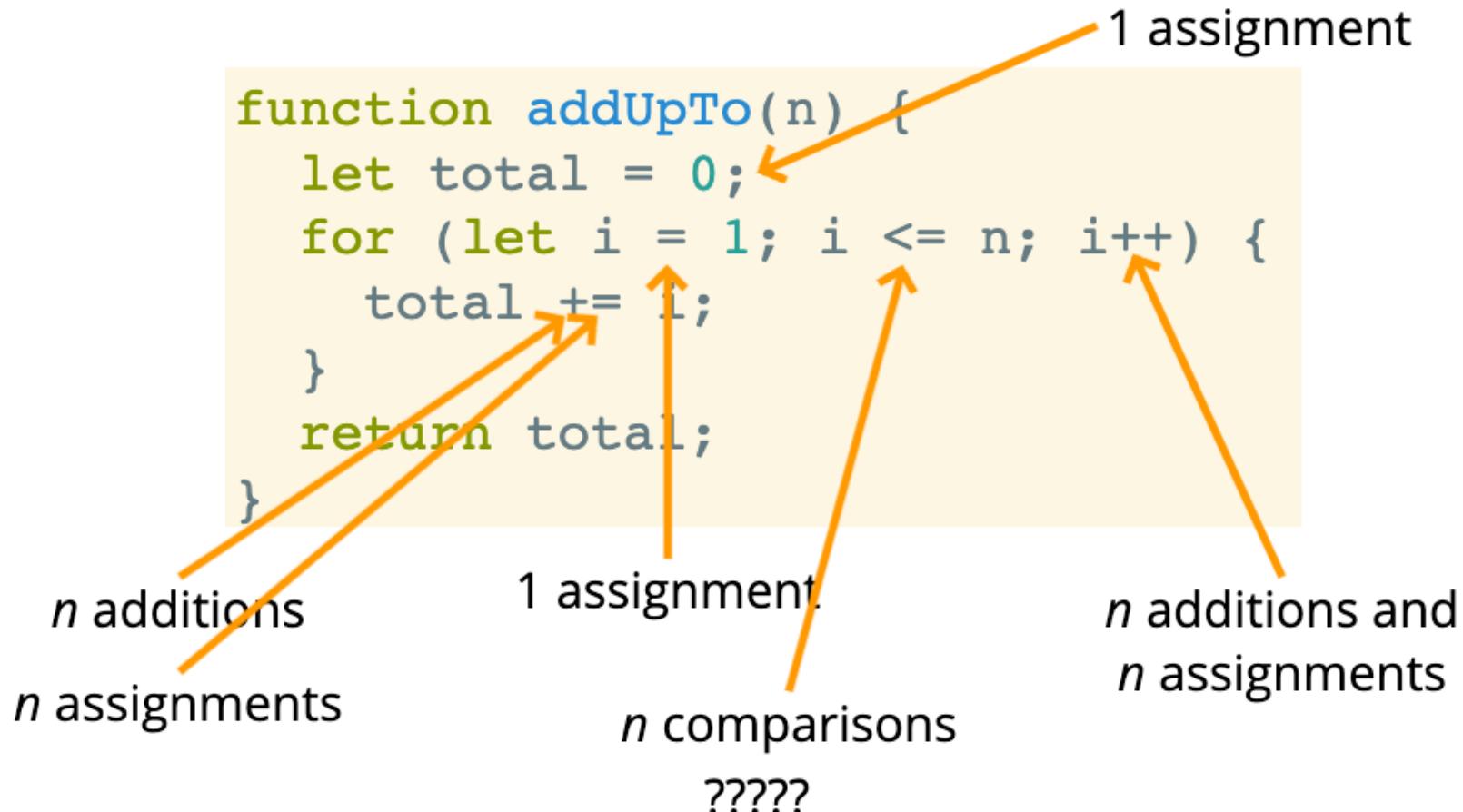
$n$  assignments

1 assignment

$n$  comparisons

?????

$n$  additions and  
 $n$  assignments



# Counting is hard!

Depending on what we count, the number of operations can be as low as  $2n$  or as high as  $5n + 2$

But regardless of the exact number, the number of operations grows roughly *proportionally with  $n$*

If  $n$  doubles, the number of operations will also roughly double

# Introducing....Big O

Big O Notation is a way to formalize fuzzy counting

It allows us to talk formally about how the runtime of an algorithm grows as the inputs grow

We won't care about the details, only the trends

# Big O Definition

We say that an algorithm is **O(f(n))** if the number of simple operations the computer has to do is eventually less than a constant times **f(n)**, as **n** increases

- $f(n)$  could be linear ( $f(n) = n$ )
- $f(n)$  could be quadratic ( $f(n) = n^2$ )
- $f(n)$  could be constant ( $f(n) = 1$ )
- $f(n)$  could be something entirely different!

# Example

```
function addUpTo(n) {  
    return n * (n + 1) / 2;  
}
```

Always 3 operations

# Example

```
function addUpTo(n) {  
    return n * (n + 1) / 2;  
}
```

Always 3 operations

**O(1)**

# Example

```
function addUpTo(n) {  
    return n * (n + 1) / 2;  
}
```

Always 3 operations

**O(1)**

```
function addUpTo(n) {  
    let total = 0;  
    for (let i = 1; i <= n; i++) {  
        total += i;  
    }  
    return total;  
}
```

Number of operations is (eventually)  
bounded by a multiple of  $n$  (say,  $10n$ )

# Example

```
function addUpTo(n) {  
    return n * (n + 1) / 2;  
}
```

Always 3 operations

**O(1)**

```
function addUpTo(n) {  
    let total = 0;  
    for (let i = 1; i <= n; i++) {  
        total += i;  
    }  
    return total;  
}
```

Number of operations is (eventually) bounded by a multiple of  $n$  (say,  $10n$ )

**O( $n$ )**

# Another Example

```
O(n)   function countUpAndDown(n) {  
O(n)     console.log("Going up!");  
O(n)     for (let i = 0; i < n; i++) {  
O(n)       console.log(i);  
O(n)     }  
O(n)     console.log("At the top!\nGoing down...");  
O(n)     for (let j = n - 1; j >= 0; j--) {  
O(n)       console.log(j);  
O(n)     }  
O(n)     console.log("Back down. Bye!");  
}
```

Number of operations is  
(eventually) bounded by a  
multiple of  $n$  (say,  $10n$ )

# Another Example

```
function countUpAndDown(n) {  
    console.log("Going up!");  
    for (let i = 0; i < n; i++) {  
        console.log(i);  
    }  
    console.log("At the top!\nGoing down...");  
    for (let j = n - 1; j >= 0; j--) {  
        console.log(j);  
    }  
    console.log("Back down. Bye!");  
}
```

Number of operations is  
(eventually) bounded by a  
multiple of  $n$  (say,  $10n$ )

**O( $n$ )**

# Another Example

```
O(n) | function printAllPairs(n) {  
O(n) |   for (var i = 0; i < n; i++) {  
O(n) |     for (var j = 0; j < n; j++) {  
O(n) |       console.log(i, j);  
O(n) |     }  
O(n) |   }  
O(n) | }
```

$O(n)$  operation inside of an  
 $O(n)$  operation.

# Another Example

```
O(n) | function printAllPairs(n) {  
O(n) |   for (var i = 0; i < n; i++) {  
O(n) |     for (var j = 0; j < n; j++) {  
O(n) |       console.log(i, j);  
O(n) |     }  
O(n) |   }  
O(n) | }
```

$O(n)$  operation inside of an  
 $O(n)$  operation.

$O(n * n)$

# Another Example

```
function printAllPairs(n) {  
    for (var i = 0; i < n; i++) {  
        O(n) | O(n) | for (var j = 0; j < n; j++) {  
            console.log(i, j);  
        }  
    }  
}
```

$O(n)$  operation inside of an  
 $O(n)$  operation.

$O(n^2)$

# Simplifying Big O Expressions

When determining the time complexity of an algorithm, there are some helpful rule of thumbs for big O expressions.

These rules of thumb are consequences of the definition of big O notation.

# Constants Don't Matter

$O(2n)$

# Constants Don't Matter

O(n)

O(n)

O(500)

# Constants Don't Matter

O~~(n)~~

O(*n*)

O~~(n<sup>2</sup>)~~

O(1)

# Constants Don't Matter

O~~(n)~~

O( $n$ )

O~~(100)~~

O(1)

O( $13n^2$ )

# Constants Don't Matter

~~O( $n$ )~~

O( $n$ )

~~O( $n^0$ )~~

O(1)

~~O( $3n^2$ )~~

O( $n^2$ )

# Smaller Terms Don't Matter

$O(n + 10)$

# Smaller Terms Don't Matter

$O(n \times 10)$

$O(n)$

# Smaller Terms Don't Matter

$O(n + 10)$

$O(n)$

$O(1000n + 50)$

# Smaller Terms Don't Matter

$O(n \times 10)$

$O(n)$

$O(10 \times n + 50)$

$O(n)$

# Smaller Terms Don't Matter

$O(n \times 10)$

$O(n)$

$O(100 \times n + 50)$

$O(n)$

$O(n^2 + 5n + 8)$

# Smaller Terms Don't Matter

$O(n \times 10)$

$O(n)$

$O(100 \times n + 50)$

$O(n)$

$O(n^2 \times 5n + 8)$

$O(n^2)$

# Big O Shorthands

1. Arithmetic operations are constant
2. Variable assignment is constant
3. Accessing elements in an array (by index) or object (by key) is constant
4. In a loop, the complexity is the length of the loop times the complexity of whatever happens inside of the loop

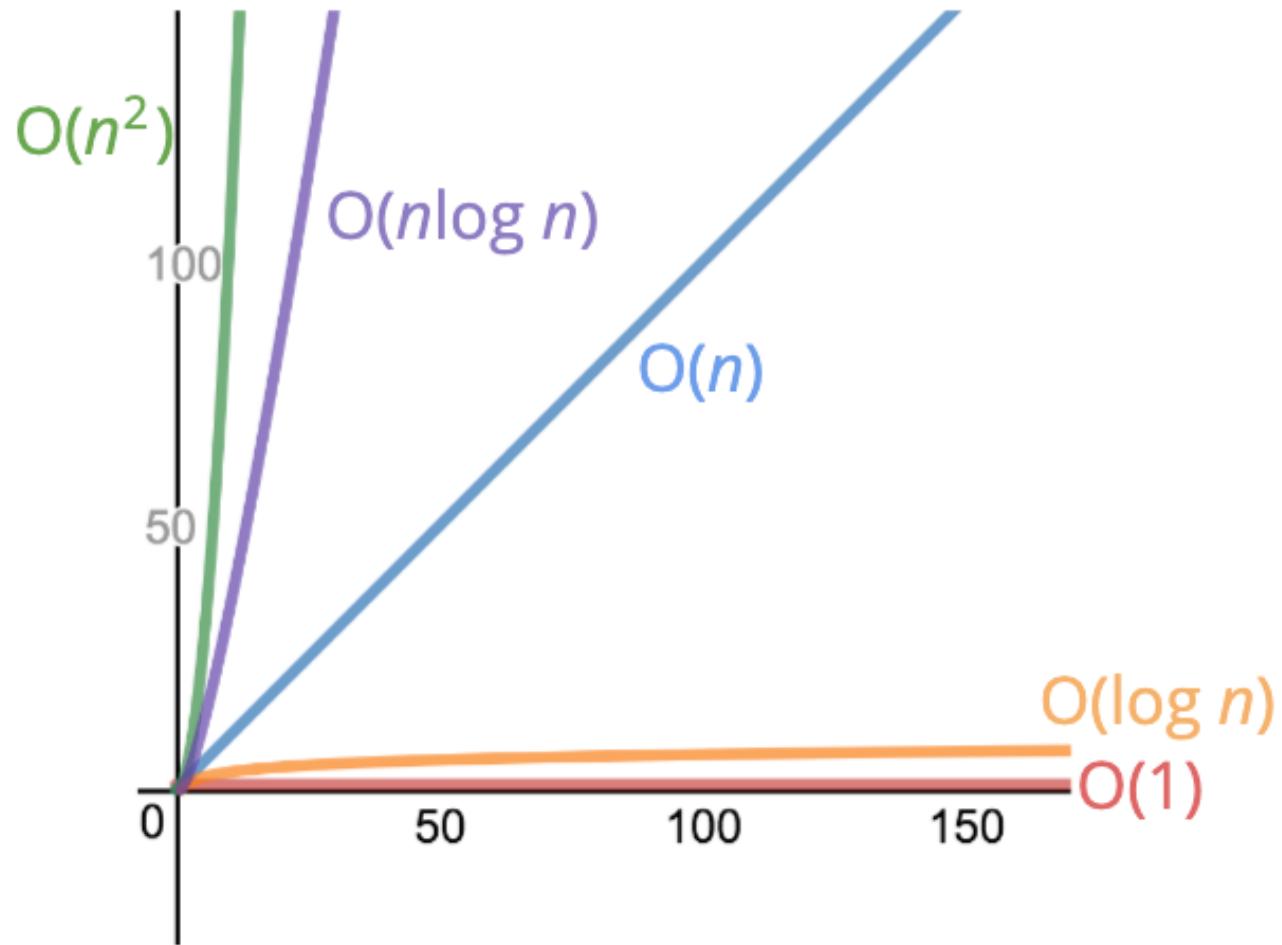
# A Couple More Examples

```
function logAtLeast5(n) {
  for (var i = 1; i <= Math.max(5, n); i++) {
    console.log(i);
  }
}
```

**O(*n*)**

```
function logAtMost5(n) {
  for (var i = 1; i <= Math.min(5, n); i++) {
    console.log(i);
  }
}
```

**O(1)**



# Space Complexity

So far, we've been focusing on **time complexity**: how can we analyze the *runtime* of an algorithm as the size of the inputs increases?

We can also use big O notation to analyze **space complexity**: how much additional memory do we need to allocate in order to run the code in our algorithm?

# What about the inputs?

Sometimes you'll hear the term **auxiliary space complexity** to refer to space required by the algorithm, not including space taken up by the inputs.

Unless otherwise noted, when we talk about space complexity, technically we'll be talking about auxiliary space complexity.

# Space Complexity in JS

- Most primitives (booleans, numbers, undefined, null) are constant space
- Strings require  $O(n)$  space (where  $n$  is the string length)
- Reference types are generally  $O(n)$ , where  $n$  is the length (for arrays) or the number of keys (for objects)

# An Example

```
function sum(arr) {  
    let total = 0;  
    for (let i = 0; i < arr.length; i++) {  
        total += arr[i];  
    }  
    return total;  
}
```

one number

another number

# An Example

```
function sum(arr) {  
    let total = 0;  
    for (let i = 0; i < arr.length; i++) {  
        total += arr[i];  
    }  
    return total;  
}
```

one number

another number

O(1) space!

# Another Example

```
function double(arr) {  
    let newArr = [];  
    for (let i = 0; i < arr.length; i++) {  
        newArr.push(2 * arr[i]);  
    }  
    return newArr;  
}
```

$n$  numbers

# Another Example

```
function double(arr) {  
    let newArr = [];  
    for (let i = 0; i < arr.length; i++) {  
        newArr.push(2 * arr[i]);  
    }  
    return newArr;  
}
```

$n$  numbers

$O(n)$  space!

**YOUR  
TURN**

# Logarithms

We've encountered some of the most common complexities:  
 $O(1)$ ,  $O(n)$ ,  $O(n^2)$

Sometimes big O expressions involve more complex mathematical expressions

One that appears more often than you might like is the logarithm!

# Wait, what's a log again?

$$\log_2(8) = 3 \longrightarrow 2^3 = 8$$

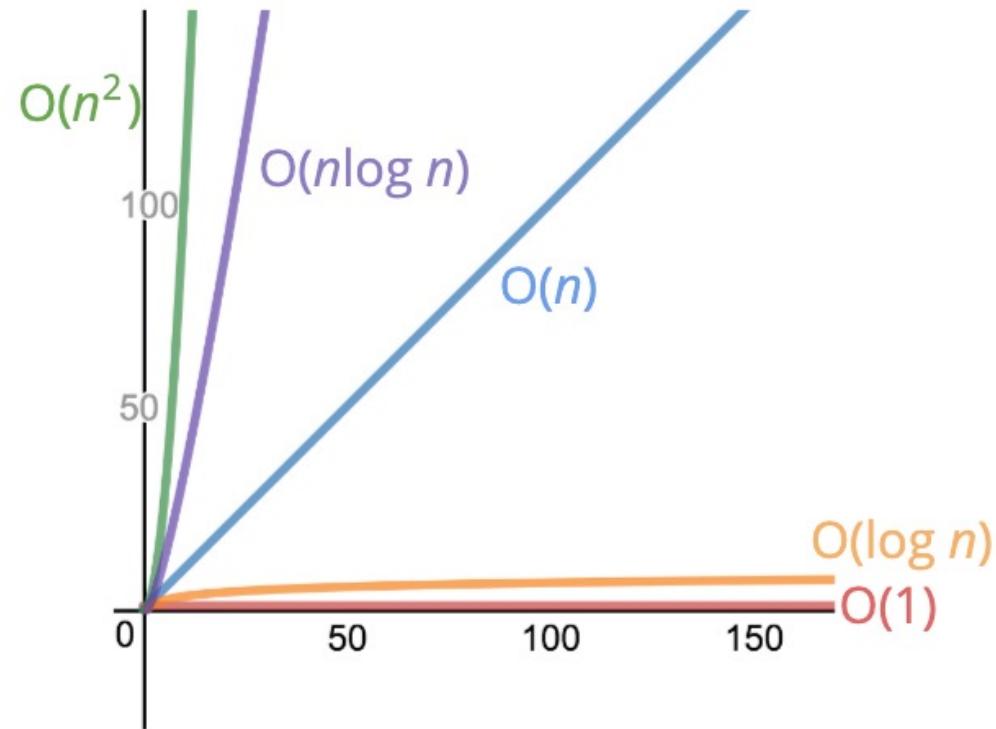
$$\log_2(\text{value}) = \text{exponent} \longrightarrow 2^{\text{exponent}} = \text{value}$$

We'll omit the 2.

$$\log == \log_2$$

# Logarithm Complexity

Logarithmic time complexity is great!



# Recap

- To analyze the performance of an algorithm, we use Big O Notation
- Big O Notation can give us a high level understanding of the time or space complexity of an algorithm
- Big O Notation doesn't care about precision, only about general trends (linear? quadratic? constant?)
- The time or space complexity (as measured by Big O) depends only on the algorithm, not the hardware used to run the algorithm
- Big O Notation is everywhere, so get lots of practice!

# Big O of Objects

Insertion - **O(1)**

Removal - **O(1)**

Searching - **O(N)**

Access - **O(1)**

**When you don't need any ordering, objects  
are an excellent choice!**

# Big O of Object Methods

`Object.keys` - **O(N)**

`Object.values` - **O(N)**

`Object.entries` - **O(N)**

`hasOwnProperty` - **O(1)**

# Big O of Arrays

Insertion - **It depends....**

Removal - **It depends....**

Searching - **O(N)**

Access - **O(1)**

# Big O of Array Operations

- push - **O(1)**
- pop - **O(1)**
- shift - **O(N)**
- unshift - **O(N)**
- concat - **O(N)**
- slice - **O(N)**
- splice - **O(N)**
- sort - **O(N \* log N)**
- forEach/map/filter/reduce/etc. - **O(N)**

# Limitations of Arrays

Inserting at the beginning is not as easy as we might think! There are **more efficient** data structures for that!

# Algorithms and Problem Solving Patterns

# PROBLEM SOLVING

- Understand the Problem
- Explore Concrete Examples
- Break It Down
- Solve/Simplify
- Look Back and Refactor

Note: many of these strategies are adapted from [George Polya](#), whose book *How To Solve It* is a great resource for anyone who wants to become a better problem solver

# UNDERSTAND THE PROBLEM

1. Can I restate the problem in my own words?
2. What are the inputs that go into the problem?
3. What are the outputs that should come from the solution to the problem?
4. Can the outputs be determined from the inputs? In other words, do I have enough information to solve the problem? (You may not be able to answer this question until you set about solving the problem. That's okay; it's still worth considering the question at this early stage.)
5. How should I label the important pieces of data that are a part of the problem?

# EXPLORE EXAMPLES

- Start with Simple Examples
- Progress to More Complex Examples
- Explore Examples with Empty Inputs
- Explore Examples with Invalid Inputs

**Write a function which takes in a string and returns counts of each character in the string.**

# BREAK IT DOWN

Explicitly write out the steps you need to take.

This forces you to think about the code you'll write before you write it, and helps you catch any lingering conceptual issues or misunderstandings before you dive in and have to worry about details (e.g. language syntax) as well.

# SOLVE THE PROBLEM

If you can't...

# SOLVE A SIMPLER PROBLEM!

# SIMPLIFY

- Find the core difficulty in what you're trying to do
- Temporarily ignore that difficulty
- Write a simplified solution
- Then incorporate that difficulty back in

# LOOK BACK & REFACTOR

Congrats on solving it, but you're not done!

# REFACTORING QUESTIONS

- Can you check the result?
- Can you derive the result differently?
- Can you understand it at a glance?
- Can you use the result or method for some other problem?
- Can you improve the performance of your solution?
- Can you think of other ways to refactor?
- How have other people solved this problem?

# RECAP!

- Understand the Problem
- Explore Concrete Examples
- Break It Down
- Solve/Simplify
- Look Back and Refactor

# HOW DO YOU IMPROVE?

1. **Devise** a plan for solving problems
2. **Master** common problem solving patterns

# HOW DO YOU IMPROVE?

1. **Devise** a plan for solving problems 
2. **Master** common problem solving patterns

# HOW DO YOU IMPROVE?

1. **Devise** a plan for solving  
problems 

2. **Master** common problem  
solving patterns

Next Up!!!

# SOME PATTERNS...

- Frequency Counter
- Multiple Pointers
- Sliding Window
- Divide and Conquer
- Dynamic Programming
- Greedy Algorithms
- Backtracking
- **Many more!**

# FREQUENCY COUNTERS

This pattern uses objects or sets to collect values/frequencies of values

This can often avoid the need for nested loops or  $O(N^2)$  operations with arrays / strings

## AN EXAMPLE

Write a function called **same**, which accepts two arrays. The function should return true if every value in the array has it's corresponding value squared in the second array. The frequency of values must be the same.

```
same([1,2,3], [4,1,9]) // true
same([1,2,3], [1,9]) // false
same([1,2,1], [4,4,1]) // false (must be same frequency)
```

# A NAIVE SOLUTION

```
function same(arr1, arr2){  
    if(arr1.length !== arr2.length){  
        return false;  
    }  
    for(let i = 0; i < arr1.length; i++){  
        let correctIndex = arr2.indexOf(arr1[i] ** 2)  
        if(correctIndex === -1) {  
            return false;  
        }  
        arr2.splice(correctIndex, 1)  
    }  
    return true  
}
```

**Time Complexity -  $N^2$**

# REFACTORED

```
function same(arr1, arr2){  
    if(arr1.length !== arr2.length){  
        return false;  
    }  
    let frequencyCounter1 = {}  
    let frequencyCounter2 = {}  
    for(let val of arr1){  
        frequencyCounter1[val] = (frequencyCounter1[val] || 0) + 1  
    }  
    for(let val of arr2){  
        frequencyCounter2[val] = (frequencyCounter2[val] || 0) + 1  
    }  
    for(let key in frequencyCounter1){  
        if(!(key ** 2 in frequencyCounter2)){  
            return false  
        }  
        if(frequencyCounter2[key ** 2] !== frequencyCounter1[key]){  
            return false  
        }  
    }  
    return true
```

**Time Complexity - O(n)**

# ANAGRAMS

Given two strings, write a function to determine if the second string is an anagram of the first. An anagram is a word, phrase, or name formed by rearranging the letters of another, such as *cinema*, formed from *iceman*.

```
validAnagram(' ', '') // true
validAnagram('aaz', 'zaa') // false
validAnagram('anagram', 'nagaram') // true
validAnagram("rat","car") // false) // false
validAnagram('awesome', 'awesom') // false
validAnagram('qwerty', 'qeywrt') // true
validAnagram('texttwisttime', 'timetwisttext') // true
```

YOUR  
TURN

# MULTIPLE POINTERS

Creating **pointers** or values that correspond to an index or position and move towards the beginning, end or middle based on a certain condition

**Very** efficient for solving problems with minimal space complexity as well

## AN EXAMPLE

Write a function called **sumZero** which accepts a **sorted** array of integers. The function should find the **first** pair where the sum is 0. Return an array that includes both values that sum to zero or undefined if a pair does not exist

```
sumZero([-3,-2,-1,0,1,2,3]) // [-3,3]
sumZero([-2,0,1,3]) // undefined
sumZero([1,2,3]) // undefined
```

# NAIVE SOLUTION

```
function sumZero(arr){  
    for(let i = 0; i < arr.length; i++){  
        for(let j = i+1; j < arr.length; j++){  
            if(arr[i] + arr[j] === 0){  
                return [arr[i], arr[j]];  
            }  
        }  
    }  
}
```

**Time Complexity -  $O(N^2)$**

**Space Complexity -  $O(1)$**

# REFACTOR

```
function sumZero(arr){  
    let left = 0;  
    let right = arr.length - 1;  
    while(left < right){  
        let sum = arr[left] + arr[right];  
        if(sum === 0){  
            return [arr[left], arr[right]];  
        } else if(sum > 0){  
            right--;  
        } else {  
            left++;  
        }  
    }  
}
```

**Time Complexity - O(N)**  
**Space Complexity - O(1)**

# countUniqueValues

Implement a function called **countUniqueValues**, which accepts a sorted array, and counts the unique values in the array. There can be negative numbers in the array, but it will always be sorted.

```
countUniqueValues([1,1,1,1,1,2]) // 2
countUniqueValues([1,2,3,4,4,4,7,7,12,12,13]) // 7
countUniqueValues([]) // 0
countUniqueValues([-2,-1,-1,0,1]) // 4
```

**YOUR  
TURN**

# SLIDING WINDOW

This pattern involves creating a **window** which can either be an array or number from one position to another

Depending on a certain condition, the window either increases or closes (and a new window is created)

Very useful for keeping track of a subset of data in an array/string etc.

# An Example

Write a function called `maxSubarraySum` which accepts an array of integers and a number called `n`. The function should calculate the maximum sum of `n` consecutive elements in the array.

```
maxSubarraySum([1,2,5,2,8,1,5],2) // 10
maxSubarraySum([1,2,5,2,8,1,5],4) // 17
maxSubarraySum([4,2,1,6],1) // 6
maxSubarraySum([4,2,1,6,2],4) // 13
maxSubarraySum([],4) // null
```

# A naive solution

```
function maxSubarraySum(arr, num) {  
    if ( num > arr.length){  
        return null;  
    }  
    var max = -Infinity;  
    for (let i = 0; i < arr.length - num + 1; i ++){  
        temp = 0;  
        for (let j = 0; j < num; j++){  
            temp += arr[i + j];  
        }  
        if (temp > max) {  
            max = temp;  
        }  
    }  
    return max;  
}
```

**Time Complexity - O(N^2)**

# Refactor

```
function maxSubarraySum(arr, num){  
    let maxSum = 0;  
    let tempSum = 0;  
    if (arr.length < num) return null;  
    for (let i = 0; i < num; i++) {  
        maxSum += arr[i];  
    }  
    tempSum = maxSum;  
    for (let i = num; i < arr.length; i++) {  
        tempSum = tempSum - arr[i - num] + arr[i];  
        maxSum = Math.max(maxSum, tempSum);  
    }  
    return maxSum;  
}
```

**Time Complexity - O(N)**

# Divide and Conquer

This pattern involves dividing a data set into smaller chunks and then repeating a process with a subset of data.

This pattern can tremendously **decrease time complexity**

# An Example

Given a **sorted** array of integers, write a function called search, that accepts a value and returns the index where the value passed to the function is located. If the value is not found, return -1

```
search([1,2,3,4,5,6],4) // 3
search([1,2,3,4,5,6],6) // 5
search([1,2,3,4,5,6],11) // -1
```

# A naive solution

```
function search(arr, val){  
    for(let i = 0; i < arr.length; i++){  
        if(arr[i] === val){  
            return i;  
        }  
    }  
    return -1;  
}
```

Linear Search

**Time Complexity O(N)**

# Refactor

```
function search(array, val) {  
  
    let min = 0;  
    let max = array.length - 1;  
  
    while (min <= max) {  
        let middle = Math.floor((min + max) / 2);  
        let currentElement = array[middle];  
  
        if (array[middle] < val) {  
            min = middle + 1;  
        }  
        else if (array[middle] > val) {  
            max = middle - 1;  
        }  
        else {  
            return middle;  
        }  
    }  
  
    return -1;  
}
```

**Time Complexity - Log(N) - Binary Search!**

# Recap

- Developing a problem solving approach is incredibly important
- Thinking about code before writing code will always make you solve problems faster
- Be mindful about problem solving patterns
- Using frequency counters, multiple pointers, sliding window and divide and conquer will help you reduce time and space complexity and help solve more challenging problems

## What is recursion?

A **process** (a function in our case)  
that **calls itself**

Why do I need  
to know this?

# It's EVERYWHERE!

- `JSON.parse` / `JSON.stringify`
- `document.getElementById` and DOM traversal algorithms
- Object traversal
- Very common with more complex algorithms
- It's sometimes a cleaner alternative to iteration

# How recursive functions work

Invoke the **same** function with a different input until you reach your base case!

## Base Case

The condition when the recursion ends.

**This is the most important concept to understand**

# Two essential parts of a recursive function!

- Base Case

# Two essential parts of a recursive function!

- Base Case
- Different Input

# Our first recursive function

```
function sumRange(num){  
    if(num === 1) return 1;  
    return num + sumRange(num-1);  
}
```

**Can you spot the base case?  
Do you notice the different input?  
What would happen if we didn't  
return?**

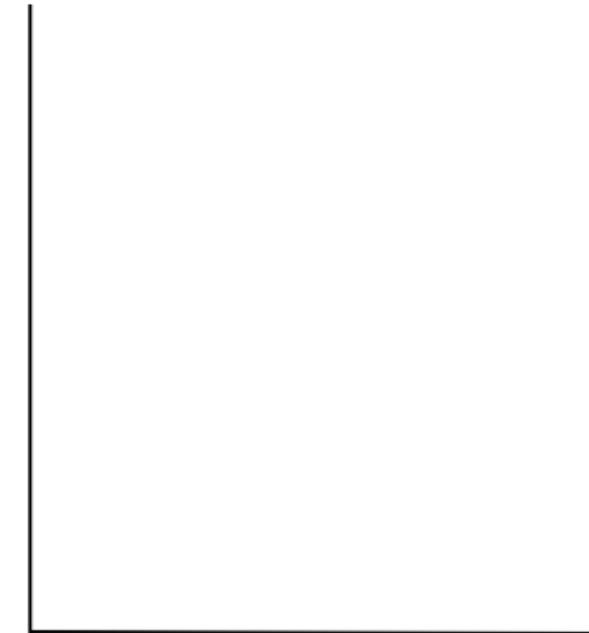
# The ALL important `return` keyword

```
function sumRange(num){  
    if(num === 1) return 1;  
    return num + sumRange(num-1);  
}
```

Let's break this down step by step!

# sumRange with the call stack

```
function sumRange(num){  
    if(num === 1) return 1;  
    return num + sumRange(num-1);  
}  
  
sumRange(5)
```



# sumRange with the call stack

```
function sumRange(num){  
    if(num === 1) return 1;  
    return num + sumRange(num-1);  
}  
  
sumRange(5)
```

sumRange(5)

# sumRange with the call stack

```
function sumRange(num){  
    if(num === 1) return 1;  
    return num + sumRange(num-1);  
}  
  
sumRange(5)
```

sumRange(4)  
sumRange(5)

# sumRange with the call stack

```
function sumRange(num){  
    if(num === 1) return 1;  
    return num + sumRange(num-1);  
}  
  
sumRange(5)
```

sumRange(3)  
sumRange(4)  
sumRange(5)

# sumRange with the call stack

```
function sumRange(num){  
    if(num === 1) return 1;  
    return num + sumRange(num-1);  
}  
  
sumRange(5)
```

sumRange(2)  
sumRange(3)  
sumRange(4)  
sumRange(5)

# sumRange with the call stack

```
function sumRange(num){  
    if(num === 1) return 1;  
    return num + sumRange(num-1);  
}  
  
sumRange(5)
```

sumRange(1)  
sumRange(2)  
sumRange(3)  
sumRange(4)  
sumRange(5)

# Another example

```
function factorial(num){  
    if(num === 1) return 1;  
    return num * factorial(num-1);  
}
```

Let's visualize the call stack!

# Where things go wrong

- No base case
- Forgetting to return or returning the wrong thing!
- Stack overflow!

```
function factorial(num){  
    if(num === 1) return 1;  
    return num * factorial(num);  
}
```

```
function factorial(num){  
    if(num === 1) console.log(1) ;  
    return num * factorial(num-1);  
}
```

# HELPER METHOD RECURSION

```
function outer(input){  
  
    var outerScopedVariable = []  
  
    function helper(helperInput){  
        // modify the outerScopedVariable  
        helper(helperInput--)  
    }  
  
    helper(input)  
  
    return outerScopedVariable;  
}
```

# ANOTHER EXAMPLE

Let's try to collect all of the odd values in an array!

```
function collectOddValues(arr){  
  let result = []  
  
  function helper(helperInput){  
    if(helperInput.length === 0) {  
      return;  
    }  
  
    if(helperInput[0] % 2 !== 0){  
      result.push(helperInput[0])  
    }  
  
    helper(helperInput.slice(1))  
  }  
  
  helper(arr)  
  
  return result;  
}
```

# PURE RECURSION

```
function collectOddValues(arr){  
  let newArr = [ ];  
  
  if(arr.length === 0) {  
    return newArr;  
  }  
  
  if(arr[0] % 2 !== 0){  
    newArr.push(arr[0]);  
  }  
  
  newArr = newArr.concat(collectOddValues(arr.slice(1)));  
  return newArr;  
}
```

# Pure Recursion Tips

- For arrays, use methods like **slice**, **the spread operator**, and **concat** that make copies of arrays so you do not mutate them
- Remember that strings are immutable so you will need to use methods like **slice**, **substr**, or **substring** to make copies of strings
- To make copies of objects use **Object.assign**, or **the spread operator**

# What about big O?

- Measuring **time complexity** is relatively simple. You can measure the time complexity of a recursive function as the number of recursive calls you need to make relative to the input

# What about big O?

- Measuring **time complexity** is relatively simple. You can measure the time complexity of a recursive function as the number of recursive calls you need to make relative to the input
- Measuring **space complexity** is a bit more challenging. You can measure the space complexity of a recursive function as the **maximum** number of functions on the call stack at a given time, since the call stack requires memory.

# Tail Call Optimization

- ES2015 allows for *tail call optimization*, where you can make some function calls without growing the call stack.
- By using the **return** keyword in a specific fashion we can extract output from a function without keeping it on the call stack.
- Unfortunately this has not been implemented across multiple browsers so it is not reasonable to implement in production code.

# Recap

- A recursive function is a function that invokes itself
- Your recursive functions should **always** have a base case and be invoked with different input each time
- When using recursion, it's often essential to return values from one function to another to extract data from each function call
- Helper method recursion is an alternative that allows us to use an external scope in our recursive functions
- Pure recursion eliminates the need for helper method recursion, but can be trickier to understand at first

# Searching Algorithms

# Objectives

- Describe what a searching algorithm is
- Implement linear search on arrays
- Implement binary search on sorted arrays
- Implement a naive string searching algorithm
- Implement the KMP string searching algorithm

# How do we search?

Given an array, the simplest way to search for a value is to look at every element in the array and check if it's the value we want.

# JavaScript has search!

There are many different search methods on arrays in JavaScript:

- indexOf
- includes
- Find
- findIndex

But how do these functions work?

# Linear Search

Let's search for 12:

```
[ 5, 8, 1, 100, 12, 3, 12 ]
```

# Linear Search

Let's search for 12:

```
[ 5, 8, 1, 100, 12, 3, 12 ]  
↑  
Not 12
```

# Linear Search

Let's search for 12:

```
[ 5, 8, 1, 100, 12, 3, 12 ]  
↑  
Not 12
```

# Linear Search

Let's search for 12:

[ 5, 8, 1, 100, 12, 3, 12 ]



Not 12

# Linear Search

Let's search for 12:

```
[ 5, 8, 1, 100, 12, 3, 12 ]  
          ↑  
          Not 12
```

# Linear Search

Let's search for 12:

```
[ 5, 8, 1, 100, 12, 3, 12 ]  
↑  
12!
```

# Linear Search Pseudocode

- This function accepts an array and a value
- Loop through the array and check if the current array element is equal to the value
- If it is, return the index at which the element is found
- If the value is never found, return -1

# Linear Search

## BIG O

# Linear Search

BIG O

O(1)

Best

# Linear Search

BIG O

$O(1)$

Best

$O(n)$

Worst

# Linear Search

BIG O

$O(1)$

Best

$O(n)$

Average

$O(n)$

Worst

# Binary Search

- Binary search is a much faster form of search
- Rather than eliminating one element at a time, you can eliminate *half* of the remaining elements at a time
- Binary search only works on *sorted* arrays!

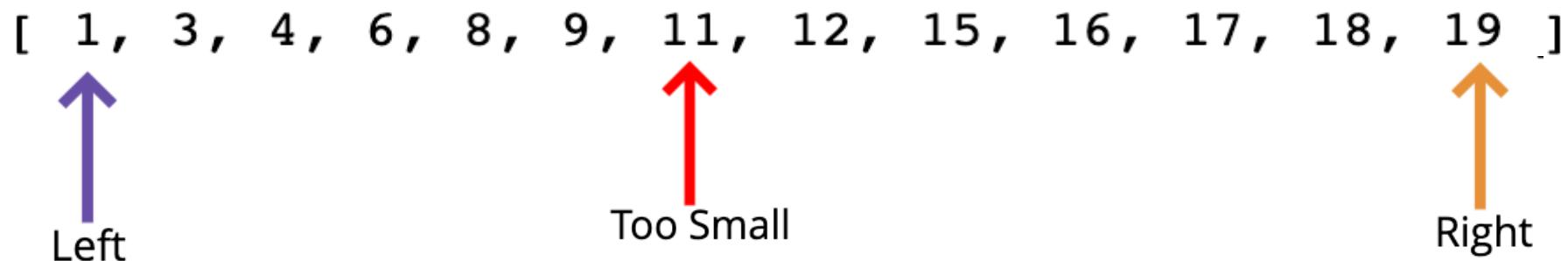
# Divide and Conquer

Let's search for 15:

```
[ 1, 3, 4, 6, 8, 9, 11, 12, 15, 16, 17, 18, 19 ]
```

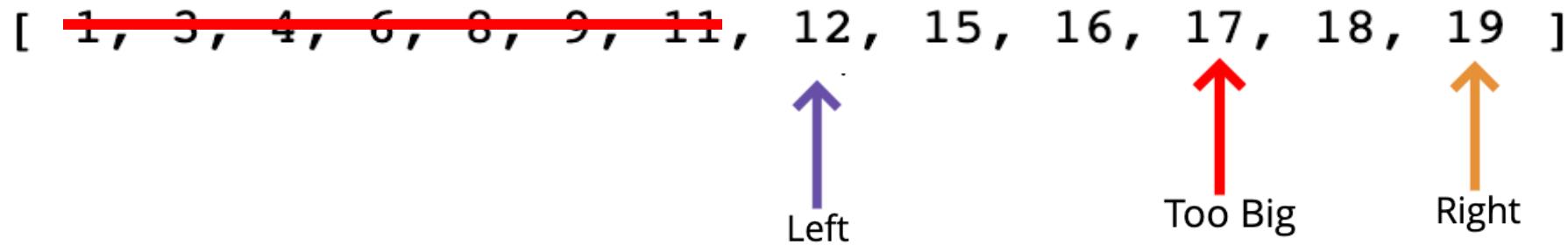
# Divide and Conquer

Let's search for 15:



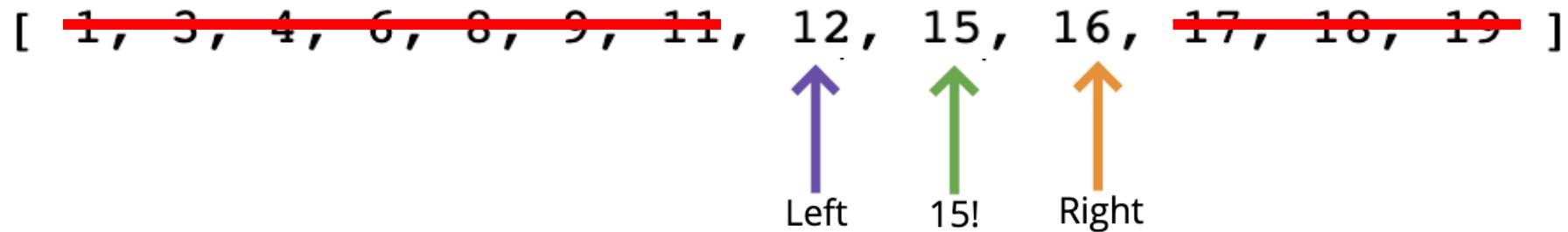
# Divide and Conquer

Let's search for 15:



# Divide and Conquer

Let's search for 15:



# Binary Search Pseudocode

- This function accepts a sorted array and a value
- Create a left pointer at the start of the array, and a right pointer at the end of the array
- While the left pointer comes before the right pointer:
  - Create a pointer in the middle
  - If you find the value you want, return the index
  - If the value is too small, move the left pointer up
  - If the value is too large, move the right pointer down
- If you never find the value, return -1

YOUR  
TURN

NOW LET'S DO IT  
RECURSIVELY!

# WHAT ABOUT BIG O?

$O(\log n)$

Worst and Average Case

$O(1)$

Best Case

Suppose we're searching for 13

[2,4,5,9,11,14,15,**19**,21,25,28,30,50,52,60,63]

Suppose we're searching for 13

[2,4,5,9,11,14,15,**19**,21,25,28,30,50,52,60,63]

[2,4,5,**9**,11,14,15]

Suppose we're searching for 13

[2,4,5,9,11,14,15,**19**,21,25,28,30,50,52,60,63]

[2,4,5,**9**,11,14,15]

[11,**14**,15]

Suppose we're searching for 13

[2,4,5,9,11,14,15,**19**,21,25,28,30,50,52,60,63]

[2,4,5,**9**,11,14,15]

[11,**14**,15]

[**11**]

Suppose we're searching for 13

[2,4,5,9,11,14,15,**19**,21,25,28,30,50,52,60,63]

[2,4,5,**9**,11,14,15]

[11,**14**,15]

[**11**]

NOPE, NOT HERE!

16 elements = 4 "steps"

To add another "step", we need to double the number of elements

Let's search for 32:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

To add another "step", we need to double the number of elements

Let's search for 32:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,**16**,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

To add another "step", we need to double the number of elements

Let's search for 32:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

~~[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,~~16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

To add another "step", we need to double the number of elements

Let's search for 32:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16~~,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16~~,17,18,19,20,21,22,23,~~24~~,25,26,27,28,29,30,32,35]

To add another "step", we need to double the number of elements

Let's search for 32:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16~~,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24~~,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27~~,~~28~~,29,30,32,35]

To add another "step", we need to double the number of elements

Let's search for 32:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16~~,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24~~,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28~~,29,30,32,35]

To add another "step", we need to double the number of elements

Let's search for 32:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16~~,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24~~,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28~~,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29~~,30,32,35]

To add another "step", we need to double the number of elements

Let's search for 32:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16~~,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24~~,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28~~,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29~~,30,32,35]

To add another "step", we need to double the number of elements

Let's search for 32:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16~~,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24~~,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28~~,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29~~,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30~~,32,35]

To add another "step", we need to double the number of elements

Let's search for 32:

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16~~,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23~~,~~24~~,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27~~,~~28~~,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29~~,~~30~~,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30~~,~~32~~,35]

32 elements = 5 "steps" (worst case)

# Naive String Search

- Suppose you want to count the number of times a smaller string appears in a longer string
- A straightforward approach involves checking pairs of characters individually

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wowomgzomg

omg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

WOWOMGZOMG

omg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wowomgzomg

omg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wOWomgzomg

omg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

WOWomgzomg

Omg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wOWOMGZOMG

OMG

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wOWOMGZOMG

omg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wOWomgzomg

omg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wOWOMGZOMG

omg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wowomg<sub>g</sub>zomg

omg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wowomg~~z~~omg

~~O~~mg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wowomgzomg

omg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wowomgzomg

omg

# Naive String Search

Long string:

wowomgzomg

Short string:

omg

wowomgzomg

omg

## Pseudocode

- Loop over the longer string
- Loop over the shorter string
- If the characters don't match, break out of the inner loop
- If the characters do match, keep going
- If you complete the inner loop and find a match, increment the count of matches
- Return the count

# KMP String Search

- The Knutt-Morris-Pratt algorithm offers an improvement over the naive approach
- Published in 1977
- This algorithm more intelligently traverses the longer string to reduce the amount of redundant searching

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

**lolomglolololrofl**

**lolol**

# KMP Example

Long string:

lolomglololrof lol

Short string:

lolol

**1olomglolololrof1**

**1olol**

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

**lo**lolomglolololrofl

**lo**lol

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lolomglolololrofl

lolol

# KMP Example

Long string:

lolomglololrof lol

Short string:

lolol

lolomglolololrof l

lolol

# KMP Example

Long string:

lolomglololrof lol

Short string:

lolol

lolomglolololrof l

lolol

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lolomglolololrofl

lolol

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lol~~o~~**mglolololrofl**

**1**olol

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lolomglolololrofl

lolol

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lolomg1ololololrofl

1lolol

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lolomglolololrofl

lolol

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lolomglolololrofl

lolol

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lolomglolololrofl

lolol

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lolomg**lolololrofl**

**lolol**

1

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lolomglolololrofl

lolol

1

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lolomglolololrofl

lolol

2

# KMP Example

Long string:

lolomglololrof lol

Short string:

lolol

lolomglololol**r**of l

lolol

2

# KMP Example

Long string:

lolomglololrof lol

Short string:

lolol

lolomglololol**rof**l

**1**olol

2

# KMP Example

Long string:

lolomglololroflol

Short string:

lolol

lolomglololol**r**ofl  
**l**olol

2

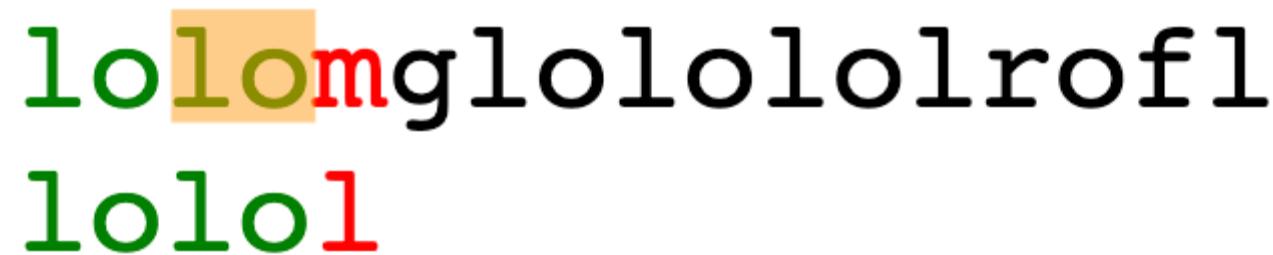
# KMP - WTF

How do you know how far to traverse?

lolol**m**glololololrofl  
**lolol**

# KMP - WTF

How do you know how far to traverse?



A diagram illustrating the KMP string matching process. The text "lolololololrofl" is at the top, and the pattern "lolol" is below it. The character 'o' in "lolol" is highlighted in green, while the character 'l' is highlighted in red. A yellow rectangular box highlights the prefix "lol" of the pattern. Below the pattern, the suffix "lol" is also highlighted in green and red.

# KMP - WTF

How do you know how far to traverse?

lololomglololololrofl  
lolol1

Find the longest (proper) suffix in the matched portion of the long string...

# KMP - WTF

How do you know how far to traverse?

lololmglo lolololrofl  
lolol

Find the longest (proper) suffix in the matched portion of the long string...

# KMP - WTF

How do you know how far to traverse?

lo **lolo**mglololololrofl

lo **lolol**1

Find the longest (proper) suffix in the matched portion of the long string...

That matches a (proper) prefix in the matched portion of the short string!

# KMP - WTF

How do you know how far to traverse?

lololomglololololrofl  
lolol

Find the longest (proper) suffix in the matched portion of the long string...

That matches a (proper) prefix in the matched portion of the short string!

Then shift the short string accordingly!

# Prefixes and Suffixes

- In order to determine how far we can shift the shorter string, we can *pre-compute* the length of the longest (proper) suffix that matches a (proper) prefix
- This tabulation should happen before you start looking for the short string in the long string

# Prefixes and Suffixes

## Example

short string

Longest prefix  
suffix match

I	o	I	o	I

# Prefixes and Suffixes

## Example

short string

Longest prefix  
suffix match

I	o	I	o	I
o				

# Prefixes and Suffixes

## Example

short string

Longest prefix  
suffix match

I	o	I	o	I
0	0	1		

# Prefixes and Suffixes

## Example

short string

Longest prefix  
suffix match

I	o	I	o	I
0	0	1	2	

# Prefixes and Suffixes

Example

short string

Longest prefix  
suffix match

I	o	I	o	I
0	0	1	2	3

# Prefixes and Suffixes

# Another Example

# Prefixes and Suffixes

## Another Example

# Prefixes and Suffixes

## Another Example

a	b	a	c	a	b	a	b	d	a
0	0								

# Prefixes and Suffixes

Another Example

a	b	a	c	a	b	a	b	d	a
0	0	1							

# Prefixes and Suffixes

Another Example

a	b	a	c	a	b	a	b	d	a
0	0	1	0						

# Prefixes and Suffixes

Another Example

a	b	a	c	a	b	a	b	d	a
0	0	1	0	1					

# Prefixes and Suffixes

Another Example

a	b	a	c	a	b	a	b	d	a
0	0	1	0	1	2				

# Prefixes and Suffixes

Another Example

a	b	a	c	a	b	a	b	d	a
0	0	1	0	1	2	3			

# Prefixes and Suffixes

Another Example

a	b	a	c	a	b	a	b	d	a
0	0	1	0	1	2	3	2		

# Prefixes and Suffixes

Another Example

a	b	a	c	a	b	a	b	d	a
0	0	1	0	1	2	3	2	0	

# Prefixes and Suffixes

Another Example

a	b	a	c	a	b	a	b	d	a
0	0	1	0	1	2	3	2	0	1

# Building the Table

```
function matchTable(word) {
  let arr = Array.from({ length: word.length }).fill(0);
  let suffixEnd = 1;
  let prefixEnd = 0;
  while (suffixEnd < word.length) {
    if (word[suffixEnd] === word[prefixEnd]) {
      // we can build a longer prefix based on this suffix
      // store the length of this longest prefix
      // move prefixEnd and suffixEnd
      prefixEnd += 1;
      arr[suffixEnd] = prefixEnd;
      suffixEnd += 1;
    } else if (word[suffixEnd] !== word[prefixEnd] && prefixEnd !== 0) {
      // there's a mismatch, so we can't build a larger prefix
      // move the prefixEnd to the position of the next largest prefix
      prefixEnd = arr[prefixEnd - 1];
    } else {
      // we can't build a proper prefix with any of the proper suffixes
      // let's move on
      arr[suffixEnd] = 0;
      suffixEnd += 1;
    }
  }
  return arr;
}
```

# KMP - FTW!

```
function kmpSearch(long, short) {
    let table = matchTable(short);
    let shortIdx = 0;
    let longIdx = 0;
    let count = 0;
    while (longIdx < long.length - short.length + shortIdx + 1) {
        if (short[shortIdx] !== long[longIdx]) {
            // we found a mismatch :(
            // if we just started searching the short, move the long pointer
            // otherwise, move the short pointer to the end of the next potential prefix
            // that will lead to a match
            if (shortIdx === 0) longIdx += 1;
            else shortIdx = table[shortIdx - 1];
        } else {
            // we found a match! shift both pointers
            shortIdx += 1;
            longIdx += 1;
            // then check to see if we've found the substring in the large string
            if (shortIdx === short.length) {
                // we found a substring! increment the count
                // then move the short pointer to the end of the next potential prefix
                count++;
                shortIdx = table[shortIdx - 1];
            }
        }
    }
    return count;
}
```

# Big O of Search Algorithms

Linear Search - **O(n)**

Binary Search - **O(log n)**

Naive String Search - **O(nm)**

KMP - **O(n + m)** time, **O(m)** space

# Recap

- Searching is a very common task that we often take for granted
- When searching through an unsorted collection, linear search is the best we can do
- When searching through a sorted collection, we can find things very quickly with binary search
- KMP provides a linear time algorithm for searches in strings

# Sorting Algorithms

# What is sorting?

Sorting is the process of rearranging the items in a collection (e.g. an array) so that the items are in some kind of order.

Examples:

- Sorting numbers from smallest to largest
- Sorting names alphabetically
- Sorting movies based on release year
- Sorting movies based on revenue

# Why do we need to learn this?

- Sorting is an incredibly common task, so it's good to know how it works
- There are many different ways to sort things, and different techniques have their own advantages and disadvantages
- Sorting sometimes has quirks, so it's good to understand how to navigate them

# JavaScript has a sort method...

Yes, it does!

...but it doesn't always work the way you expect.



```
[ "Steele", "Colt", "Data Structures", "Algorithms" ].sort();
// [ "Algorithms", "Colt", "Data Structures", "Steele" ]
```



```
[ 6, 4, 15, 10 ].sort();
// [ 10, 15, 4, 6 ]
```

# Telling JavaScript how to sort

- The built-in `sort` method accepts an optional *comparator* function
- You can use this comparator function to tell JavaScript how you want it to sort
- The comparator looks at pairs of elements (*a* and *b*), determines their sort order based on the return value
  - If it returns a negative number, *a* should come before *b*
  - If it returns a positive number, *a* should come after *b*,
  - If it returns 0, *a* and *b* are the same as far as the sort is concerned

# Telling JavaScript how to sort

## Examples

```
function numberCompare(num1, num2) {
  return num1 - num2;
}

[ 6, 4, 15, 10 ].sort(numberCompare);
// [ 4, 6, 10, 15 ]
```

```
function compareByLen(str1, str2) {
  return str1.length - str2.length;
}

[ "Steele", "Colt", "Data Structures", "Algorithms" ]
  .sort(compareByLen);
// [ "Colt", "Steele", "Algorithms", "Data Structures" ]
```

# Before we sort, we must swap!

Many sorting algorithms involve some type of swapping functionality (e.g. swapping to numbers to put them in order)

```
// ES5
function swap(arr, idx1, idx2) {
  var temp = arr[idx1];
  arr[idx1] = arr[idx2];
  arr[idx2] = temp;
}

// ES2015
const swap = (arr, idx1, idx2) => {
  [arr[idx1],arr[idx2]] = [arr[idx2],arr[idx1]];
}
```

# BubbleSort

A sorting algorithm where the largest values bubble up to the top!

```
[ 5, 3, 4, 1, 2 ]
```

# BubbleSort

A sorting algorithm where the largest values bubble up to the top!

[ 5, 3, 4, 1, 2 ]



# BubbleSort

A sorting algorithm where the largest values bubble up to the top!

[ 5, 3, 4, 1, 2 ]



[ 3, 5, 4, 1, 2 ]

# BubbleSort

A sorting algorithm where the largest values bubble up to the top!

[ 5, 3, 4, 1, 2 ]



[ 3, 5, 4, 1, 2 ]



# BubbleSort

A sorting algorithm where the largest values bubble up to the top!

[ 5, 3, 4, 1, 2 ]



[ 3, 5, 4, 1, 2 ]



[ 3, 4, 5, 1, 2 ]

# BubbleSort

A sorting algorithm where the largest values bubble up to the top!

[ 5, 3, 4, 1, 2 ]



[ 3, 5, 4, 1, 2 ]



[ 3, 4, 5, 1, 2 ]



# BubbleSort

A sorting algorithm where the largest values bubble up to the top!

[ 5, 3, 4, 1, 2 ]



[ 3, 5, 4, 1, 2 ]



[ 3, 4, 5, 1, 2 ]



[ 3, 4, 1, 5, 2 ]

# BubbleSort

A sorting algorithm where the largest values bubble up to the top!

[ 5, 3, 4, 1, 2 ]



[ 3, 5, 4, 1, 2 ]



[ 3, 4, 5, 1, 2 ]



[ 3, 4, 1, 5, 2 ]



# BubbleSort

A sorting algorithm where the largest values bubble up to the top!

[ 5, 3, 4, 1, 2 ]



[ 3, 5, 4, 1, 2 ]



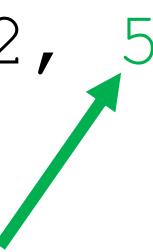
[ 3, 4, 5, 1, 2 ]



[ 3, 4, 1, 5, 2 ]



[ 3, 4, 1, 2, 5 ]



5 is now in its sorted position!

# BubbleSort Pseudocode

- Start looping from with a variable called  $i$  the end of the array towards the beginning
- Start an inner loop with a variable called  $j$  from the beginning until  $i - 1$
- If  $\text{arr}[j]$  is greater than  $\text{arr}[j+1]$ , swap those two values!
- Return the sorted array

# Selection Sort

Similar to bubble sort, but instead of first placing large values into sorted position, it places small values into sorted position

```
[ 5, 3, 4, 1, 2 ]
```

# Selection Sort

Similar to bubble sort, but instead of first placing large values into sorted position, it places small values into sorted position

[ 5, 3, 4, 1, 2 ]  
↑ ↑

# Selection Sort

Similar to bubble sort, but instead of first placing large values into sorted position, it places small values into sorted position

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]  
↑ ↑

# Selection Sort

Similar to bubble sort, but instead of first placing large values into sorted position, it places small values into sorted position

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]



# Selection Sort

Similar to bubble sort, but instead of first placing large values into sorted position, it places small values into sorted position

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]



# Selection Sort

Similar to bubble sort, but instead of first placing large values into sorted position, it places small values into sorted position

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]



[ 1, 3, 4, 5, 2 ]

# Selection Sort

Similar to bubble sort, but instead of first placing large values into sorted position, it places small values into sorted position

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]

[ 5, 3, 4, 1, 2 ]

[ 1, 3, 4, 5, 2 ]

1 is now in its sorted position!



# Selection Sort Pseudocode

- Store the first element as the smallest value you've seen so far.
- Compare this item to the next item in the array until you find a smaller number.
- If a smaller number is found, designate that smaller number to be the new "minimum" and continue until the end of the array.
- If the "minimum" is not the value (index) you initially began with, swap the two values.
- Repeat this with the next element until the array is sorted.

# Insertion Sort

Builds up the sort by gradually creating a larger left half which is always sorted

# Insertion Sort

Builds up the sort by gradually creating a larger left half which is always sorted

```
[ 5, 3, 4, 1, 2 ]
```

# Insertion Sort

Builds up the sort by gradually creating a larger left half which is always sorted

```
[ 5, 3, 4, 1, 2 ]
```

# Insertion Sort

Builds up the sort by gradually creating a larger left half which is always sorted

```
[ 5, 3, 4, 1, 2 ]
```

```
[ 3, 5, 4, 1, 2 ]
```

# Insertion Sort

Builds up the sort by gradually creating a larger left half which is always sorted

```
[ 5, 3, 4, 1, 2 ]
```

```
[ 3, 5, 4, 1, 2 ]
```

```
[ 3, 4, 5, 1, 2 ]
```

# Insertion Sort

Builds up the sort by gradually creating a larger left half which is always sorted

```
[ 5, 3, 4, 1, 2 ]
```

```
[ 3, 5, 4, 1, 2 ]
```

```
[ 3, 4, 5, 1, 2 ]
```

```
[ 1, 3, 4, 5, 2 ]
```

# Insertion Sort

Builds up the sort by gradually creating a larger left half which is always sorted

```
[ 5, 3, 4, 1, 2 ]
```

```
[ 3, 5, 4, 1, 2 ]
```

```
[ 3, 4, 5, 1, 2 ]
```

```
[ 1, 3, 4, 5, 2 ]
```

```
[ 1, 2, 3, 4, 5 ]
```

# Insertion Sort Pseudocode

- Start by picking the second element in the array
- Now compare the second element with the one before it and swap if necessary.
- Continue to the next element and if it is in the incorrect order, iterate through the sorted portion (i.e. the left side) to place the element in the correct place.
- Repeat until the array is sorted.

# Big O of Sorting Algorithms

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$

# Recap

- Sorting is *fundamental*!
- Bubble sort, selection sort, and insertion sort are all roughly equivalent
- All have average time complexities that are quadratic
- We can do better...but we need more complex algorithms!

# FASTER SORTS

- There is a family of sorting algorithms that can improve time complexity from  $O(n^2)$  to  $O(n \log n)$
- There's a tradeoff between efficiency and simplicity
- The more efficient algorithms are much less simple, and generally take longer to understand
- Let's dive in!

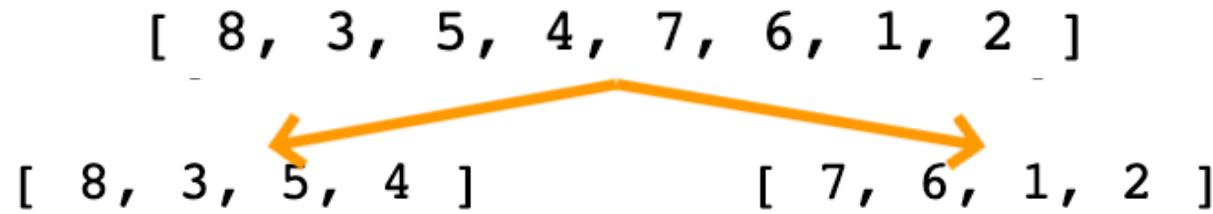
# Merge Sort

- It's a combination of two things - merging and sorting!
- Exploits the fact that arrays of 0 or 1 element are always sorted
- Works by decomposing an array into smaller arrays of 0 or 1 elements, then building up a newly sorted array

# How does it work?

```
[ 8, 3, 5, 4, 7, 6, 1, 2 ]
```

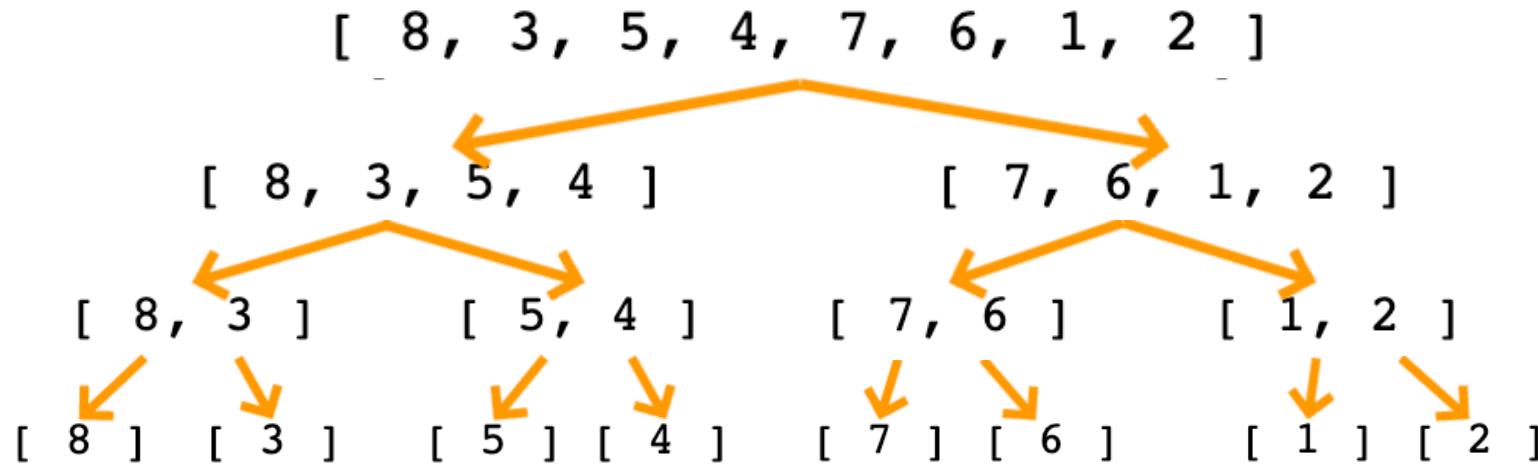
# How does it work?



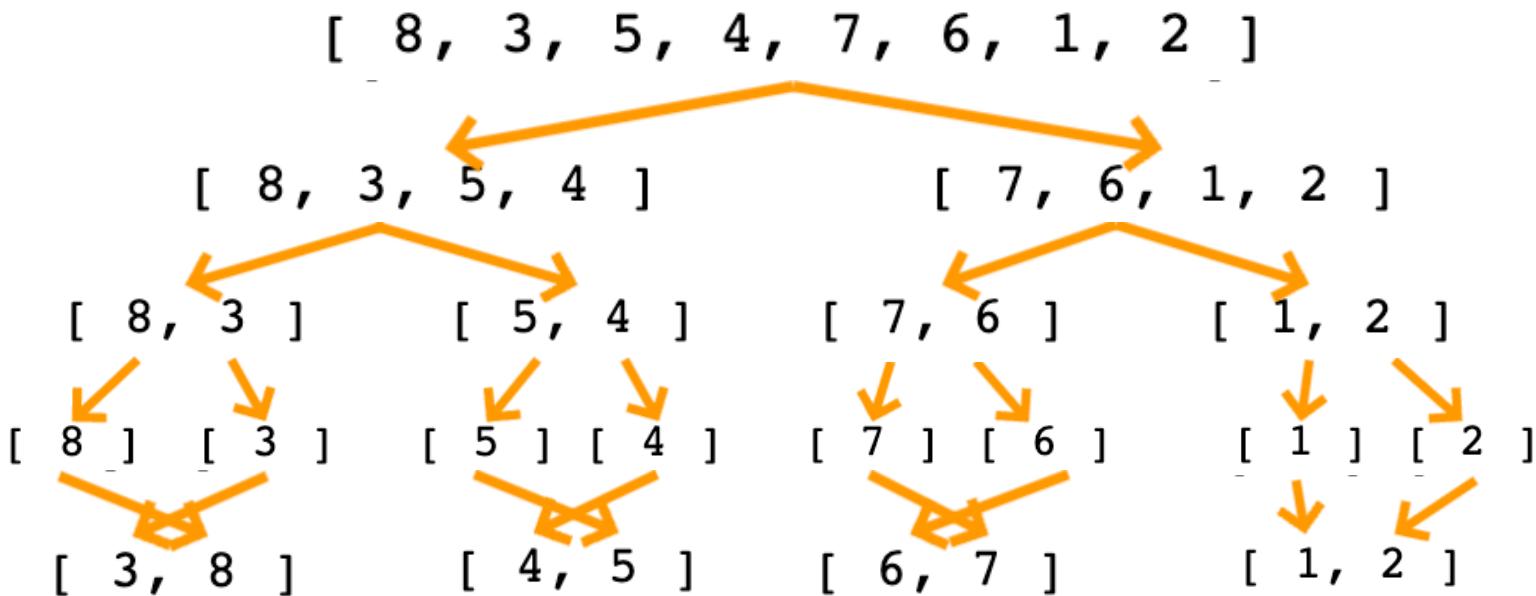
# How does it work?



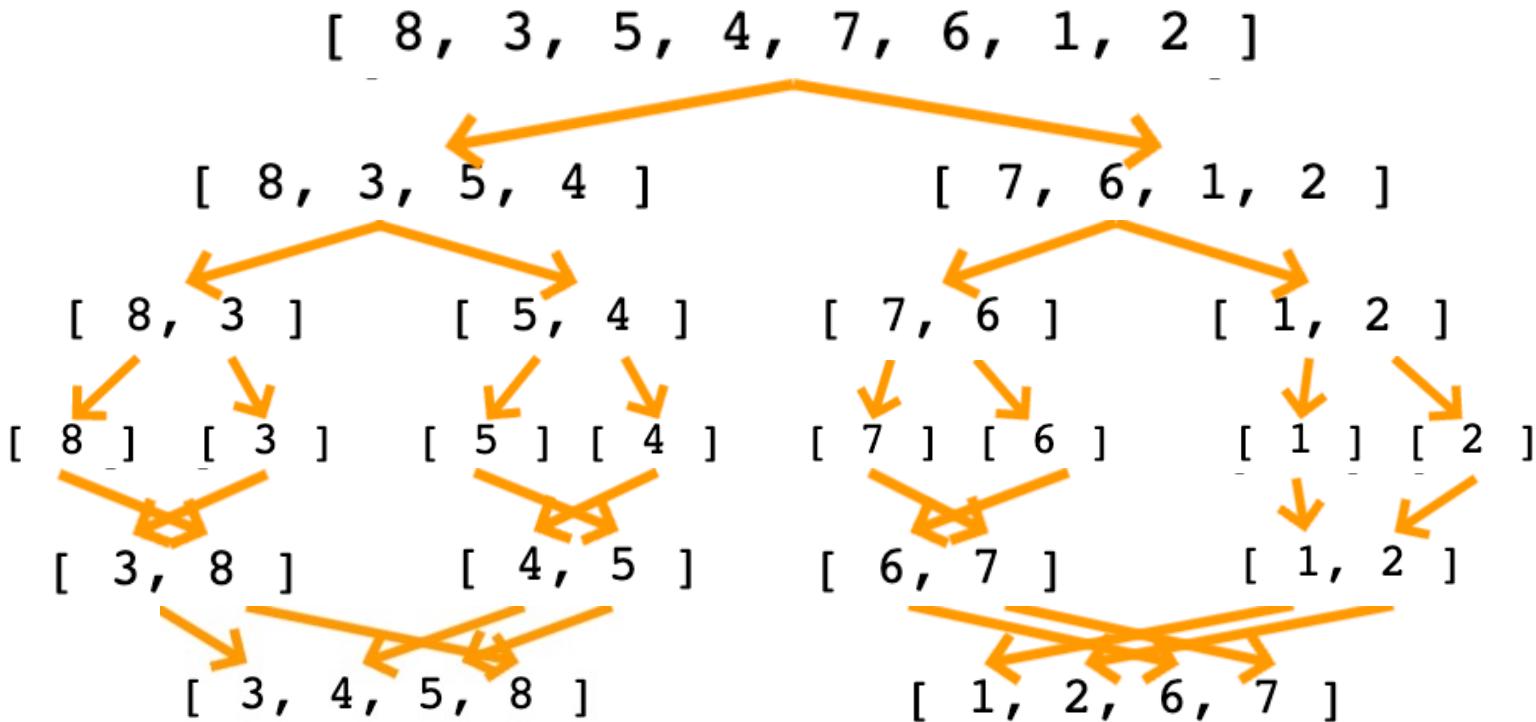
# How does it work?



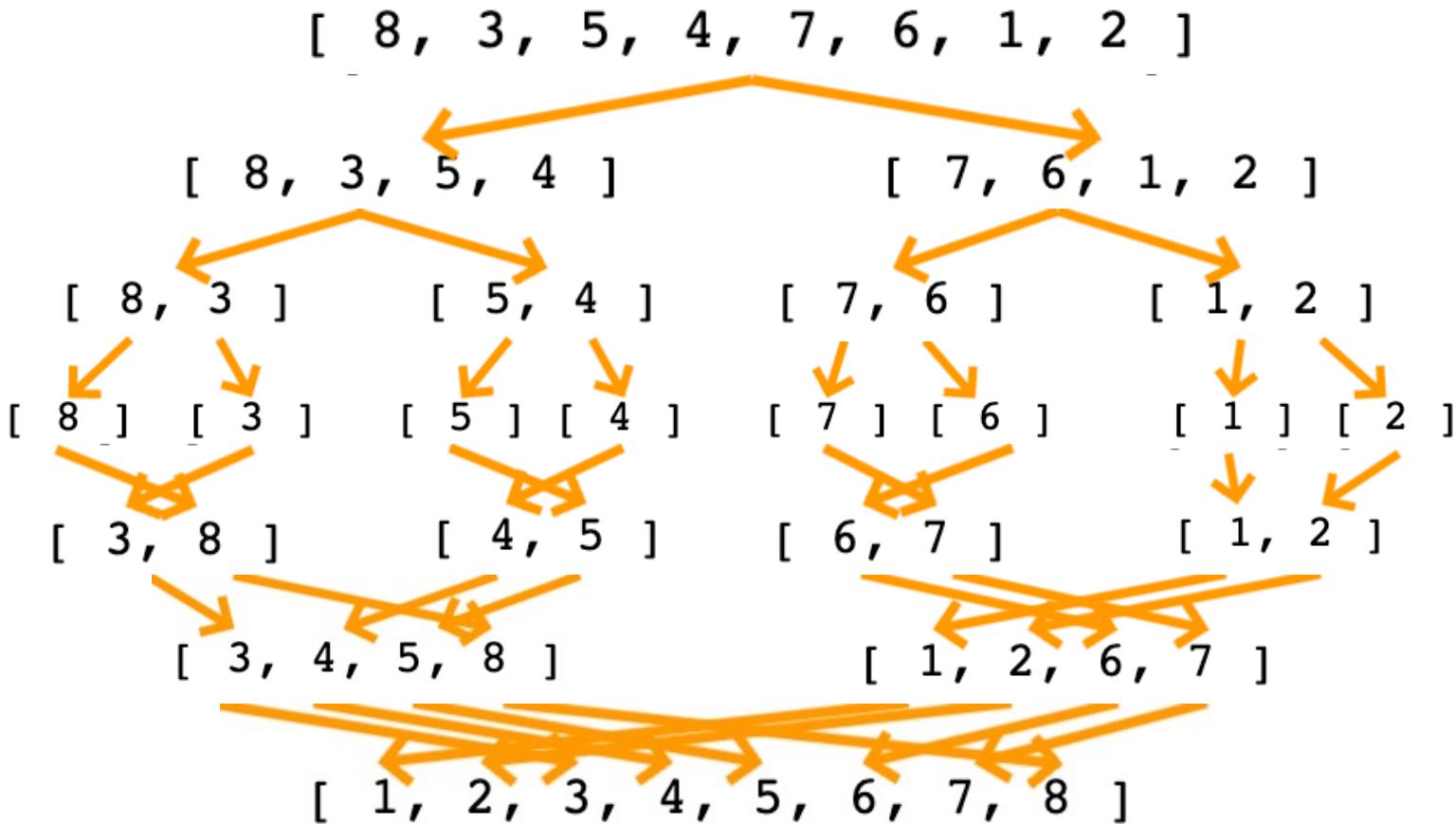
# How does it work?



# How does it work?



# How does it work?



# How does it work?

[ 8, 3, 5, 4, 7, 6, 1, 2 ]

[ 8, 3, 5, 4 ] [ 7, 6, 1, 2 ]

[ 8, 3 ] [ 5, 4 ] [ 7, 6 ] [ 1, 2 ]

[ 8 ] [ 3 ] [ 5 ] [ 4 ] [ 7 ] [ 6 ] [ 1 ] [ 2 ]

[ 3, 8 ] [ 4, 5 ] [ 6, 7 ] [ 1, 2 ]

[ 3, 4, 5, 8 ] [ 1, 2, 6, 7 ]

[ 1, 2, 3, 4, 5, 6, 7, 8 ]

# Merging Arrays

- In order to implement merge sort, it's useful to first implement a function responsible for merging two sorted arrays
- Given two arrays which are sorted, this helper function should create a new array which is also sorted, and consists of all of the elements in the two input arrays
- This function should run in  $O(n + m)$  time and  $O(n + m)$  space and **should not** modify the parameters passed to it.

# Merging Arrays Pseudocode

- Create an empty array, take a look at the smallest values in each input array
- While there are still values we haven't looked at...
  - If the value in the first array is smaller than the value in the second array, push the value in the first array into our results and move on to the next value in the first array
  - If the value in the first array is larger than the value in the second array, push the value in the second array into our results and move on to the next value in the second array
  - Once we exhaust one array, push in all remaining values from the other array

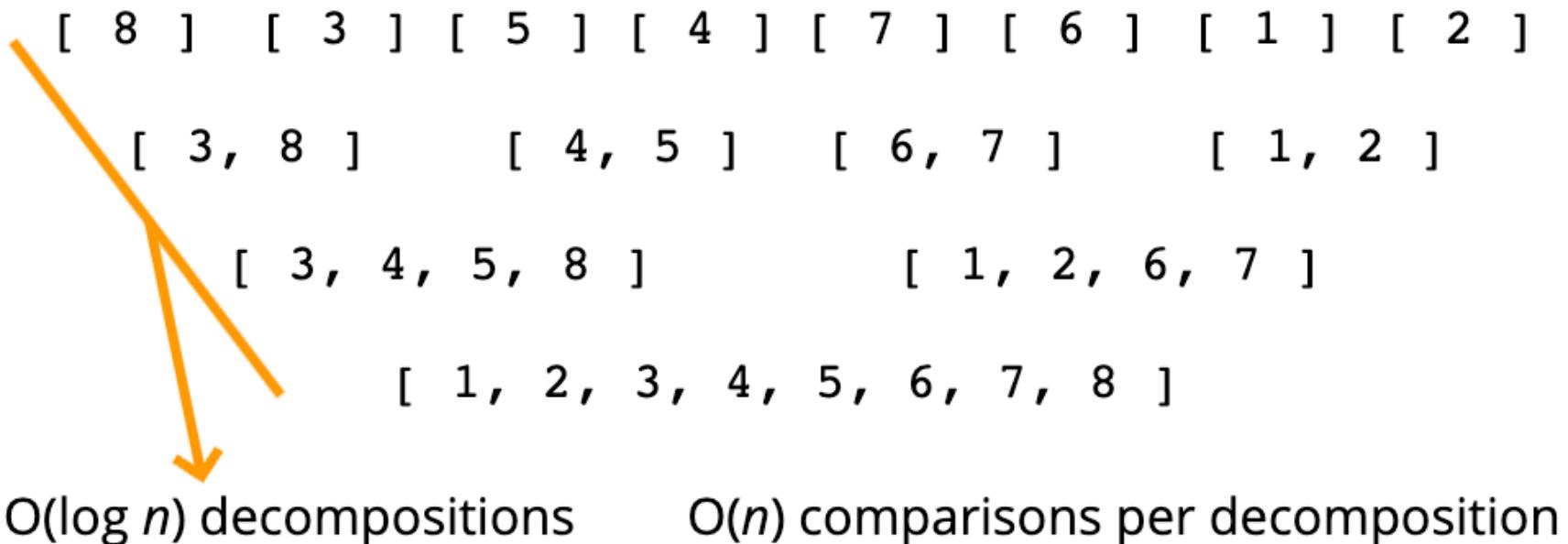
# mergeSort Pseudocode

- Break up the array into halves until you have arrays that are empty or have one element
- Once you have smaller sorted arrays, merge those arrays with other sorted arrays until you are back at the full length of the array
- Once the array has been merged back together, return the merged (and sorted!) array

# Big O of mergeSort

Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

# Big O of mergeSort



# Quick Sort

- Like merge sort, exploits the fact that arrays of 0 or 1 element are always sorted
- Works by selecting one element (called the "pivot") and finding the index where the pivot should end up in the sorted array
- Once the pivot is positioned appropriately, quick sort can be applied on either side of the pivot

# How does it work?

```
[ 5, 2, 1, 8, 4, 7, 6, 3 ]
```

# How does it work?

[ **5**, 2, 1, 8, 4, 7, 6, 3 ]

# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]

5

3, 2, 1, 4      7, 6, 8

# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]

5

3, 2, 1, 4      7, 6, 8

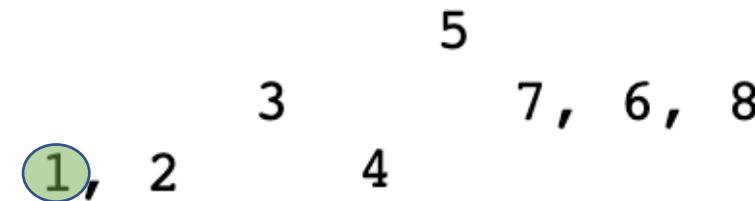
# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]

5  
3              7, 6, 8  
1, 2            4

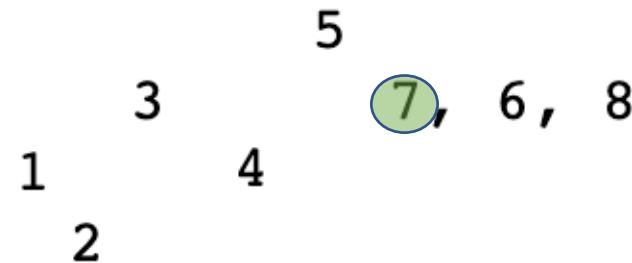
# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]



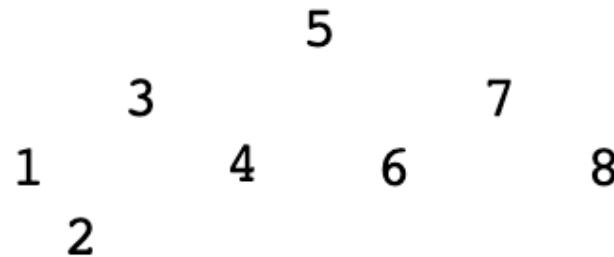
# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]



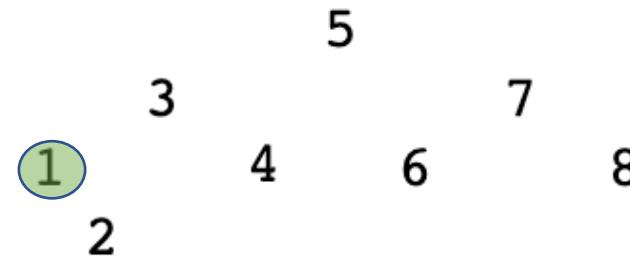
# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]



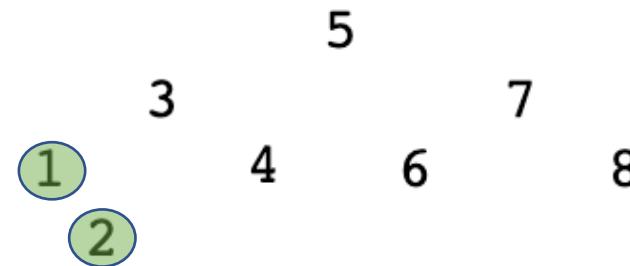
# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]



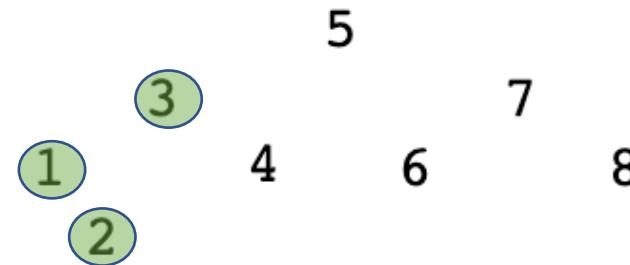
# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]



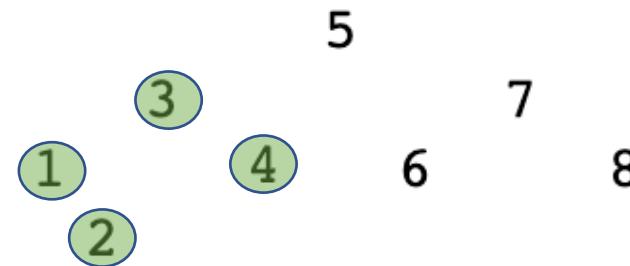
# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]



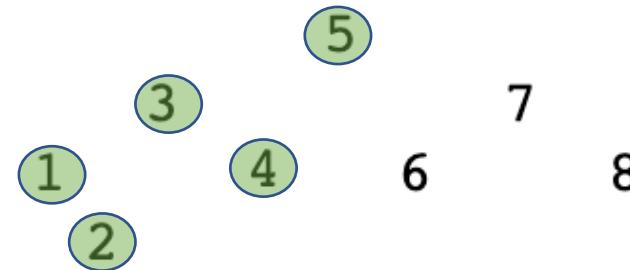
# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]



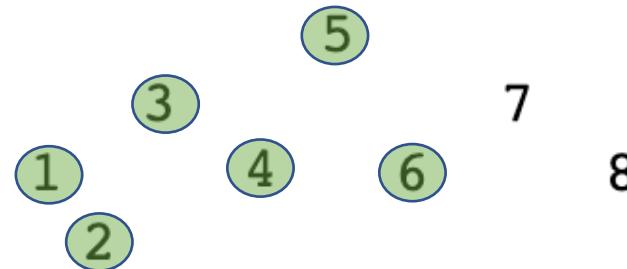
# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]



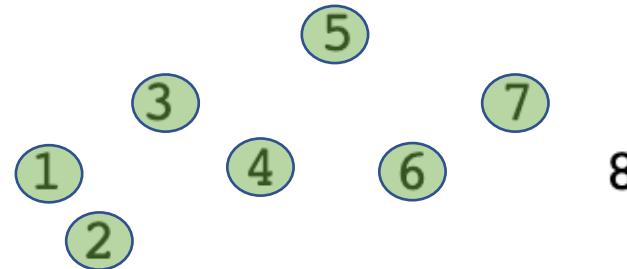
# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]



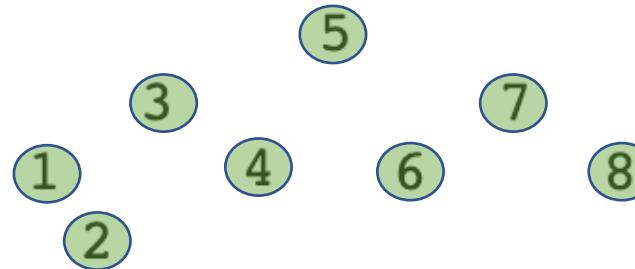
# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]



# How does it work?

[ 5, 2, 1, 8, 4, 7, 6, 3 ]



# Pivot Helper

- In order to implement merge sort, it's useful to first implement a function responsible arranging elements in an array on either side of a pivot
- Given an array, this helper function should designate an element as the pivot
- It should then rearrange elements in the array so that all values less than the pivot are moved to the left of the pivot, and all values greater than the pivot are moved to the right of the pivot
- The order of elements on either side of the pivot doesn't matter!
- The helper should do this **in place**, that is, it should not create a new array
- When complete, the helper should return the index of the pivot

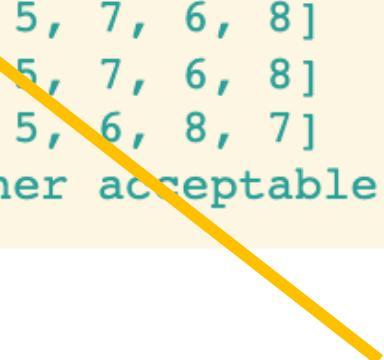
# Picking a pivot

- The runtime of quick sort depends in part on how one selects the pivot
- Ideally, the pivot should be chosen so that it's roughly the median value in the data set you're sorting
- For simplicity, we'll always choose the pivot to be the first element (we'll talk about consequences of this later)

# Pivot Helper Example

```
let arr = [ 5, 2, 1, 8, 4, 7, 6, 3 ]  
  
pivot(arr); // 4;  
  
arr;  
// any one of these is an acceptable mutation:  
// [2, 1, 4, 3, 5, 8, 7, 6]  
// [1, 4, 3, 2, 5, 7, 6, 8]  
// [3, 2, 1, 4, 5, 7, 6, 8]  
// [4, 1, 2, 3, 5, 6, 8, 7]  
// there are other acceptable mutations too!
```

# Pivot Helper Example

```
let arr = [ 5, 2, 1, 8, 4, 7, 6, 3 ]  
  
pivot(arr); // 4;  
  
arr;  
// any one of these is an acceptable mutation:  
// [2, 1, 4, 3, 5, 8, 7, 6]  
// [1, 4, 3, 2, 5, 7, 6, 8]   
// [3, 2, 1, 4, 5, 7, 6, 8]  
// [4, 1, 2, 3, 5, 6, 8, 7]  
// there are other acceptable mutations too!
```

All that matters is for 5 to be at index 4, for smaller values to be to the left, and for larger values to be to the right

# Pivot Pseudocode

- It will help to accept three arguments: an array, a start index, and an end index (these can default to 0 and the array length minus 1, respectively)
- Grab the pivot from the start of the array
- Store the current pivot index in a variable (this will keep track of where the pivot should end up)
- Loop through the array from the start until the end
  - If the pivot is greater than the current element, increment the pivot index variable and then swap the current element with the element at the pivot index
- Swap the starting element (i.e. the pivot) with the pivot index
- Return the pivot index

# Quicksort Pseudocode

- Call the pivot helper on the array
- When the helper returns to you the updated pivot index, recursively call the pivot helper on the subarray to the left of that index, and the subarray to the right of that index
- Your base case occurs when you consider a subarray with less than 2 elements

# Big O of Quicksort

<b>Time Complexity (Best)</b>	<b>Time Complexity (Average)</b>	<b>Time Complexity (Worst)</b>	<b>Space Complexity</b>
$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

# Big O of Quicksort

Why???

# Big O of Quicksort

Why???

Best Case

```
[8, 5, 6, 1, 3, 7, 2, 4, 12, 13, 14, 11, 9, 15, 10]
```

# Big O of Quicksort

Why???

Best Case

8

[4, 5, 6, 1, 3, 7, 2]

[12, 13, 14, 11, 9, 15, 10]

# Big O of Quicksort

Why???

Best Case

8

4

12

[2, 1, 3]

[6, 7, 5]

[10, 11, 9]

[14, 15, 13]

# Big O of Quicksort

Why???

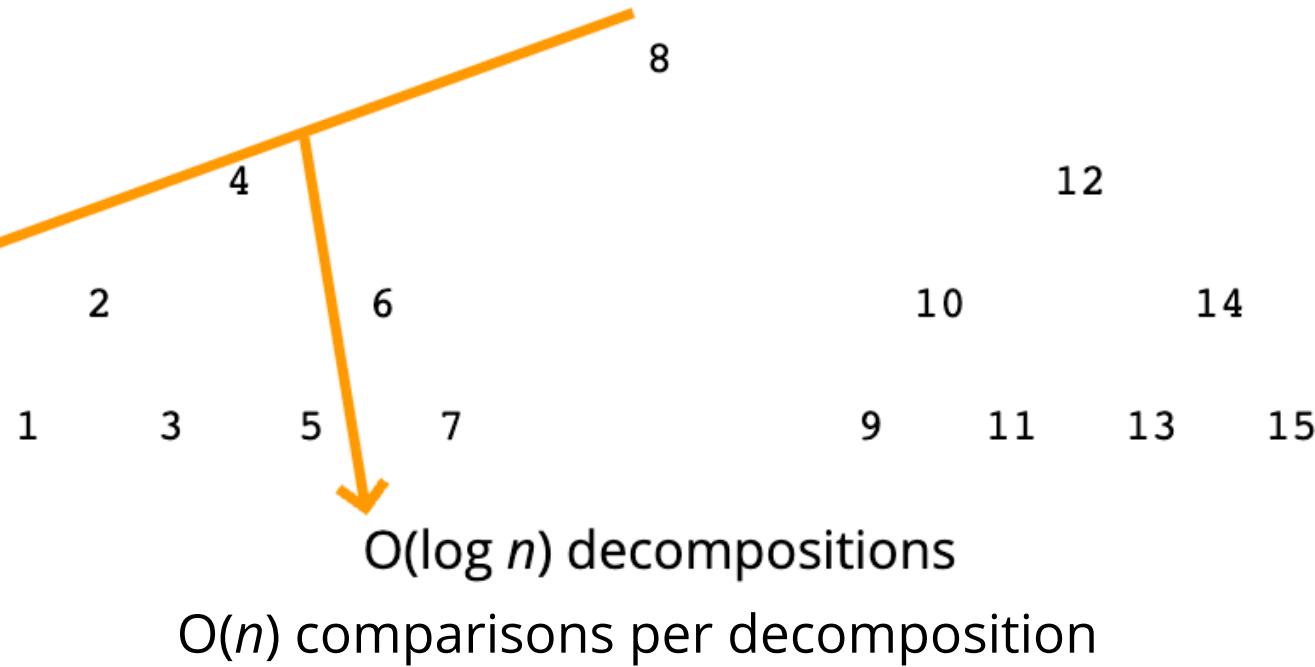
Best Case



# Big O of Quicksort

Why???

Best Case



# Big O of Quicksort

Why???

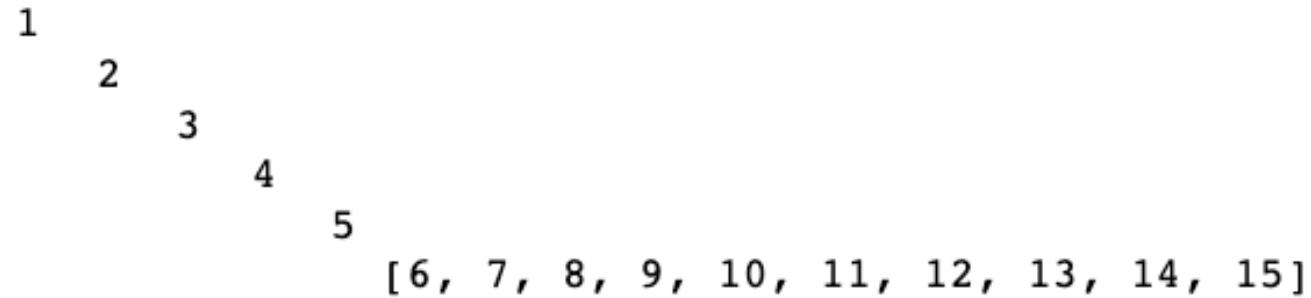
Worst Case

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

# Big O of Quicksort

Why???

Worst Case



# Big O of Quicksort

Why???

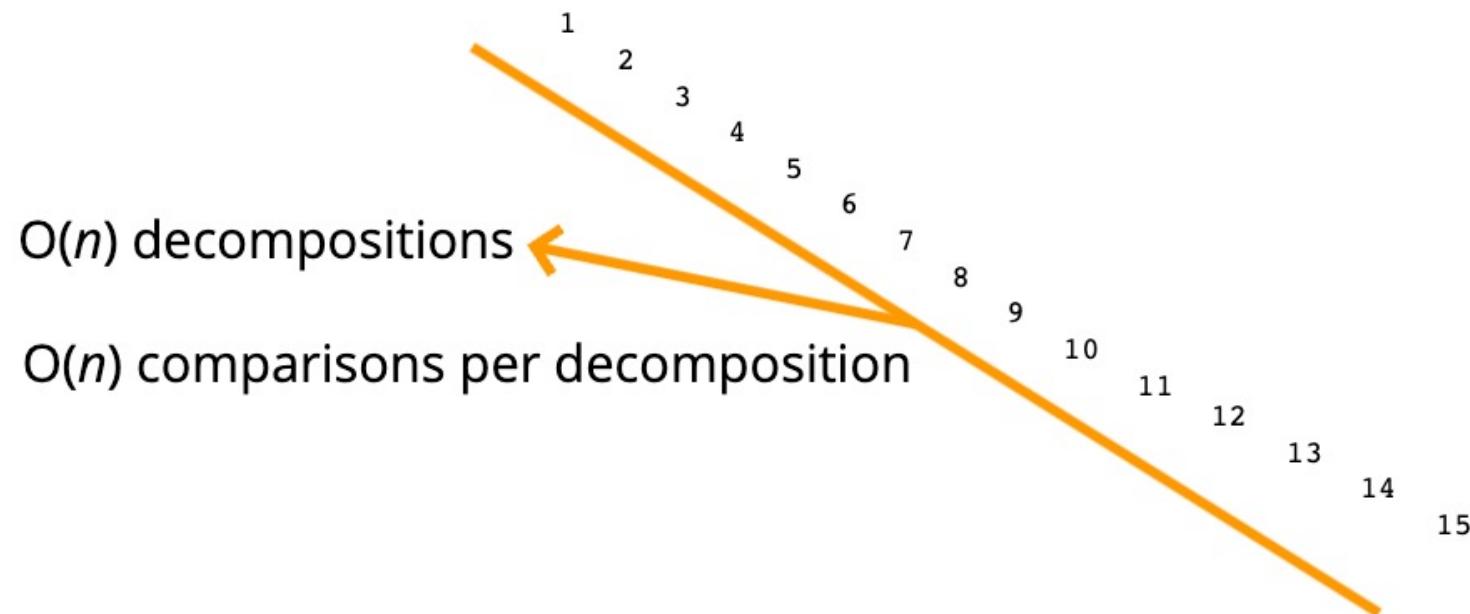
Worst Case



# Big O of Quicksort

Why???

Worst Case



# COMPARISON SORTS

## Average Time Complexity

- Bubble Sort -  $O(n^2)$
- Insertion Sort -  $O(n^2)$
- Selection Sort -  $O(n^2)$
- Quick Sort -  $O(n \log (n))$
- Merge Sort -  $O(n \log (n))$

# COMPARISON SORTS

Average Time Complexity

- Bubble Sort -  $O(n^2)$
- Insertion Sort -  $O(n^2)$
- Selection Sort -  $O(n^2)$
- Quick Sort -  $O(n \log (n))$
- Merge Sort -  $O(n \log (n))$

Can we do better?

CAN WE DO  
BETTER?

YES,  
BUT NOT BY MAKING  
COMPARISONS

# RADIX SORT

# RADIX SORT

Radix sort is a special sorting algorithm that works on lists of numbers

- It never makes comparisons between elements!

- It exploits the fact that information about the size of a number is encoded in the number of digits.

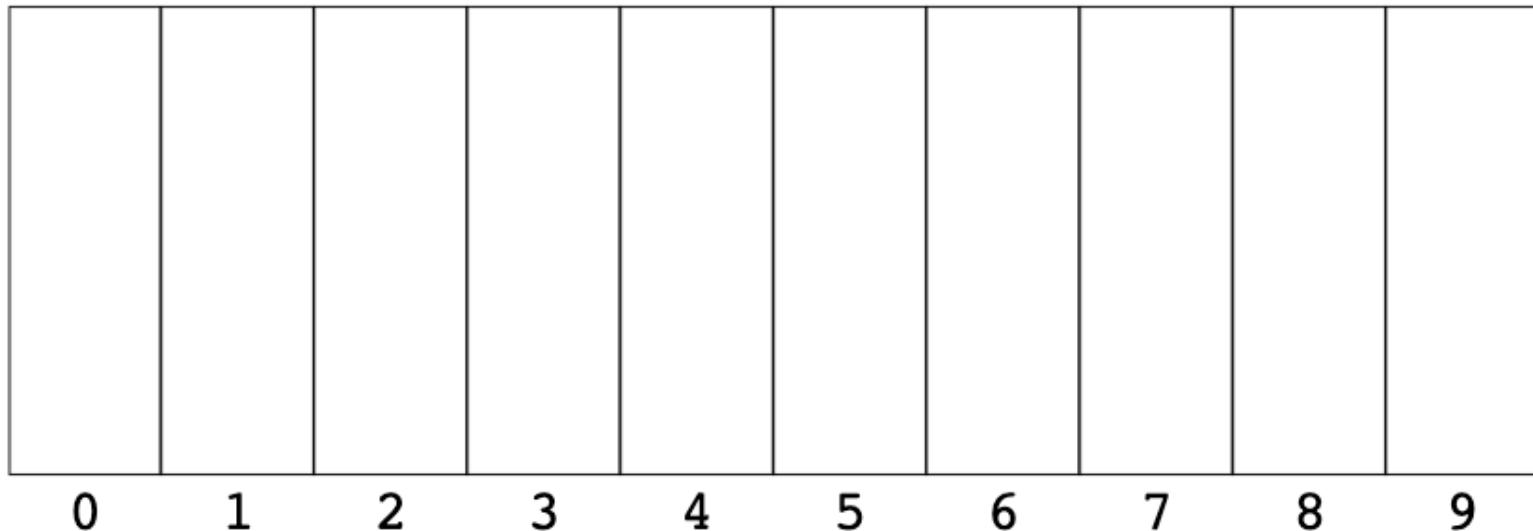
- More digits means a bigger number!

# How does it work?

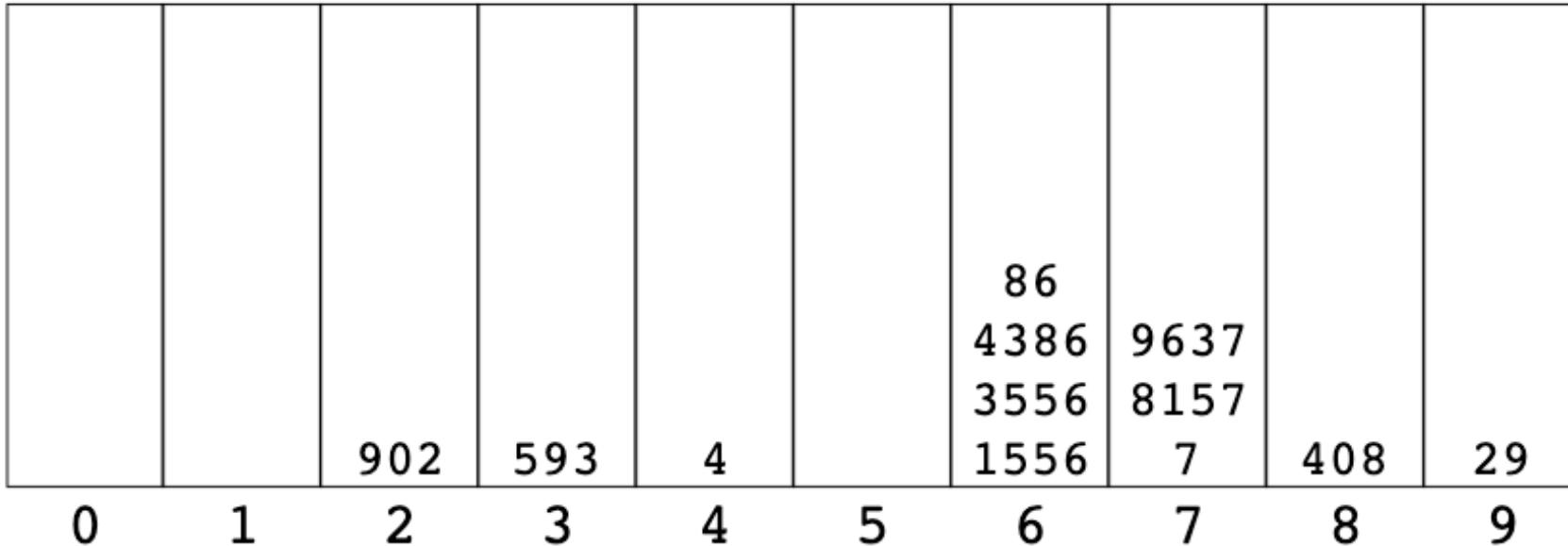
```
[1556, 4, 3556, 593, 408, 4386, 902, 7, 8157, 86, 9637, 29]
```

# How does it work?

[1556, 4, 3556, 593, 408, 4386, 902, 7, 8157, 86, 9637, 29]

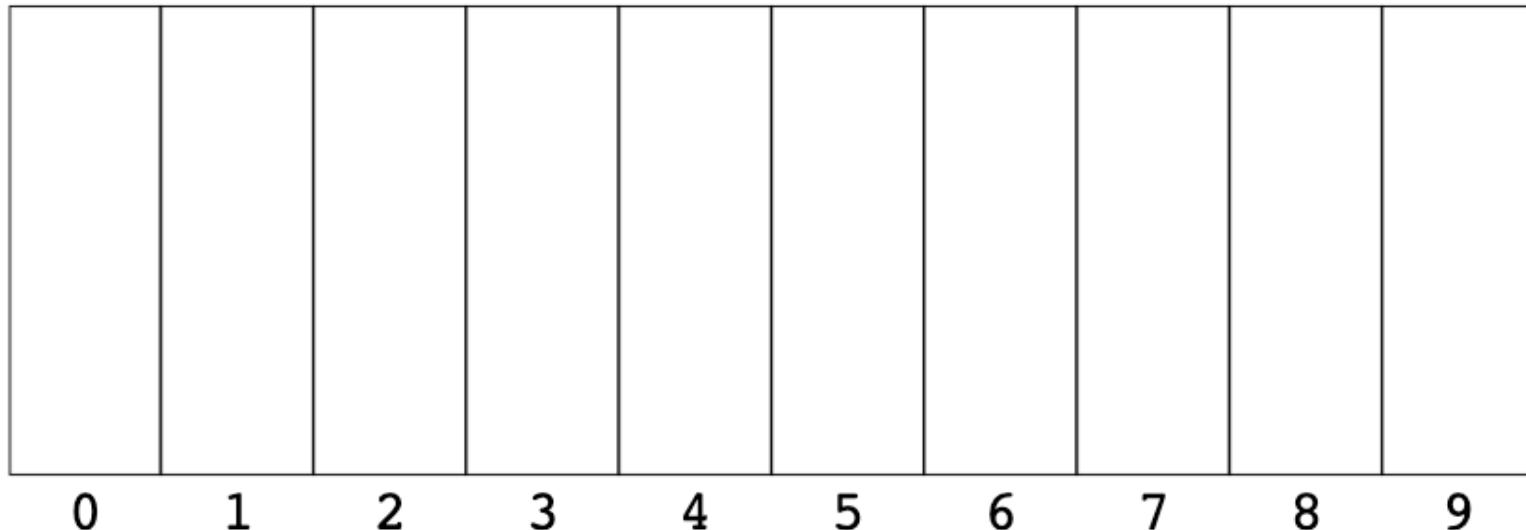


# How does it work?

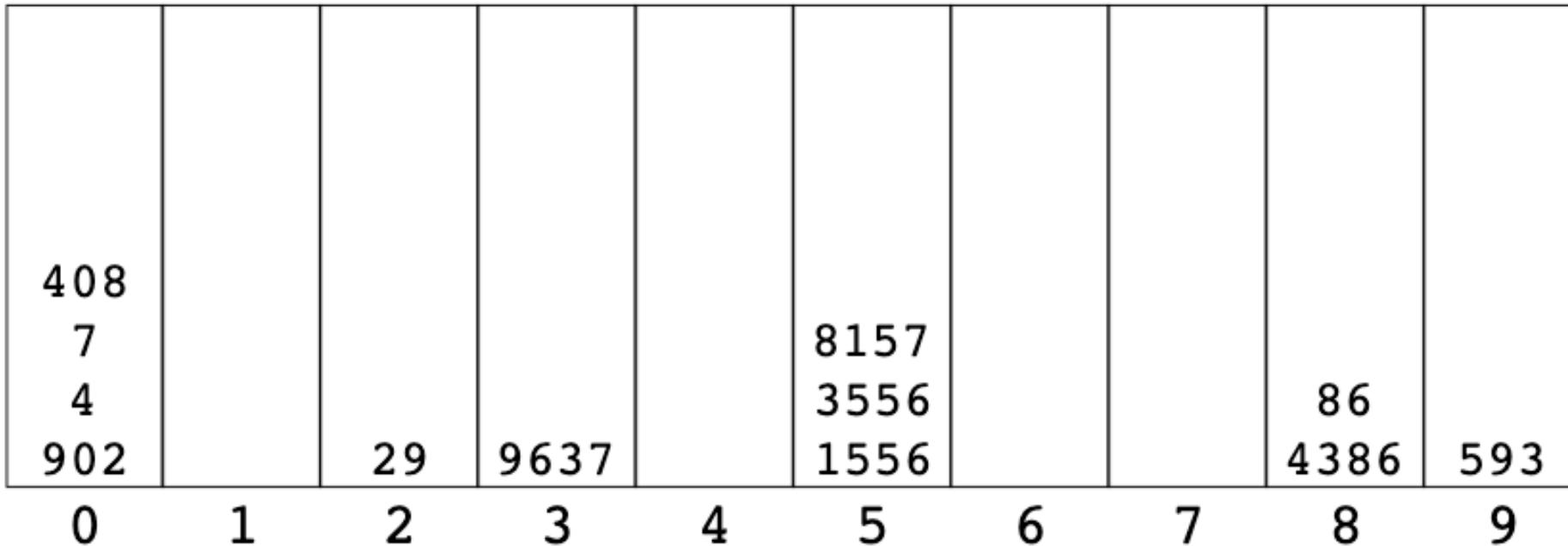


# How does it work?

[902, 593, 4, 1556, 3556, 4386, 86, 7, 8157, 9637, 408, 29]

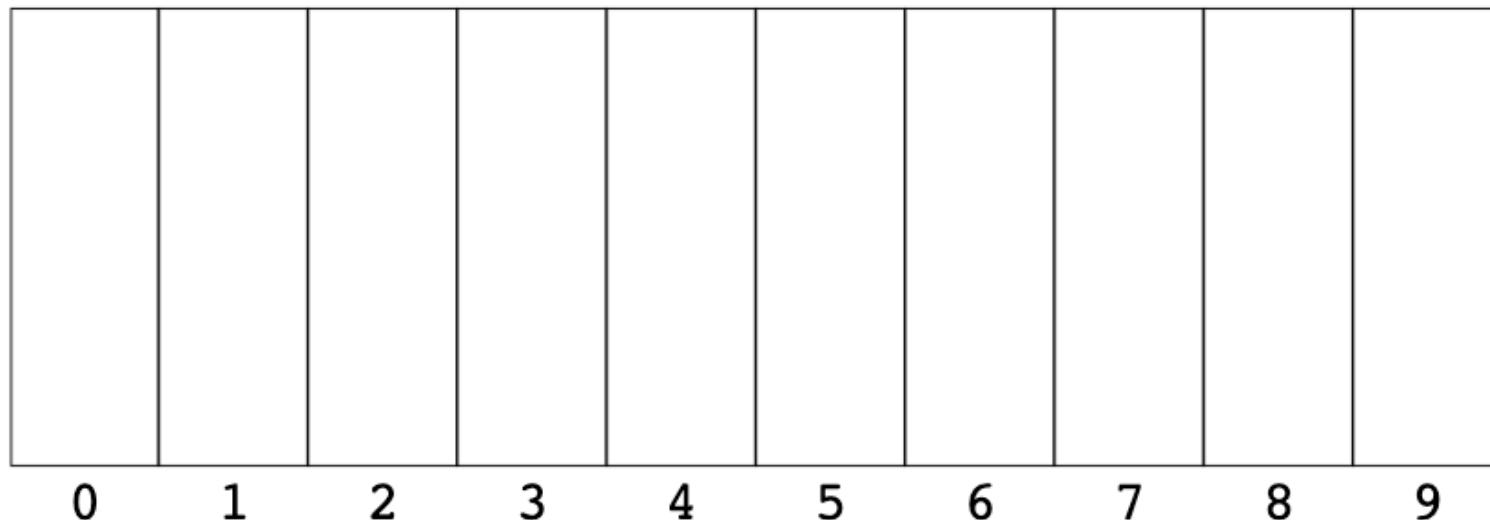


# How does it work?

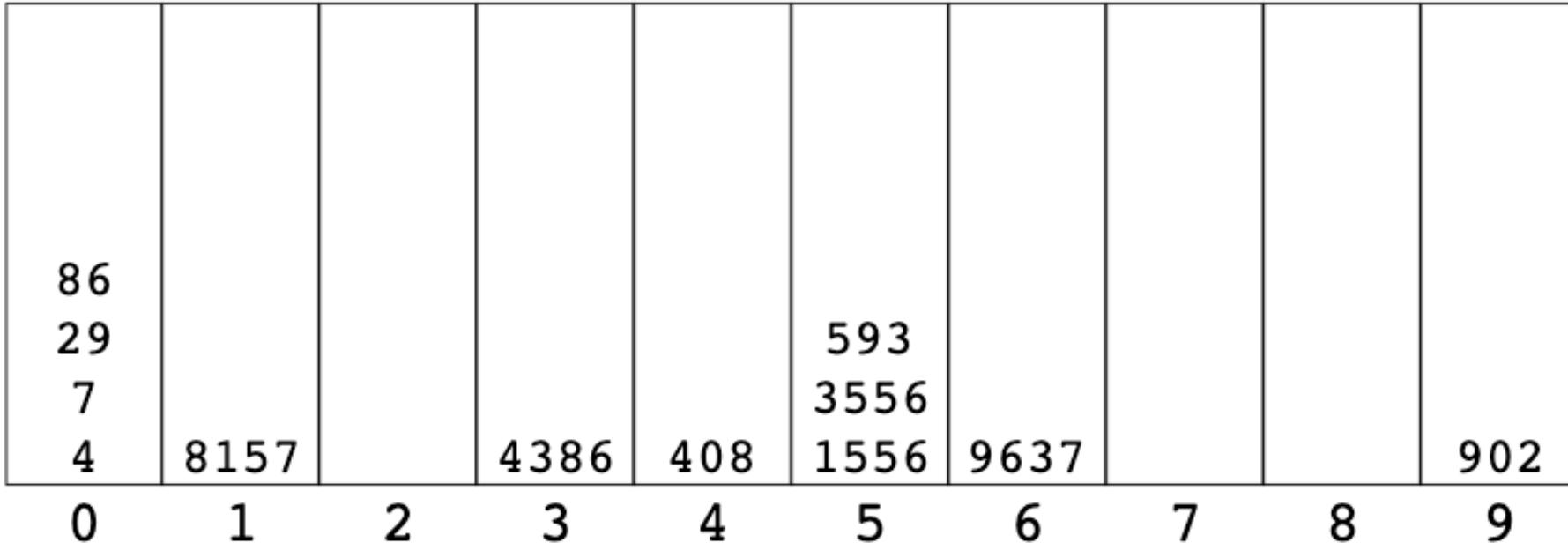


# How does it work?

[ 902, 4, 7, 408, 29, 9637, 1556, 3556, 8157, 4386, 86, 593 ]

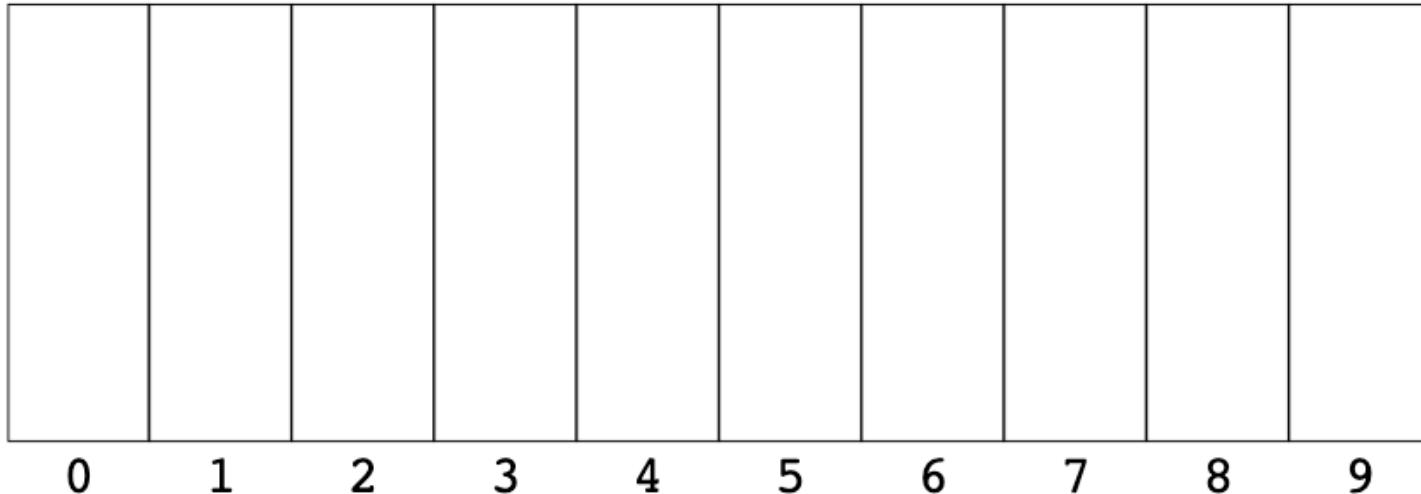


# How does it work?

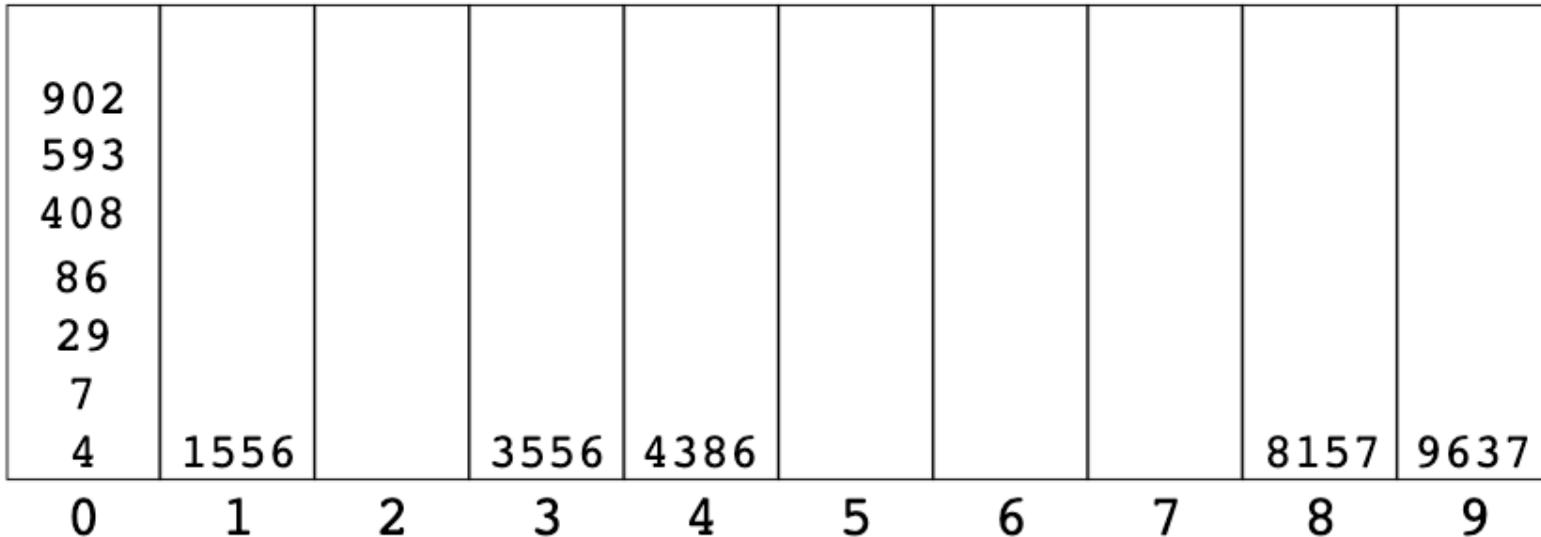


# How does it work?

[4, 7, 29, 86, 8157, 4386, 408, 1556, 3556, 593, 9637, 902]

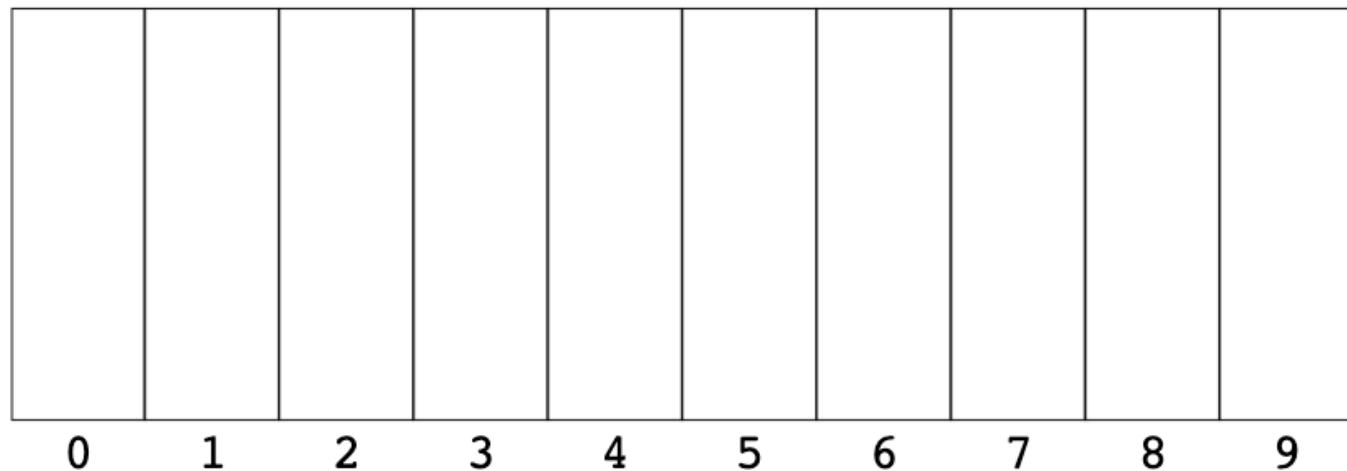


# How does it work?



# How does it work?

[ 4, 7, 29, 86, 408, 593, 902, 1556, 3556, 4386, 8157, 9637 ]



# RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`getDigit(num, place)` - returns the digit in *num* at the given *place* value

```
getDigit(12345, 0); // 5
getDigit(12345, 1); // 4
getDigit(12345, 2); // 3
getDigit(12345, 3); // 2
getDigit(12345, 4); // 1
getDigit(12345, 5); // 0
```

# RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`getDigit(num, place)` - returns the digit in *num* at the given *place* value

```
function getDigit(num, i) {
    return Math.floor(Math.abs(num) / Math.pow(10, i)) % 10;
}
```

# RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`digitCount(num)` - returns the number of digits in *num*

```
digitCount(1); // 1
digitCount(25); // 2
digitCount(314); // 3
```

# RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`digitCount(num)` - returns the number of digits in *num*

```
function digitCount(num) {  
    if (num === 0) return 1;  
    return Math.floor(Math.log10(Math.abs(num))) + 1;  
}
```

# RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`mostDigits(nums)` - Given an array of numbers, returns the number of digits in the largest numbers in the list

```
mostDigits([1234, 56, 7]); // 4
mostDigits([1, 1, 11111, 1]); // 5
mostDigits([12, 34, 56, 78]); // 2
```

# RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`mostDigits(nums)` - Given an array of numbers, returns the number of digits in the largest numbers in the list

```
function mostDigits(nums) {
    let maxDigits = 0;
    for (let i = 0; i < nums.length; i++) {
        maxDigits = Math.max(maxDigits, digitCount(nums[i]));
    }
    return maxDigits;
}
```

# RADIX SORT PSEUDOCODE

- Define a function that accepts list of numbers
- Figure out how many digits the largest number has
- Loop from  $k = 0$  up to this largest number of digits
- For each iteration of the loop:
  - Create buckets for each digit (0 to 9)
  - place each number in the corresponding bucket based on its  $k$ th digit
- Replace our existing array with values in our buckets, starting with 0 and going up to 9
- return list at the end!

# RADIX SORT BIG O

Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
$O(nk)$	$O(nk)$	$O(nk)$	$O(n + k)$

- $n$  - length of array
- $k$  - number of digits(average)

# Recap

- Merge sort and quick sort are standard efficient sorting algorithms
- Quick sort can be slow in the worst case, but is comparable to merge sort on average
- Merge sort takes up more memory because it creates a new array (in-place merge sorts exist, but they are really complex!)
- Radix sort is a fast sorting algorithm for numbers
- Radix sort exploits place value to sort numbers in linear time (for a fixed number of digits)

# DATA STRUCTURES

# Binary Search Trees

# Binary Search Trees

## Queues

# Binary Search Trees

Queues

Singly Linked Lists

**Binary Search Trees**

**Queues**

**Singly Linked Lists**

**Undirected Unweighted Graphs**

**Binary Search Trees**

**Queues**

**Singly Linked Lists**

**Undirected Unweighted Graphs**

**Binary Heaps**

**Binary Search Trees**

**Queues**

**Singly Linked Lists**

**Undirected Unweighted Graphs**

**Binary Heaps**

**Directed Graphs**

# Binary Search Trees

Queues

Singly Linked Lists

Undirected Unweighted Graphs

Binary Heaps

Directed Graphs

Hash Tables

# Binary Search Trees

Queues

Singly Linked Lists

Undirected Unweighted Graphs

Binary Heaps

Directed Graphs

Hash Tables

Doubly Linked Lists

# Binary Search Trees

Queues

Singly Linked Lists

Undirected Unweighted Graphs

Binary Heaps

Directed Graphs

Hash Tables

Doubly Linked Lists

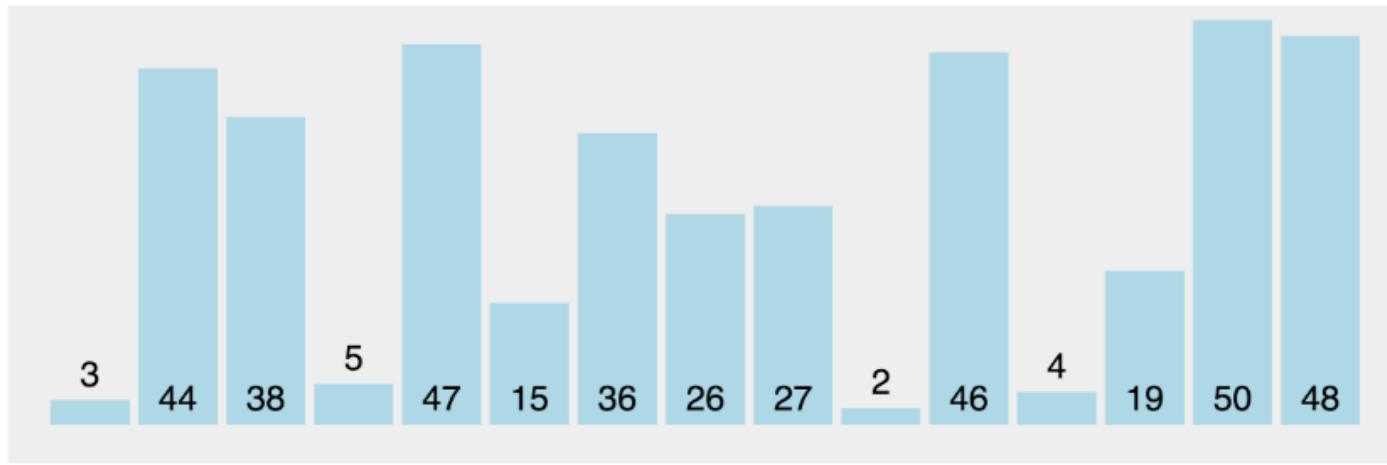
Stacks

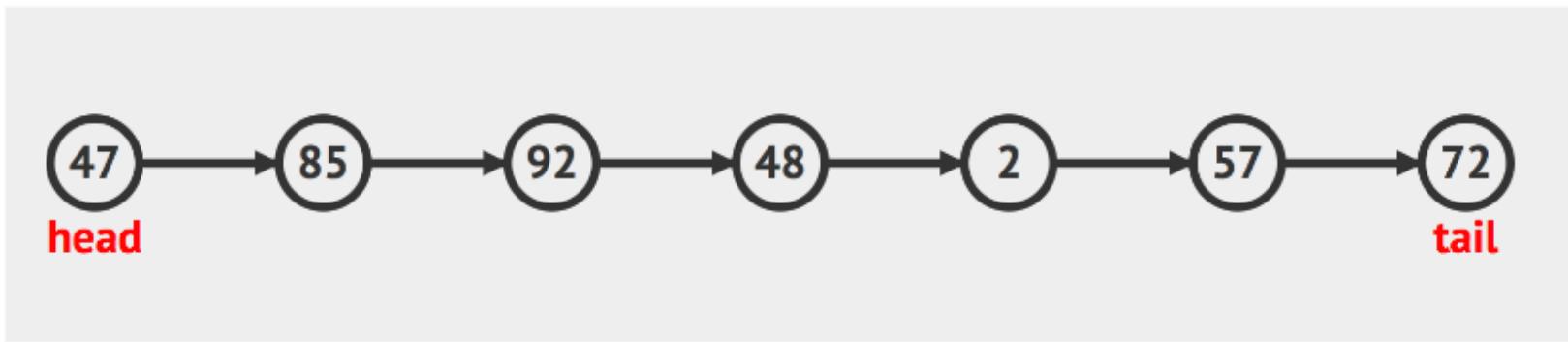
# WHAT DO THEY DO?

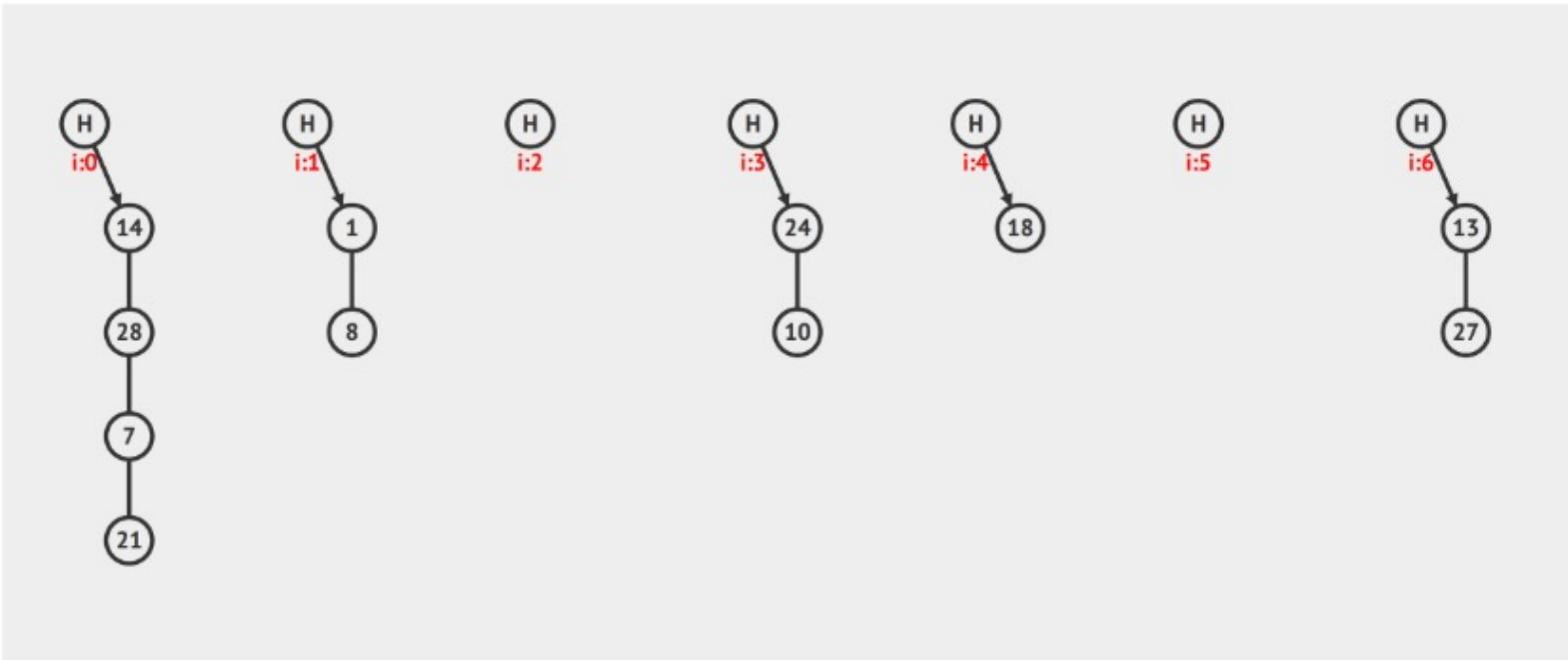
Data structures are collections of values, the relationships among them, and the functions or operations that can be applied to the data

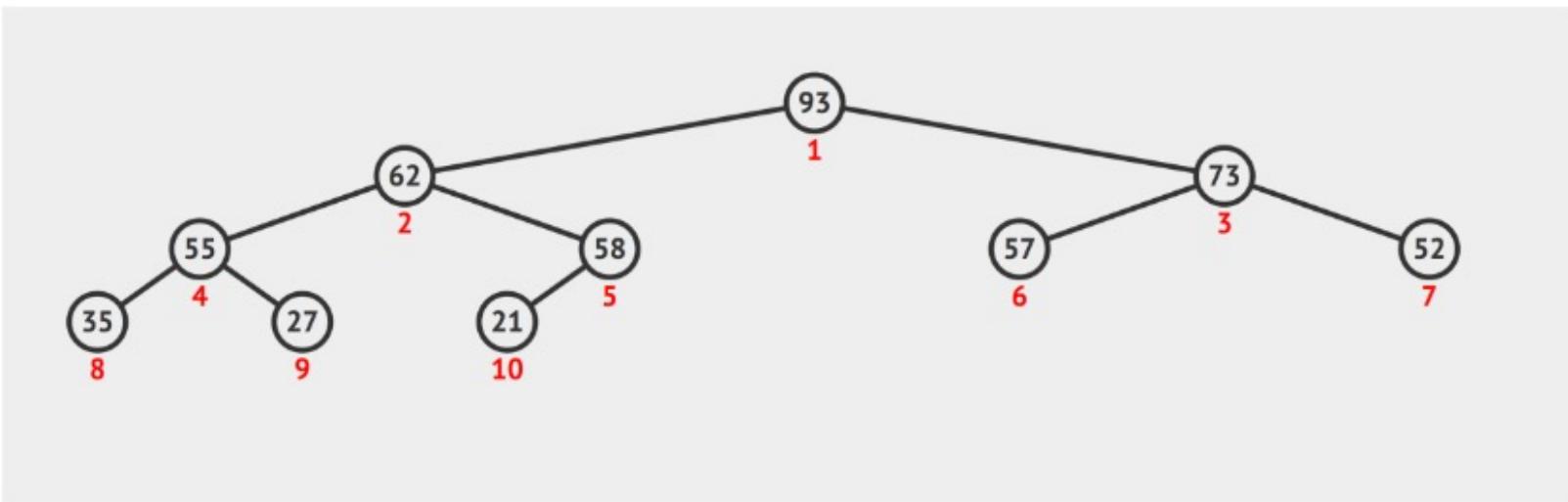
# WHY SO MANY???

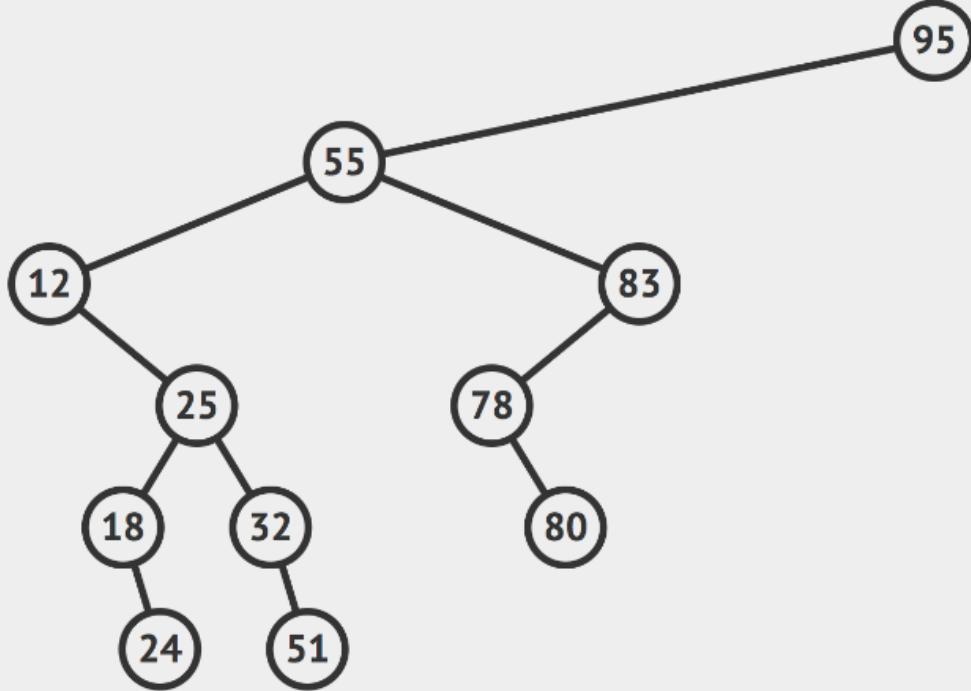
Different data structures excel at different things. Some are highly specialized, while others (like arrays) are more generally used.

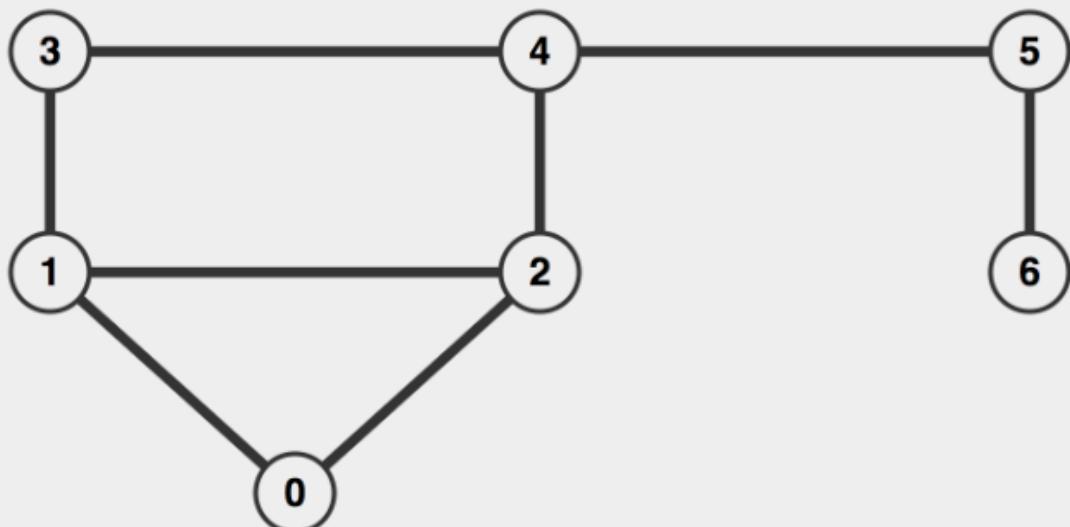


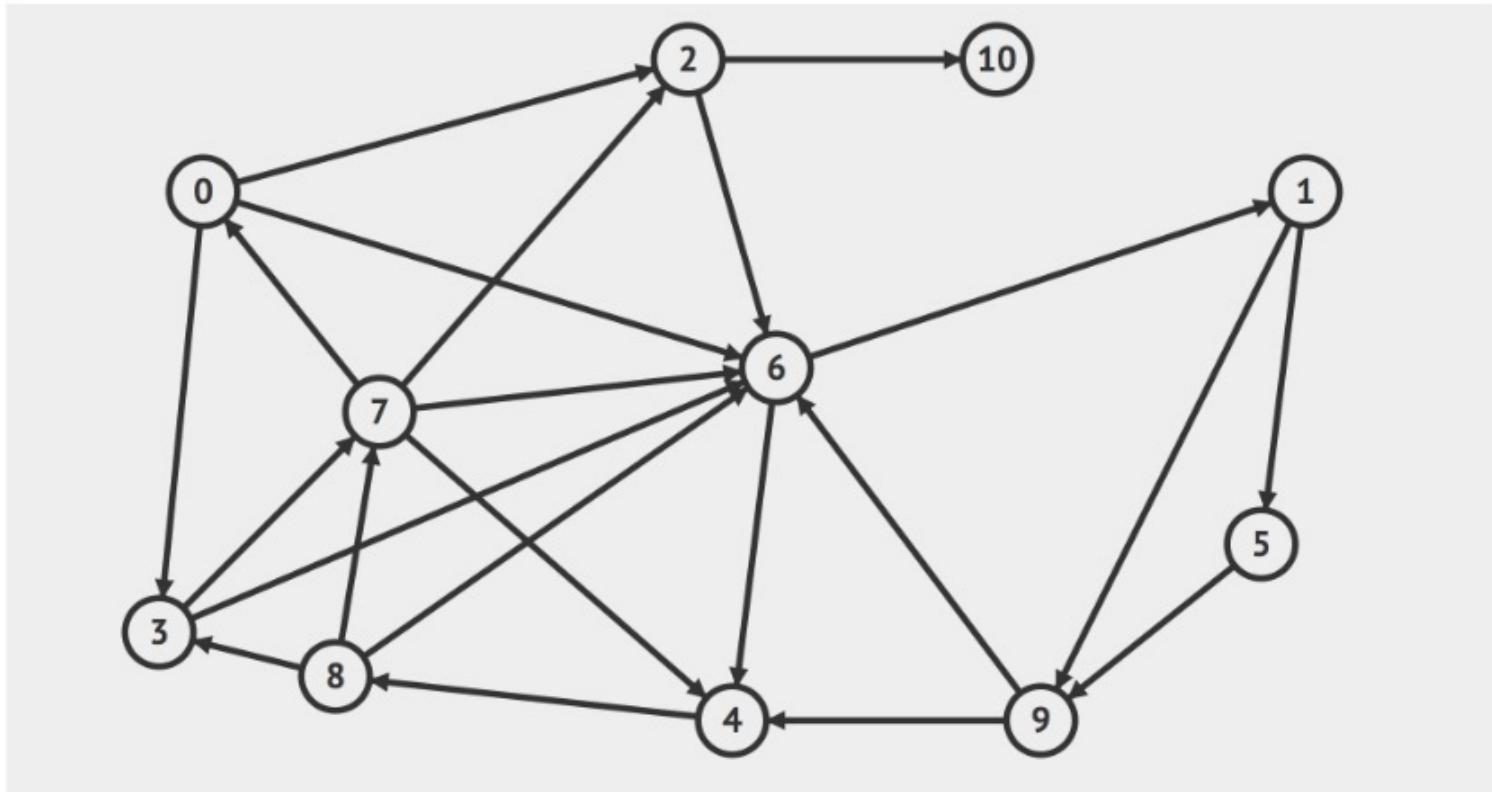












# WHY CARE?

The more time you spend as a developer, the more likely you'll need to use one of these data structures

# WHY CARE?

The more time you spend as a developer, the more likely you'll need to use one of these data structures

You've already worked with many of them unknowingly!

# WHY CARE?

The more time you spend as a developer, the more likely you'll need to use one of these data structures

You've already worked with many of them unknowingly!

And of course...**INTERVIEWS**

THERE IS NO ONE  
"BEST"  
DATA STRUCTURE

They all excel in different  
situations...

Otherwise why bother  
learning them all??

Working with  
map/location data?

Working with  
map/location data?

Use a graph!

Need an ordered list with fast  
inserts/removals at the  
beginning and end?

Need an ordered list with fast  
inserts/removals at the  
beginning and end?

Use a linked list!

# Web scraping nested HTML?

Web scraping nested HTML?

Use a tree!

Need to write a scheduler?

Need to write a scheduler?

Use a binary heap!

# BINARY HEAPS

# OBJECTIVES

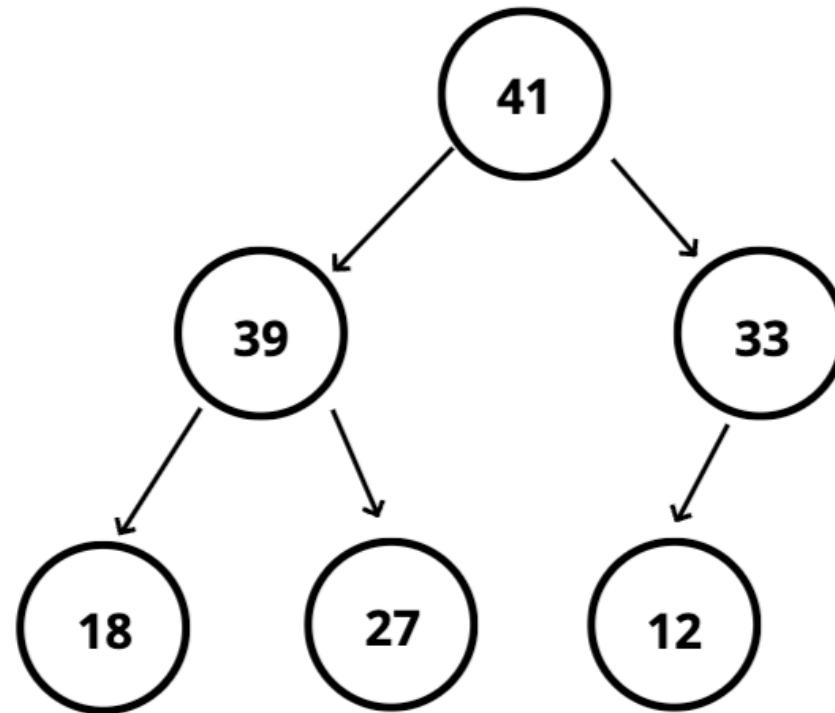
- Define what a binary heap is
- Compare and contrast min and max heaps
- Implement basic methods on heaps
- Understand where heaps are used in the real world and what other data structures can be constructed from heaps

# WHAT IS A BINARY HEAP?

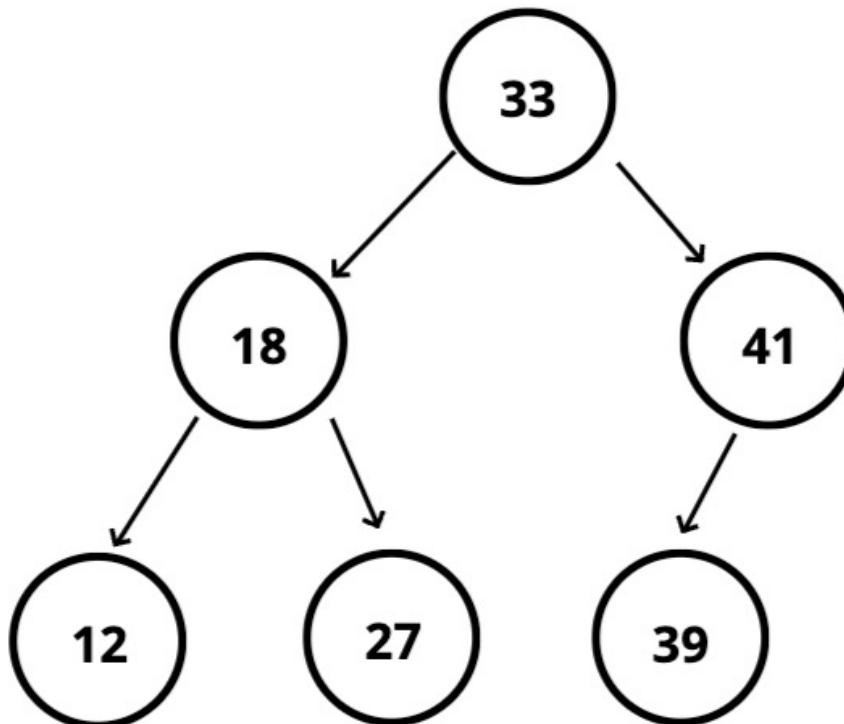
**Very** similar to a binary search tree, but with some different rules!

In a **MaxBinaryHeap**, parent nodes are always larger than child nodes. In a **MinBinaryHeap**, parent nodes are always smaller than child nodes

# WHAT DOES IT LOOK LIKE?



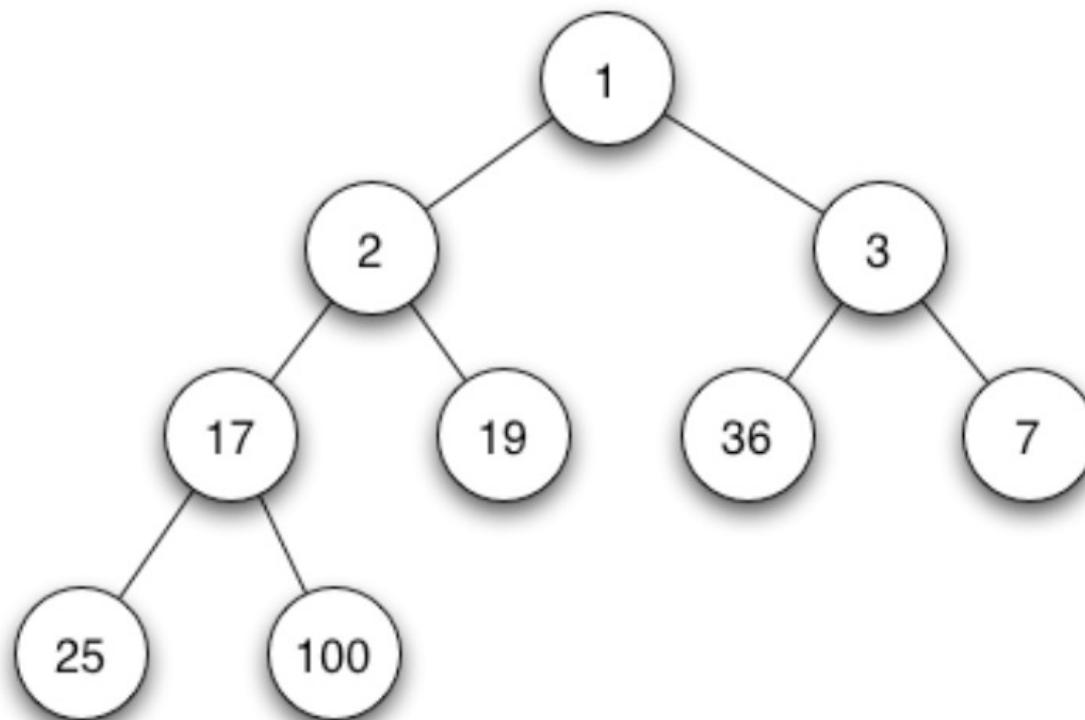
# NOT A BINARY HEAP



# MAX BINARY HEAP

- Each parent has at most two child nodes
- The value of each parent node is **always** greater than its child nodes
- In a max Binary Heap the parent is greater than the children, but there are no guarantees between sibling nodes.
- A binary heap is as compact as possible. All the children of each node are as full as they can be and left children are filled out first

# A MIN BINARY HEAP

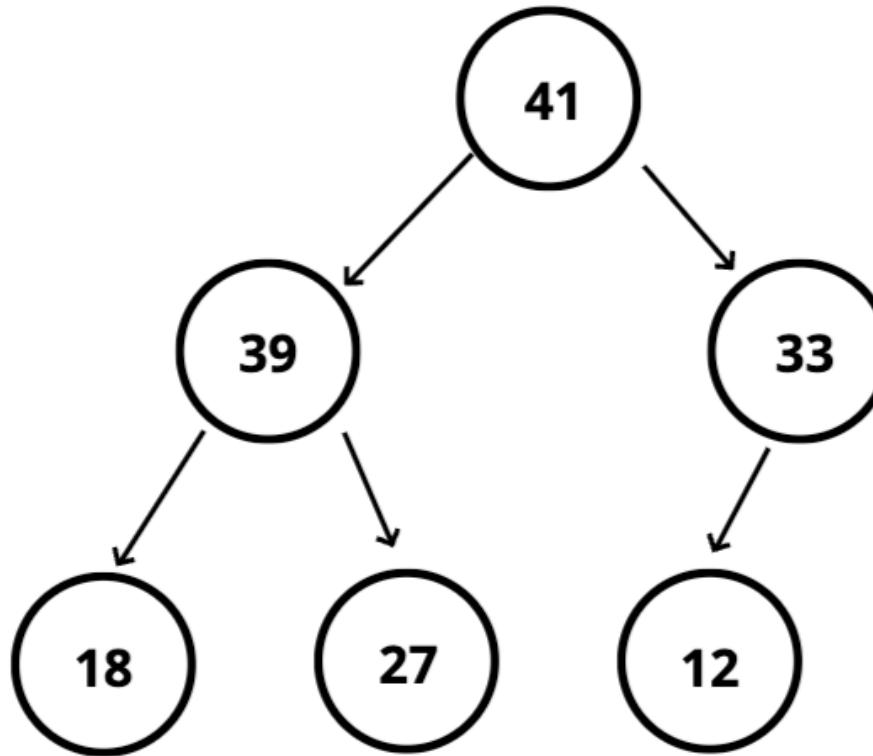


# Why do we need to know this?

Binary Heaps are used to implement Priority Queues, which are **very** commonly used data structures

They are also used quite a bit, with **graph traversal** algorithms

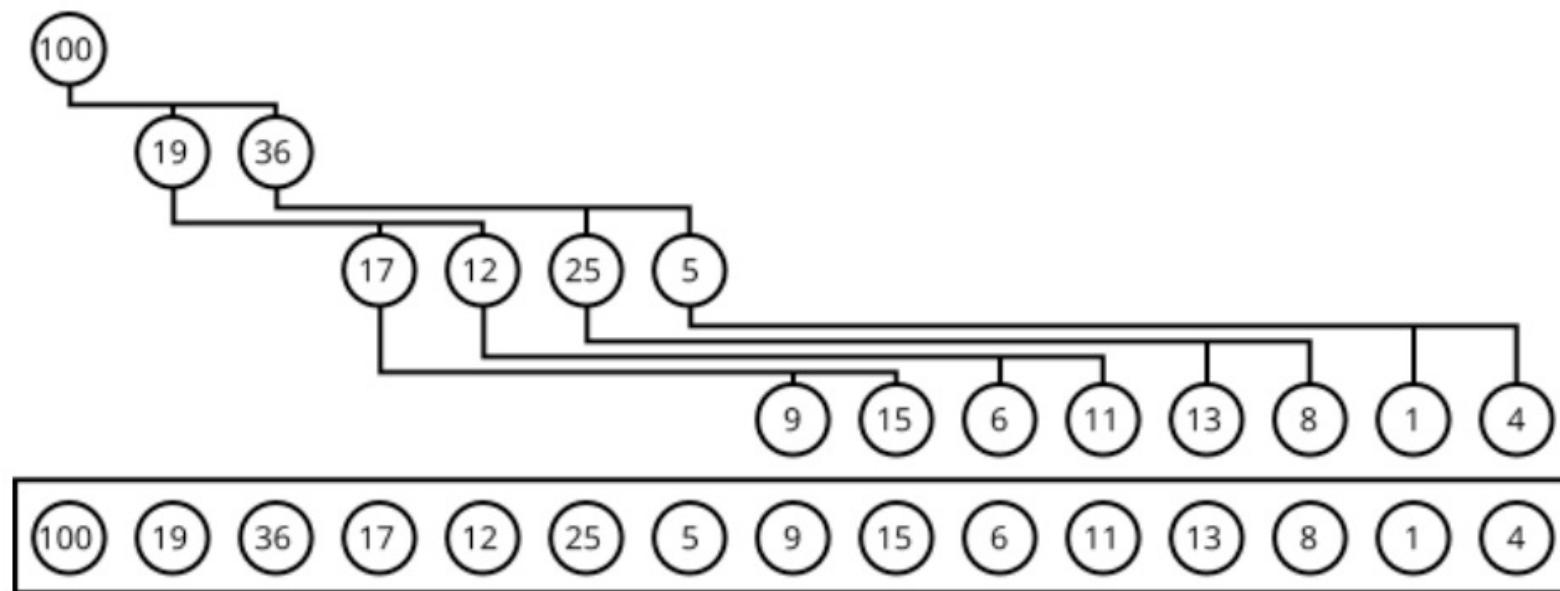
# REPRESENTING HEAPS



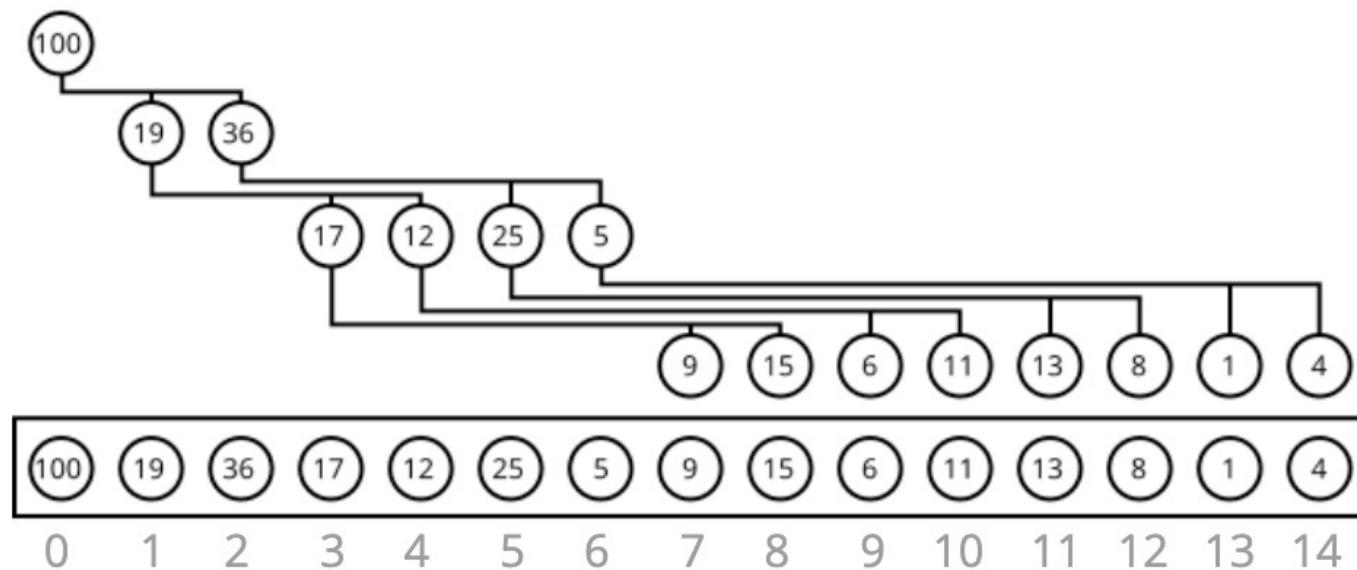
THERE'S AN EASY WAY OF  
STORING A BINARY HEAP...

A LIST/ARRAY

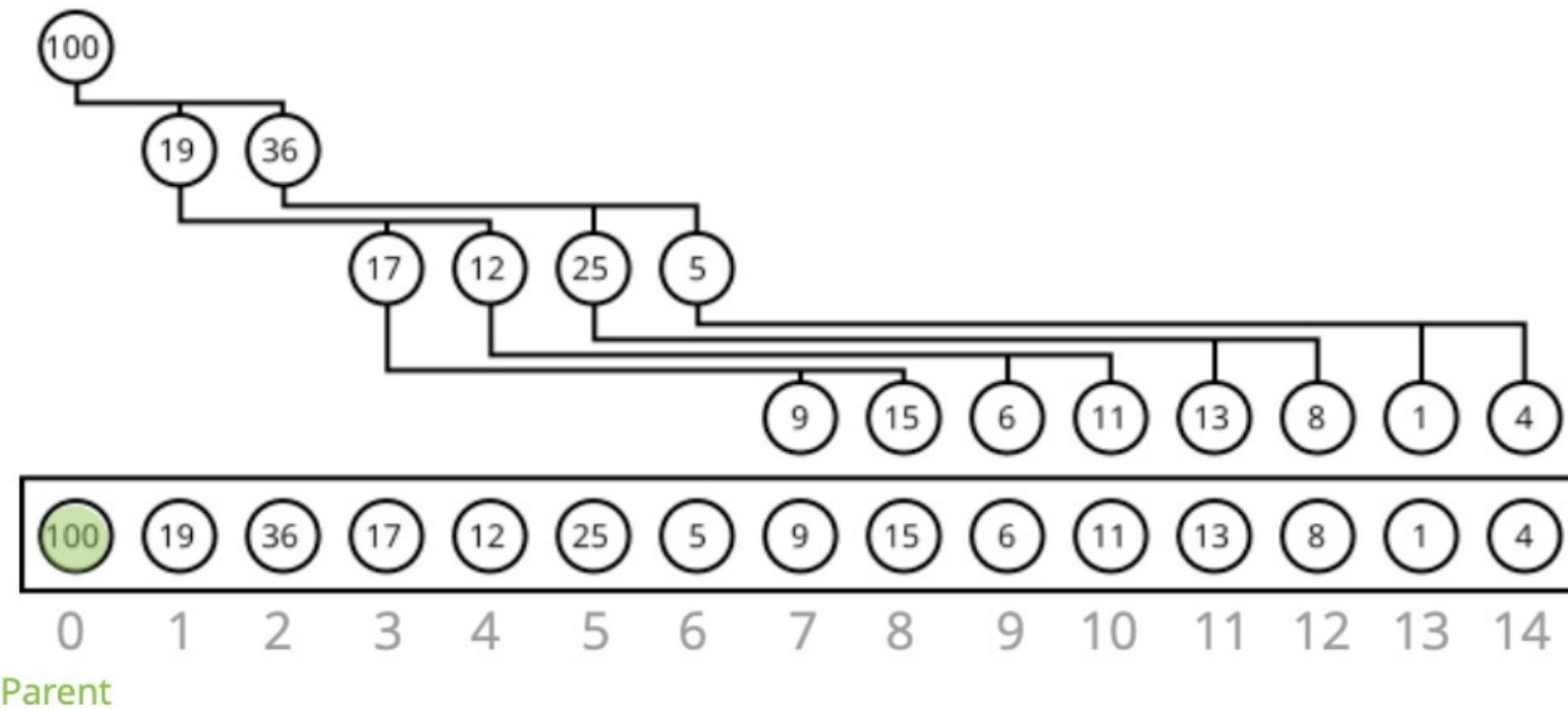
# REPRESENTING A HEAP



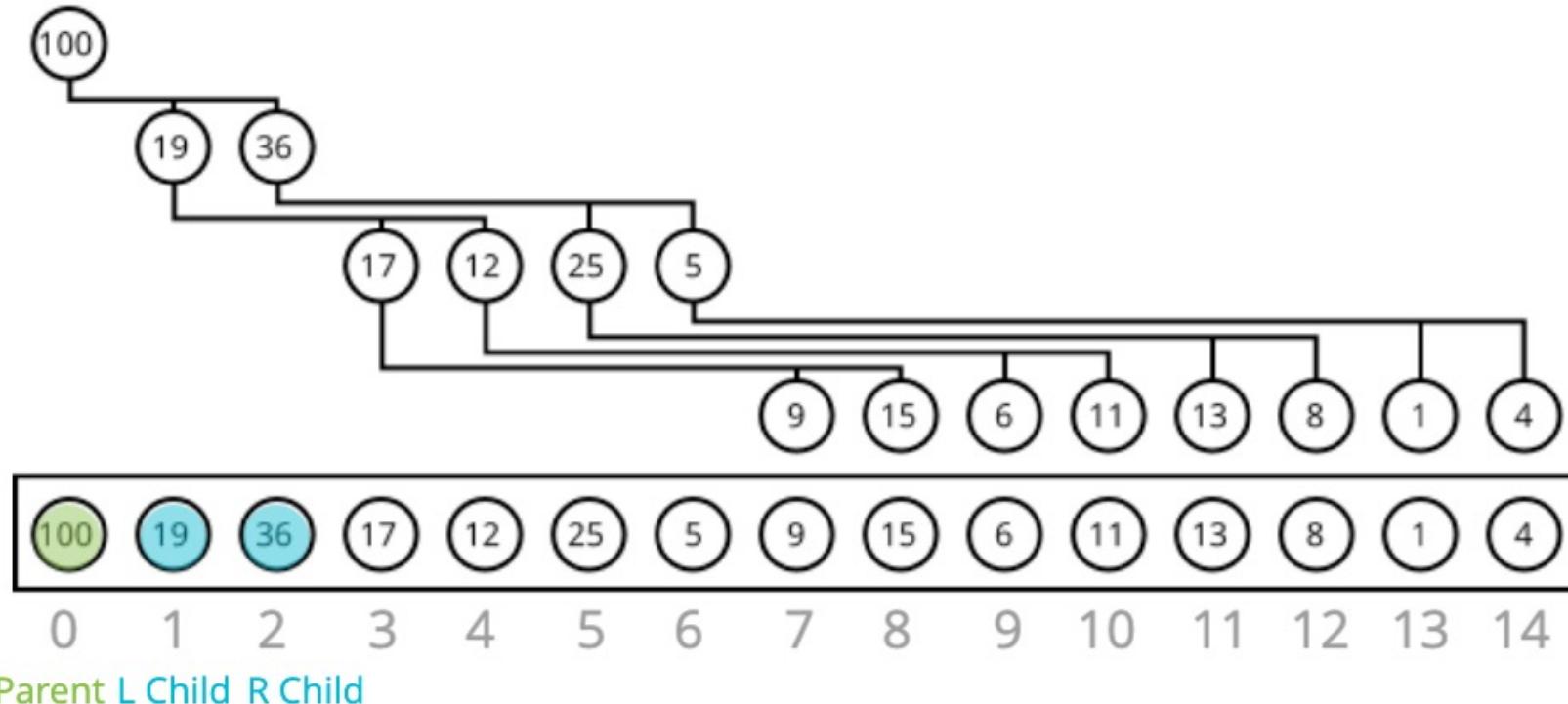
# REPRESENTING A HEAP



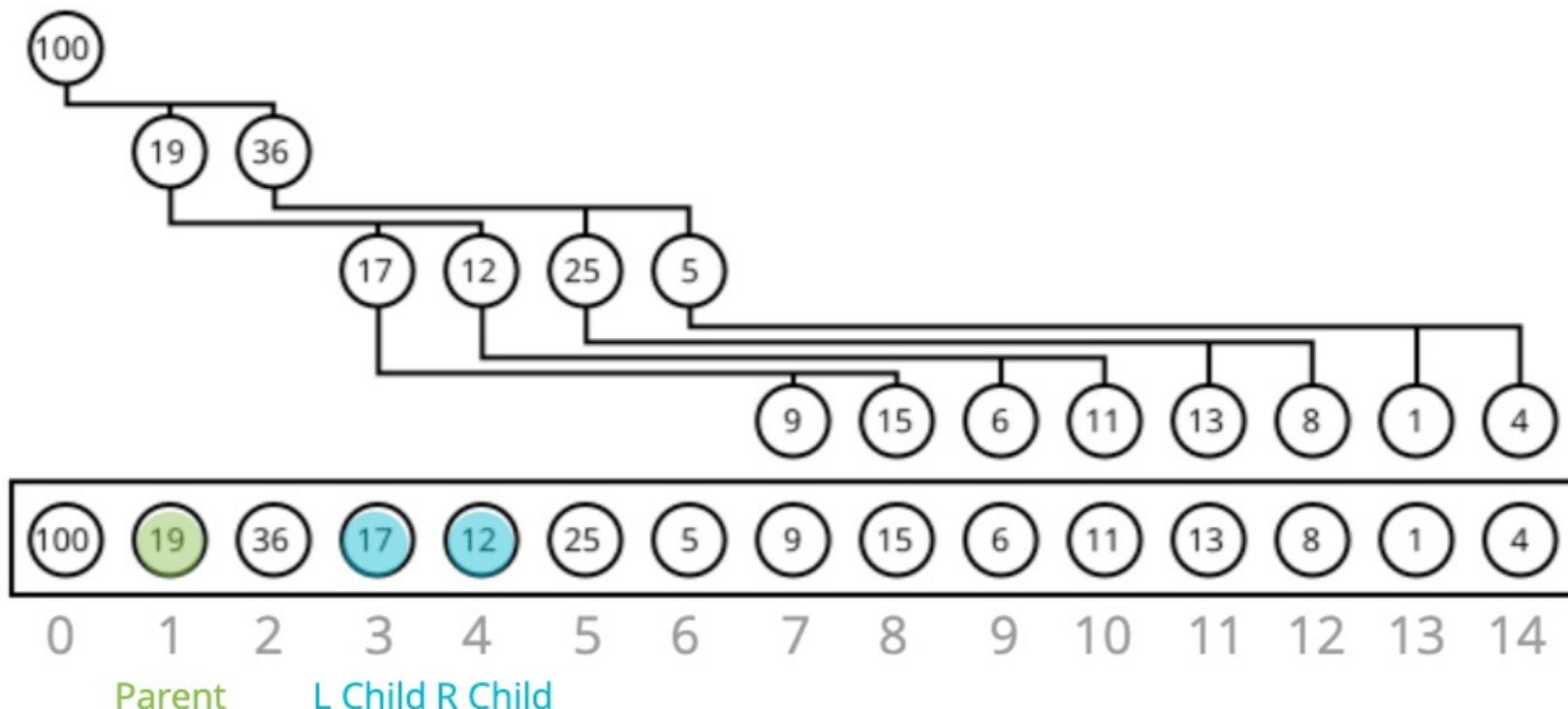
# REPRESENTING A HEAP



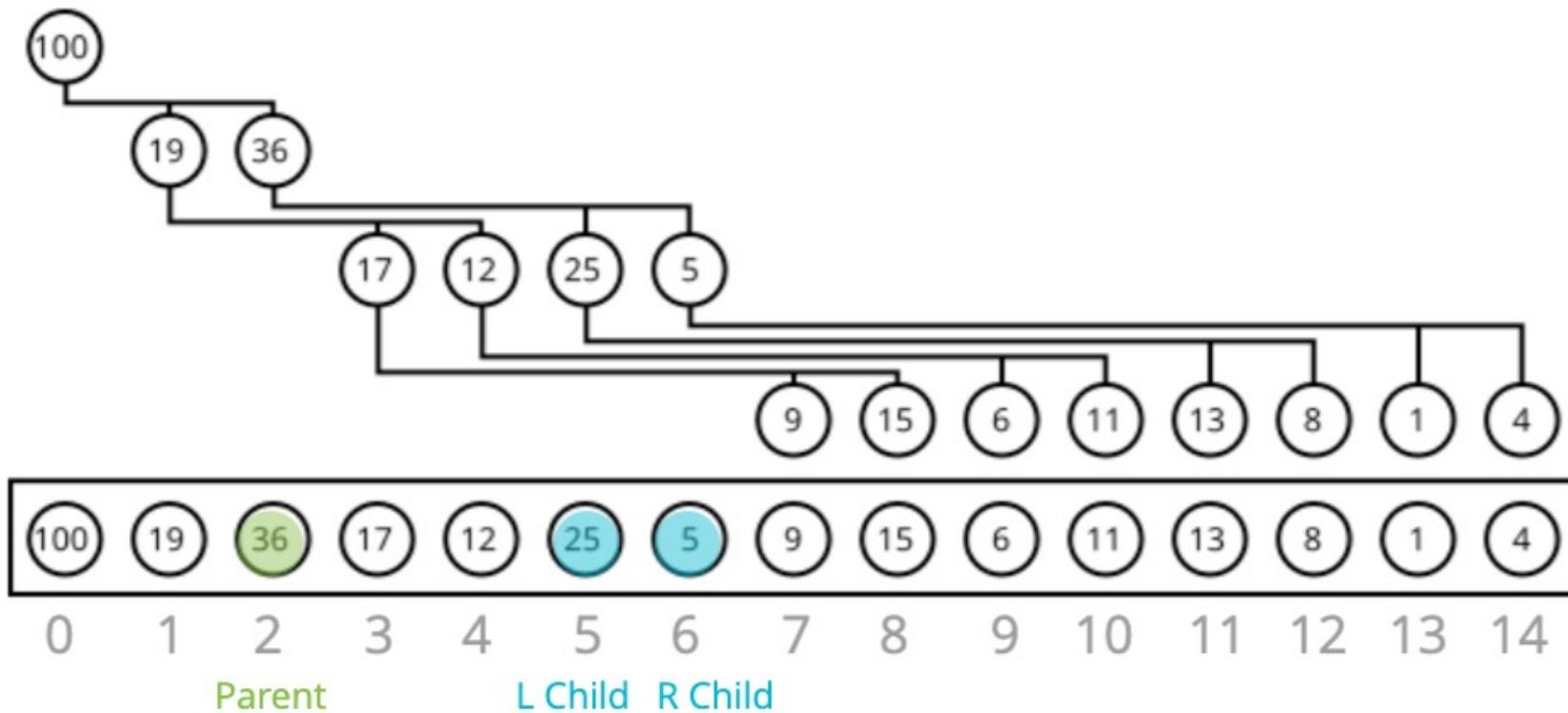
# REPRESENTING A HEAP



# REPRESENTING A HEAP



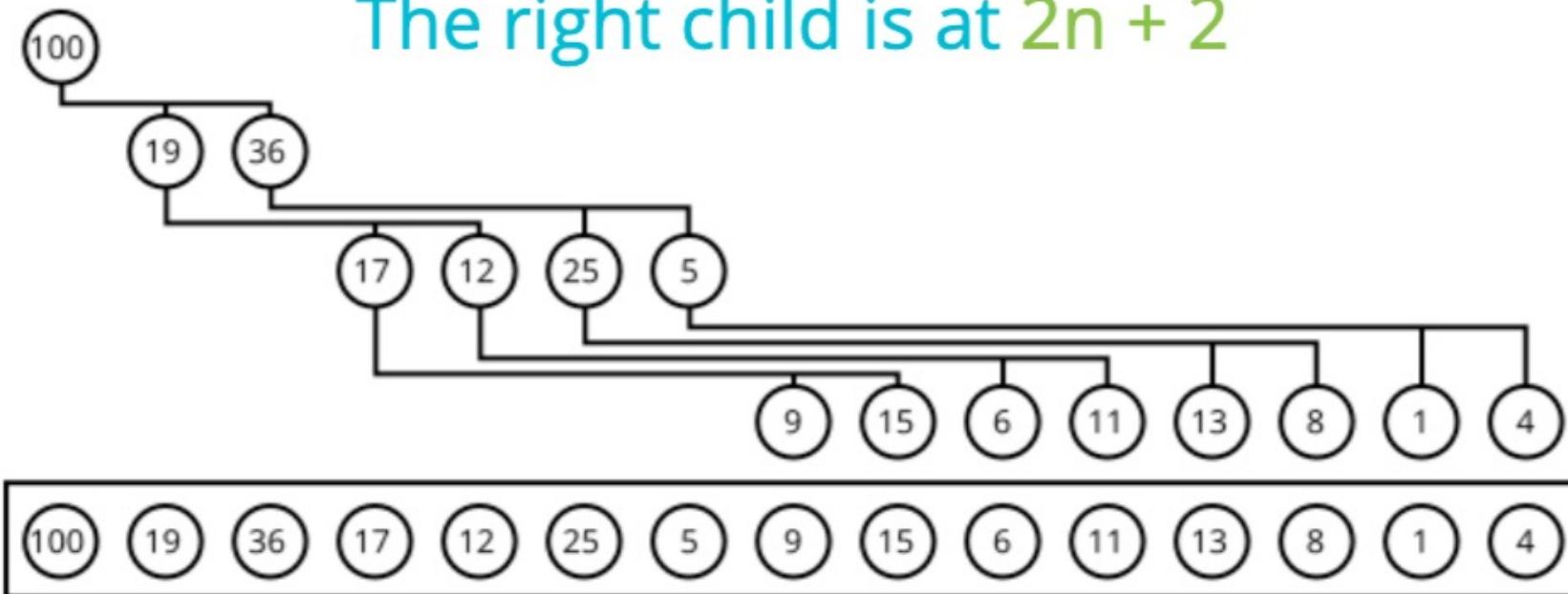
# REPRESENTING A HEAP



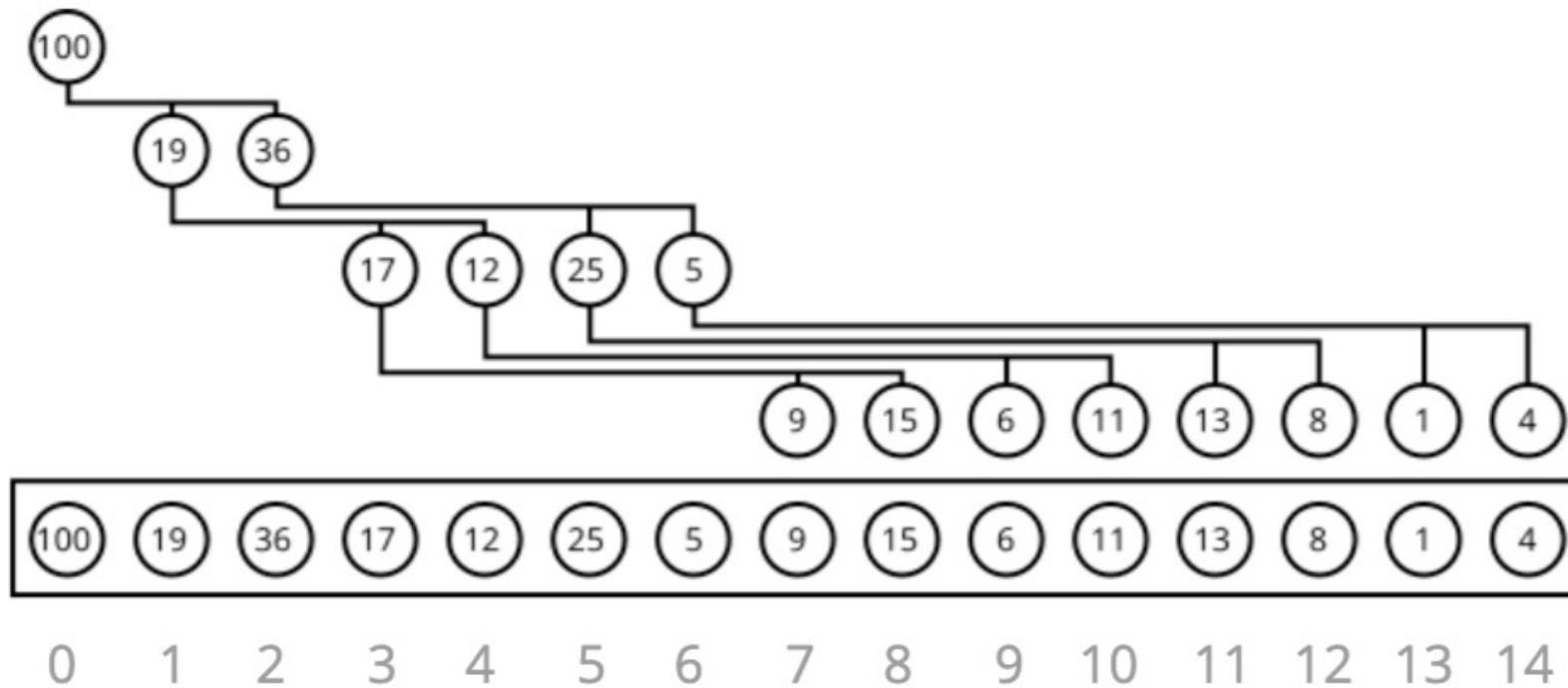
For any index of an array  $n$ ...

The left child is stored at  $2n + 1$

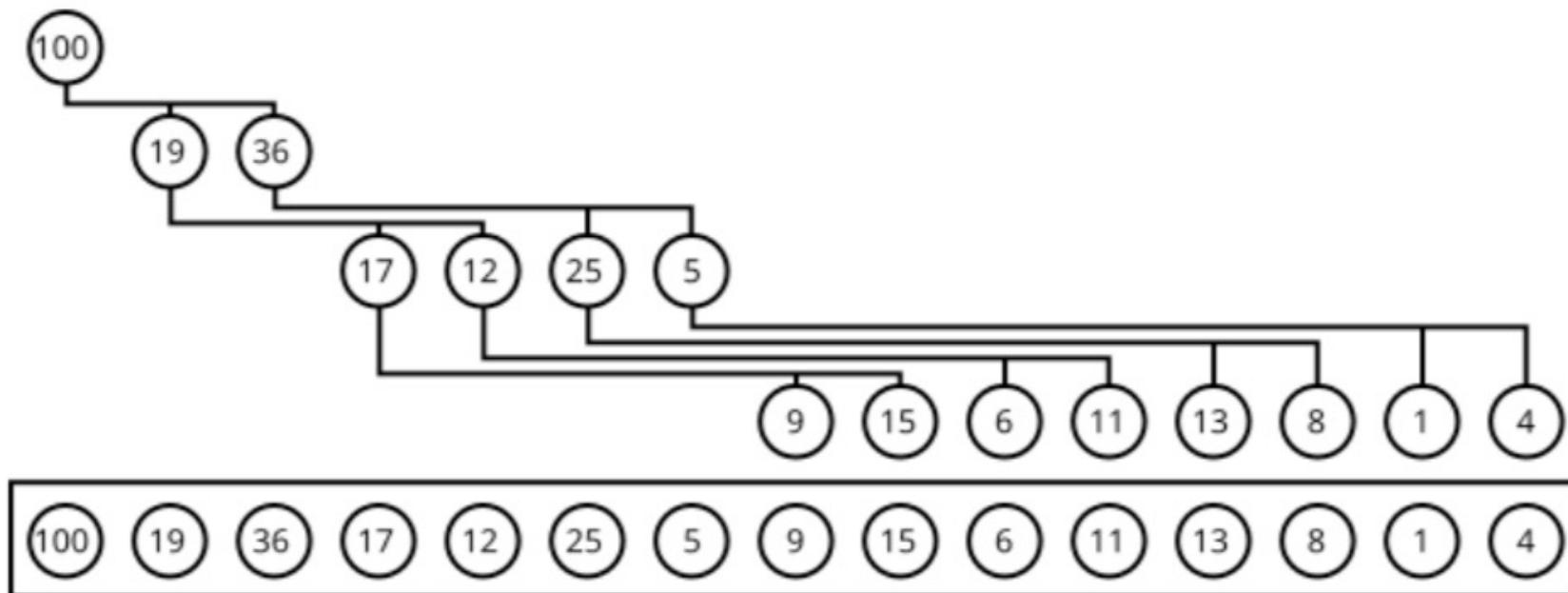
The right child is at  $2n + 2$



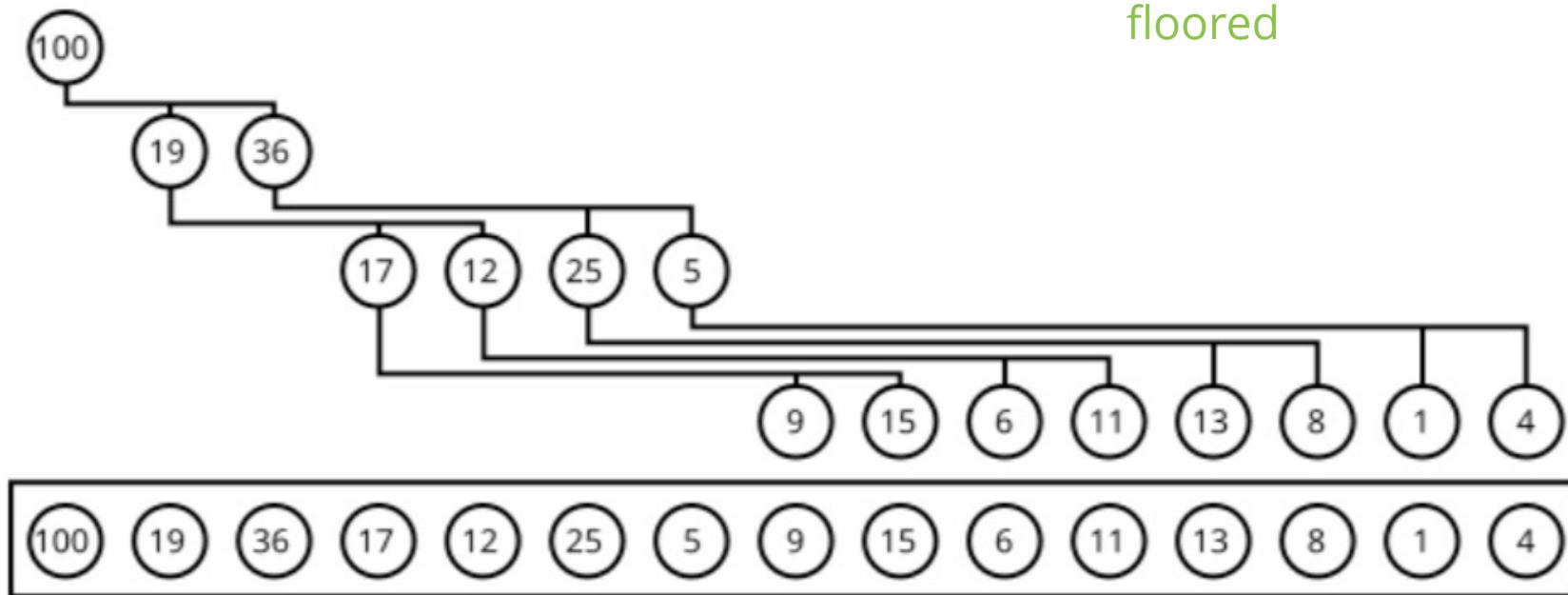
## WHAT IF WE HAVE A CHILD NODE AND WANT TO FIND ITS PARENT?



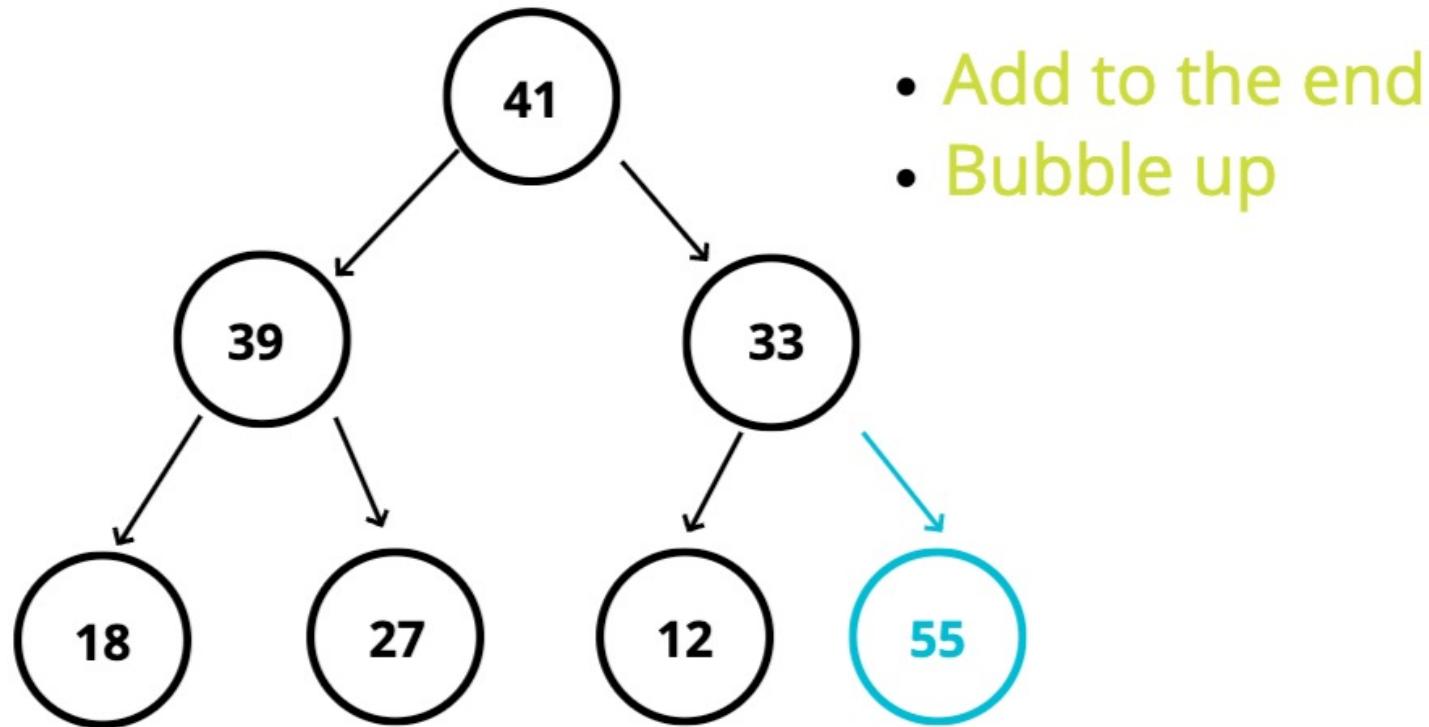
For any child node at index  $n$ ...  
Its parent is at index  $(n-1)/2$



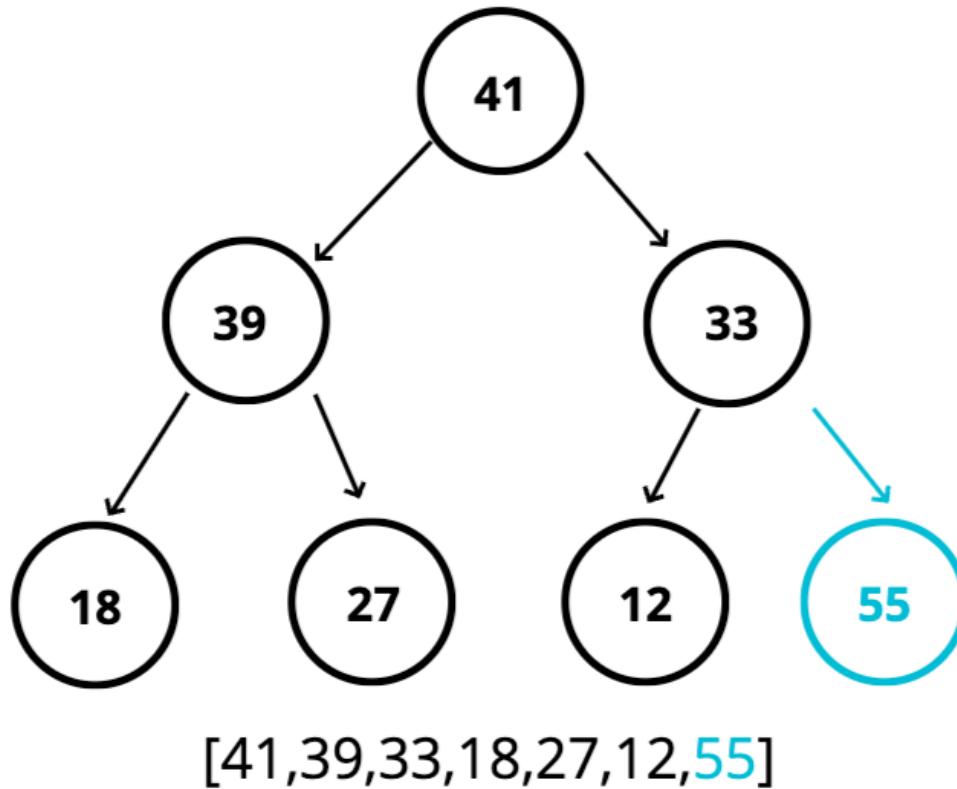
For any child node at index  $n$ ...  
Its parent is at index  $(n-1)/2$



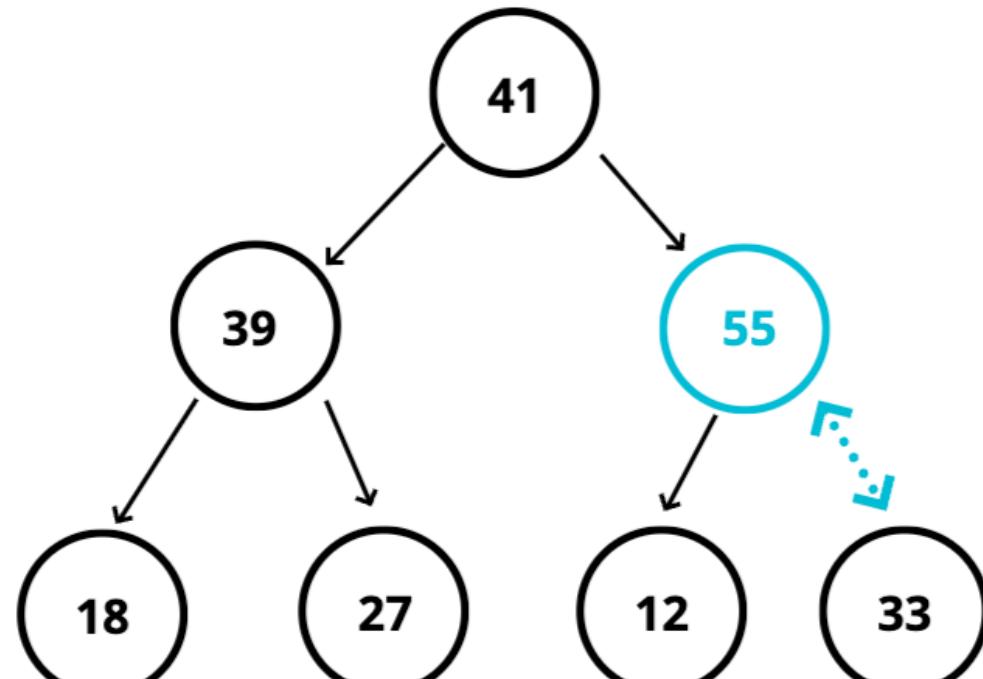
# Adding to a MaxBinaryHeap



# ADD TO THE END

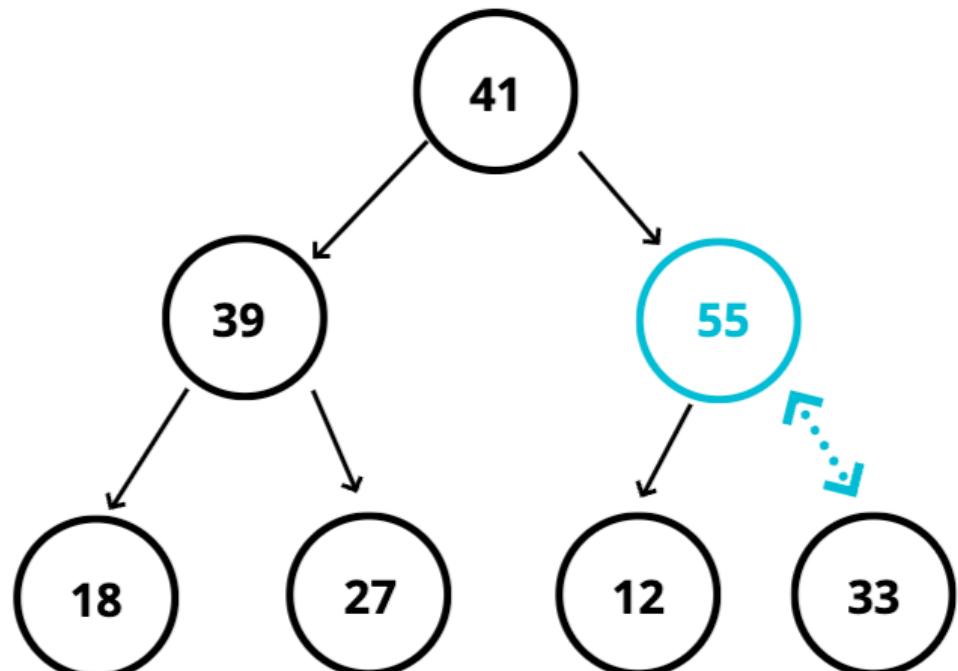


# BUBBLE UP



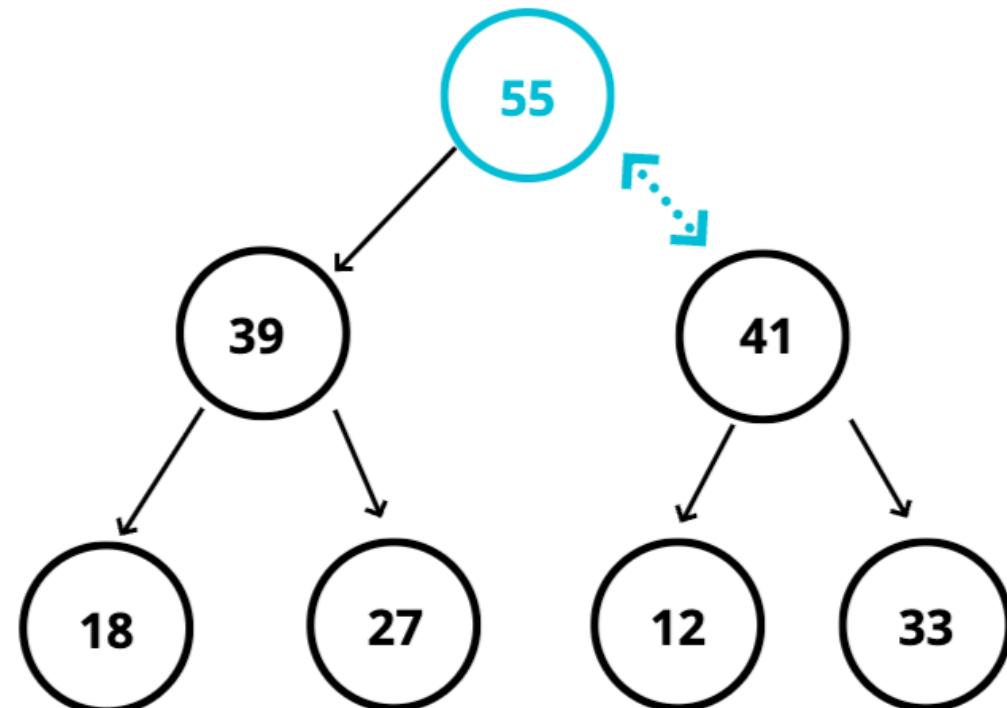
[41,39,33,18,27,12,55]

# BUBBLE UP



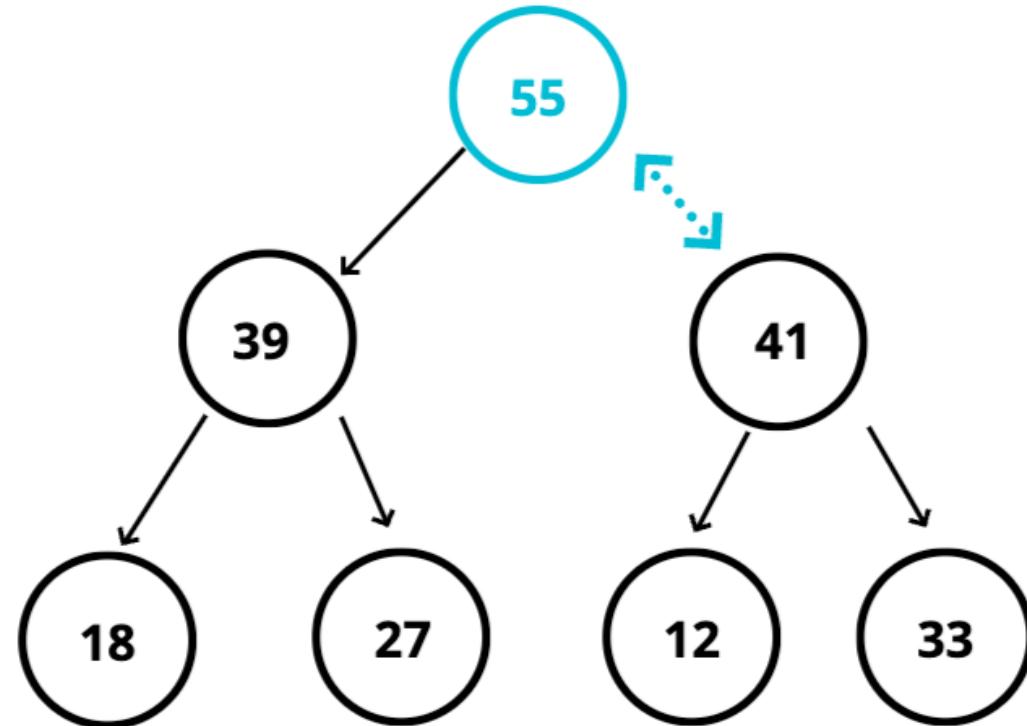
[41,39,**55**,18,27,12,**33**]

# BUBBLE UP



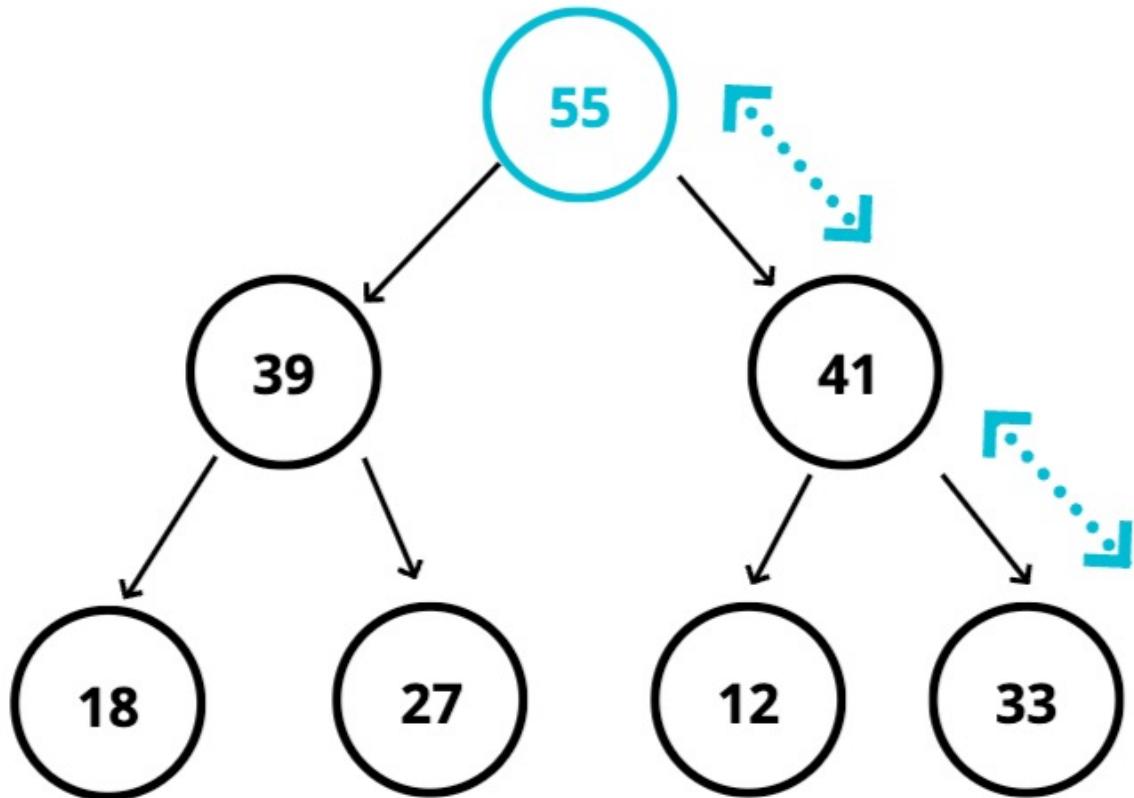
[41, 39, 55, 18, 27, 12, 33]

# BUBBLE UP



[55, 39, 41, 18, 27, 12, 33]

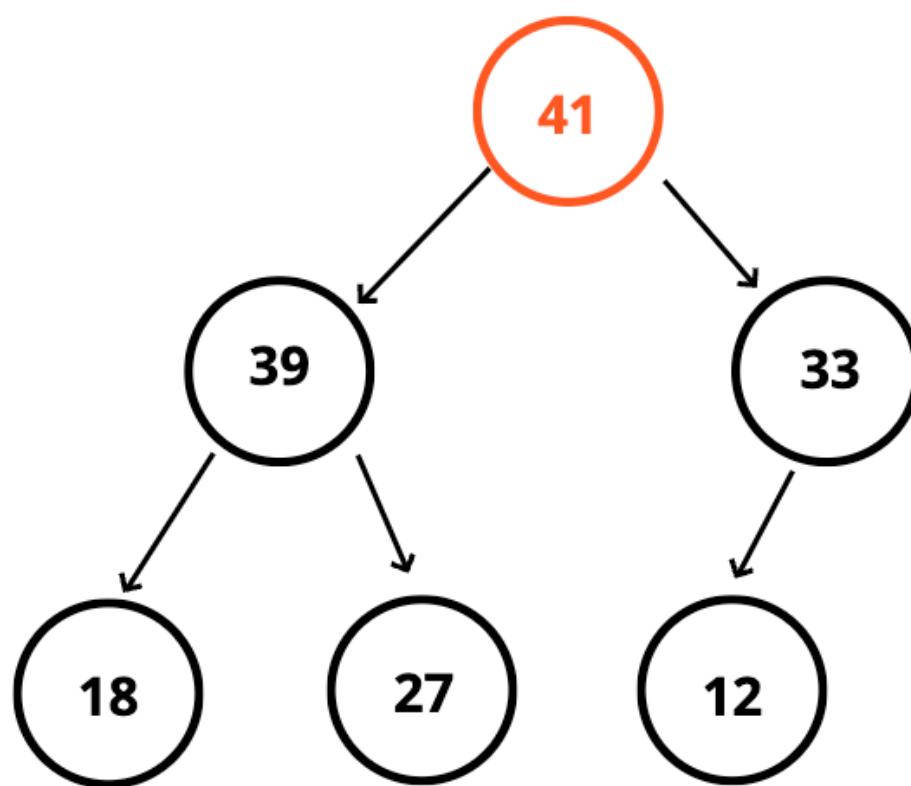
# Bubbling Up



# INSERT PSEUDOCODE

- Push the value into the values property on the heapBubble Up:  
Create a variable called index which is the length of the values property - 1
- Create a variable called parentIndex which is the floor of (index-1)/2
- Keep looping as long as the values element at the parentIndex is less than the values element at the child index
  - Swap the value of the values element at the parentIndex with the value of the element property at the child index
  - Set the index to be the parentIndex, and start over!

# REMOVING FROM A HEAP

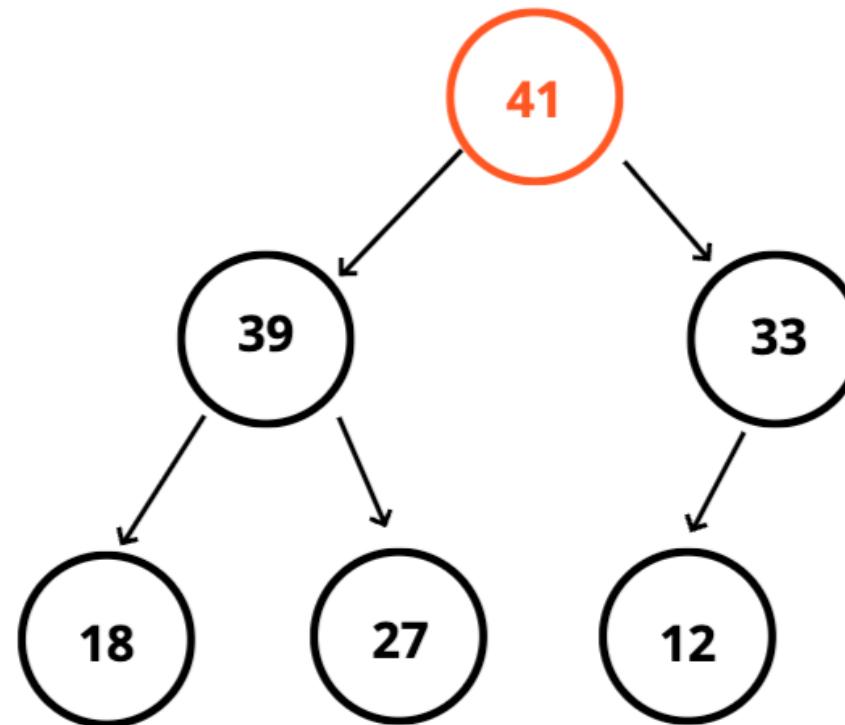


- Remove the root
- Replace with the most recently added
- Adjust (sink down)

# SINK DOWN?

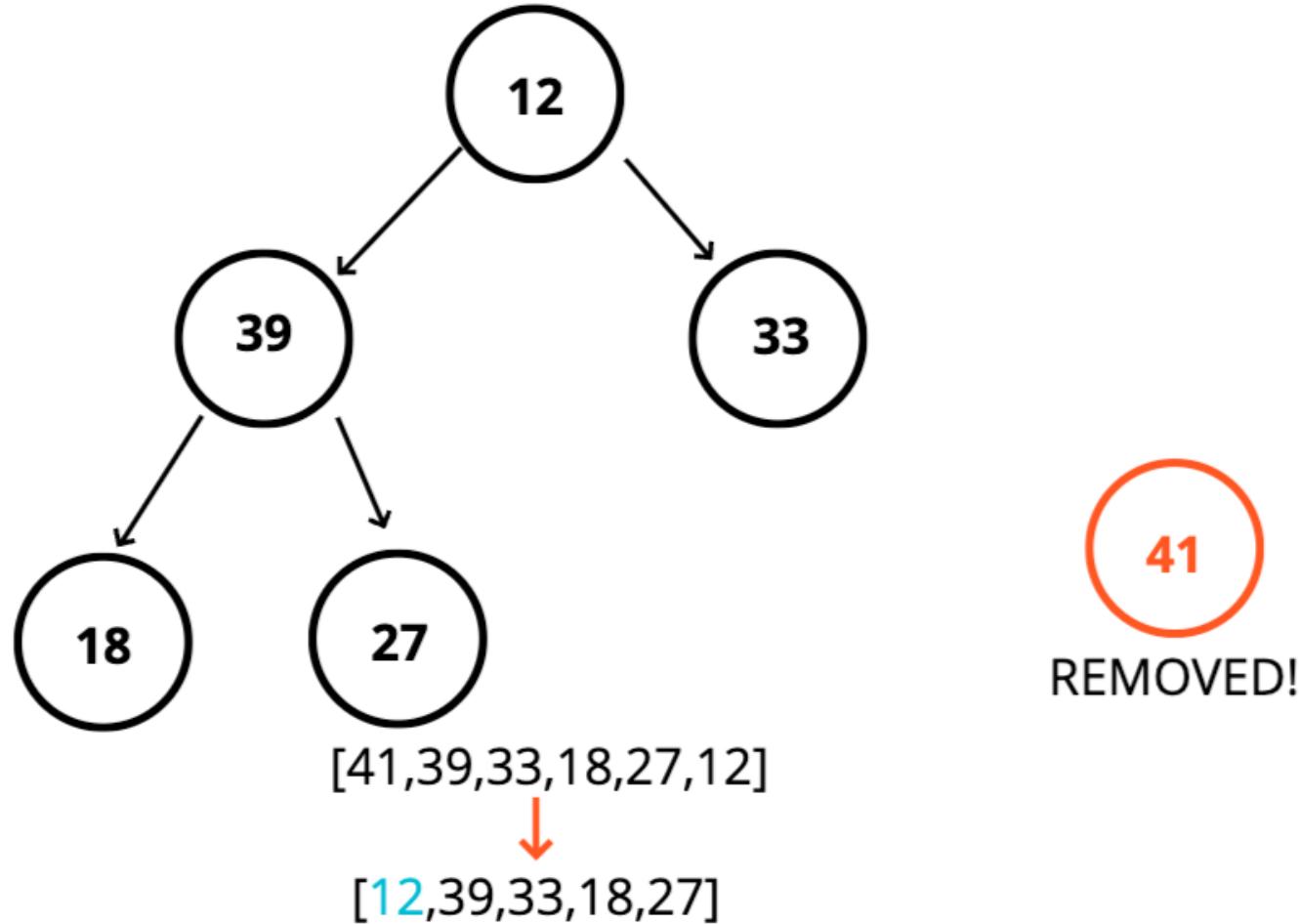
The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down*, *percolate-down*, *sift-down*, *trickle down*, *heapify-down*, *cascade-down*, and *extract-min/max*).

# REMOVE AND SWAP

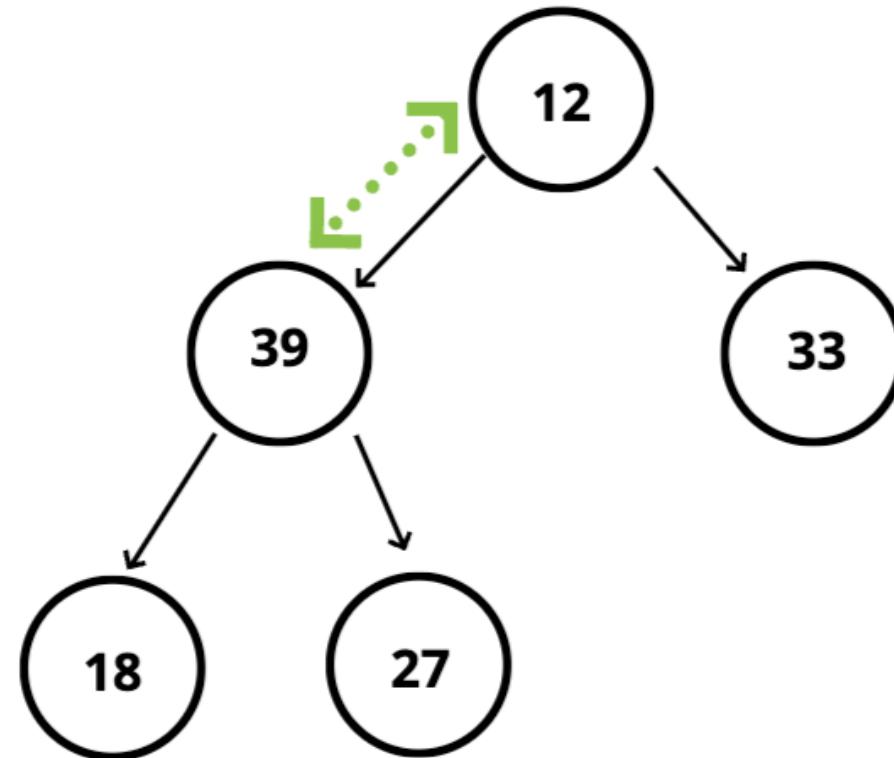


[41,39,33,18,27,12]

# REMOVE AND SWAP

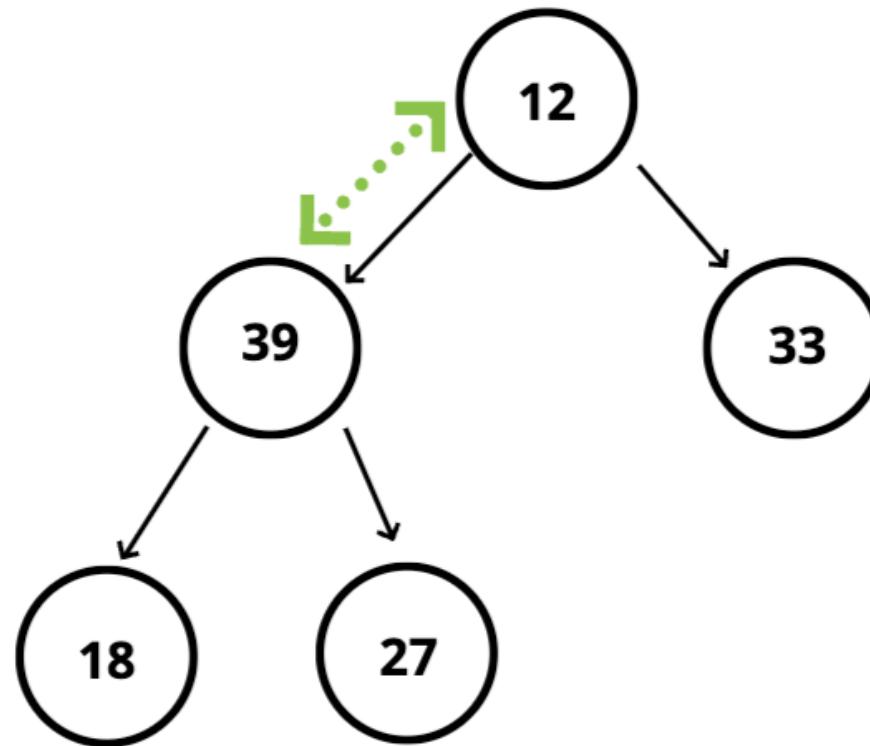


# SINKING DOWN



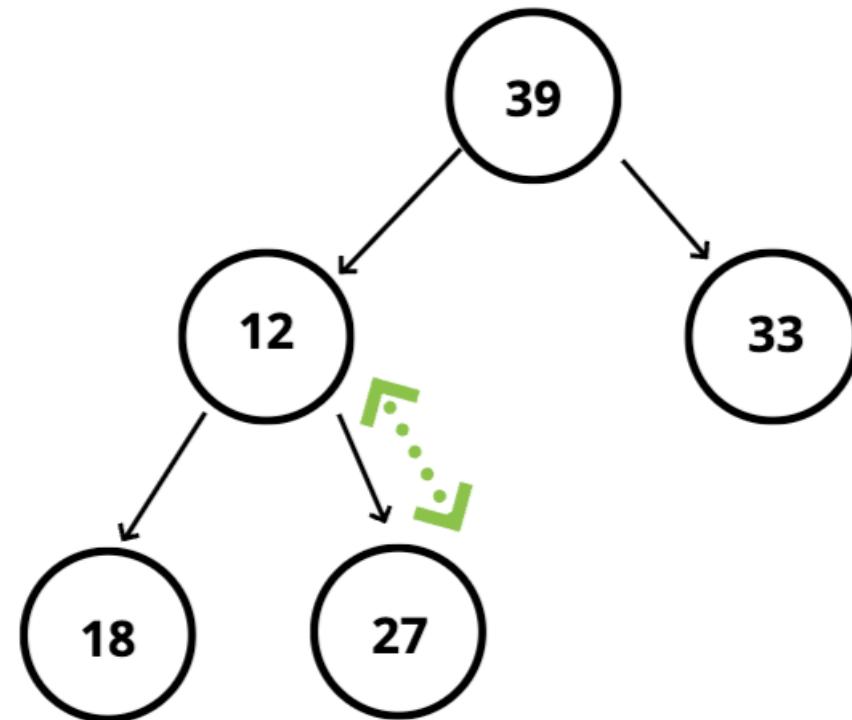
[12,39,33,18,27]

# SINKING DOWN



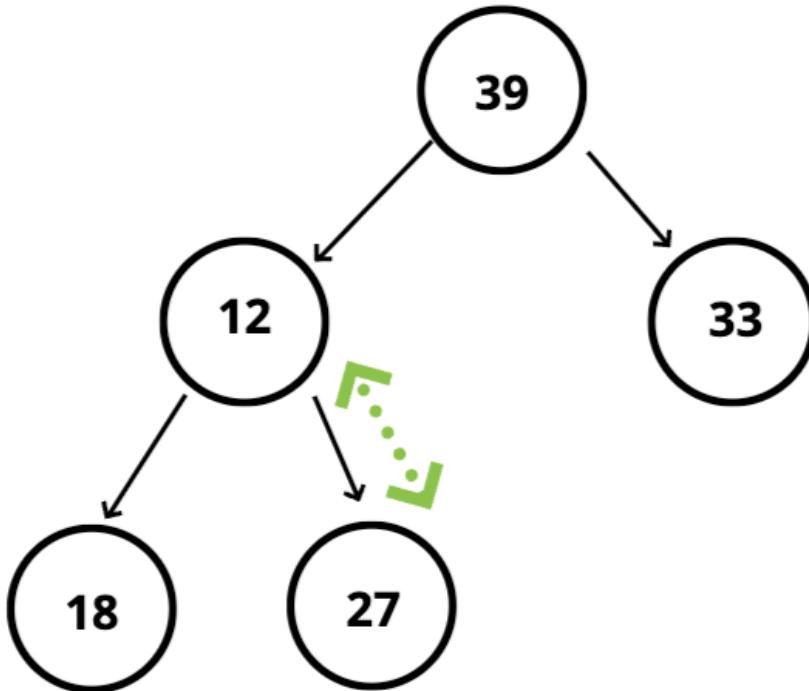
[39, 12, 33, 18, 27]

# SINKING DOWN



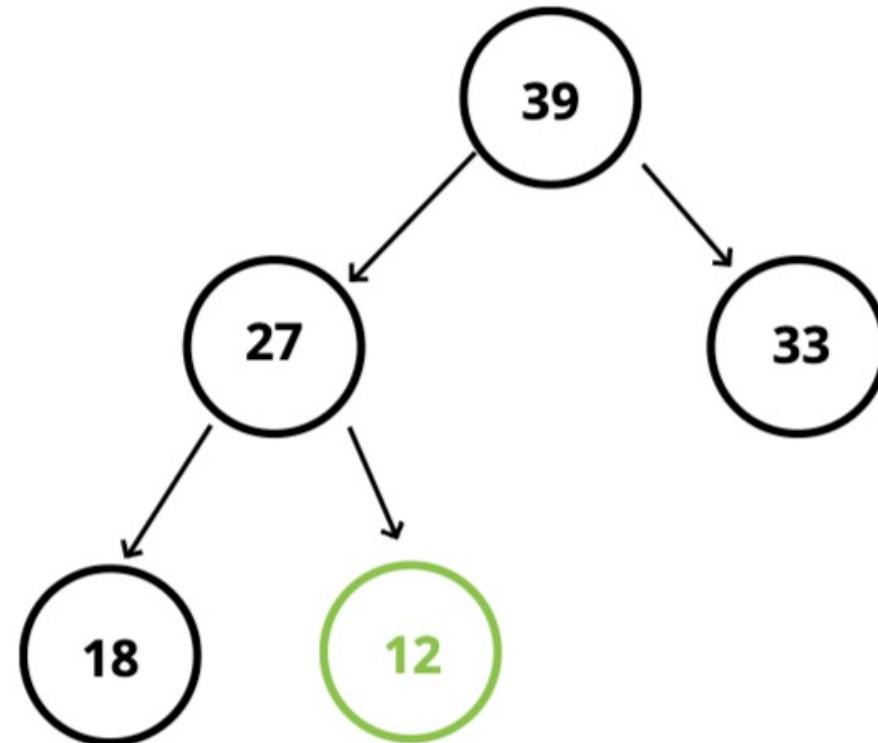
[39, 12, 33, 18, 27]

# SINKING DOWN



[39,**27**,33,18,**12**]

# SINKING DOWN



[39,27,33,18,12]

# REMOVING

(also called extractMax)

- Swap the first value in the values property with the last one
- Pop from the values property, so you can return the value at the end.
- Have the new root "sink down" to the correct spot...
  - Your parent index starts at 0 (the root)
  - Find the index of the left child:  $2 * \text{index} + 1$  (make sure its not out of bounds)
  - Find the index of the right child:  $2 * \text{index} + 2$  (make sure its not out of bounds)
  - If the left or right child is greater than the element...swap. If both left and right children are larger, swap with the largest child.
  - The child index you swapped to now becomes the new parent index.
  - Keep looping and swapping until neither child is larger than the element.
  - Return the old root!

# BUILDING A PRIORITY QUEUE

# WHAT IS A PRIORITY QUEUE?

A data structure where each element has a priority.

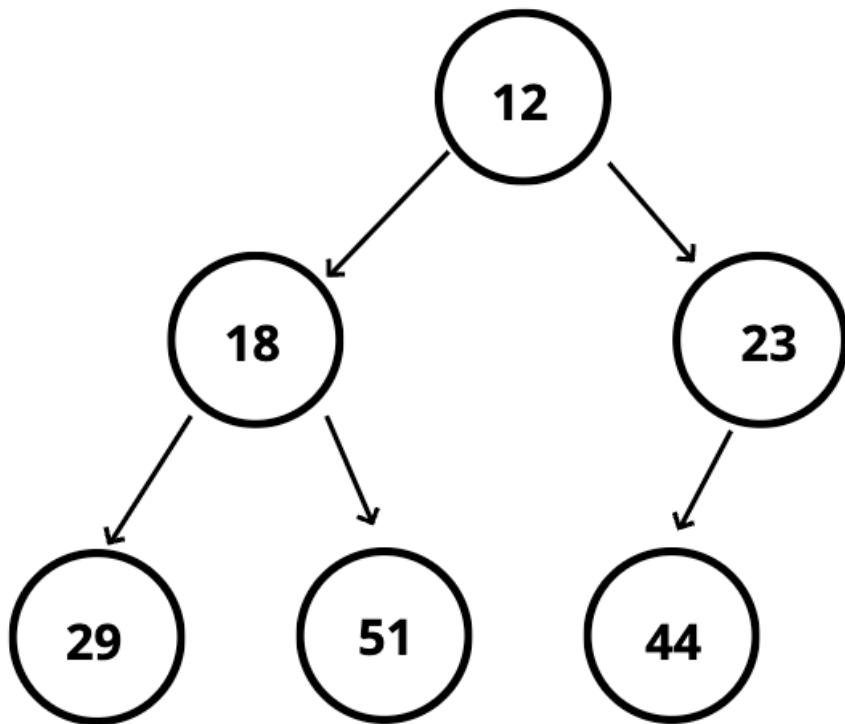
Elements with higher priorities are served before elements with lower priorities.

# Heapsort

We can sort an array in  **$O(n \log n)$**  time and  **$O(1)$**  space by making it a heap!

- Make the array a max heap (use **maxHeapify**)
- Loop over the array, swap the root node with last item in the array
- After swapping each item, run **maxHeapify** again to find the next root node
- Next loop you'll swap the root node with the second-to-last item in the array and run **maxHeapify** again.
- Once you've run out of items to swap, you have a sorted array!

# MinBinaryHeap



**Same idea, min  
values go  
upwards**

# Big O of Binary Heaps

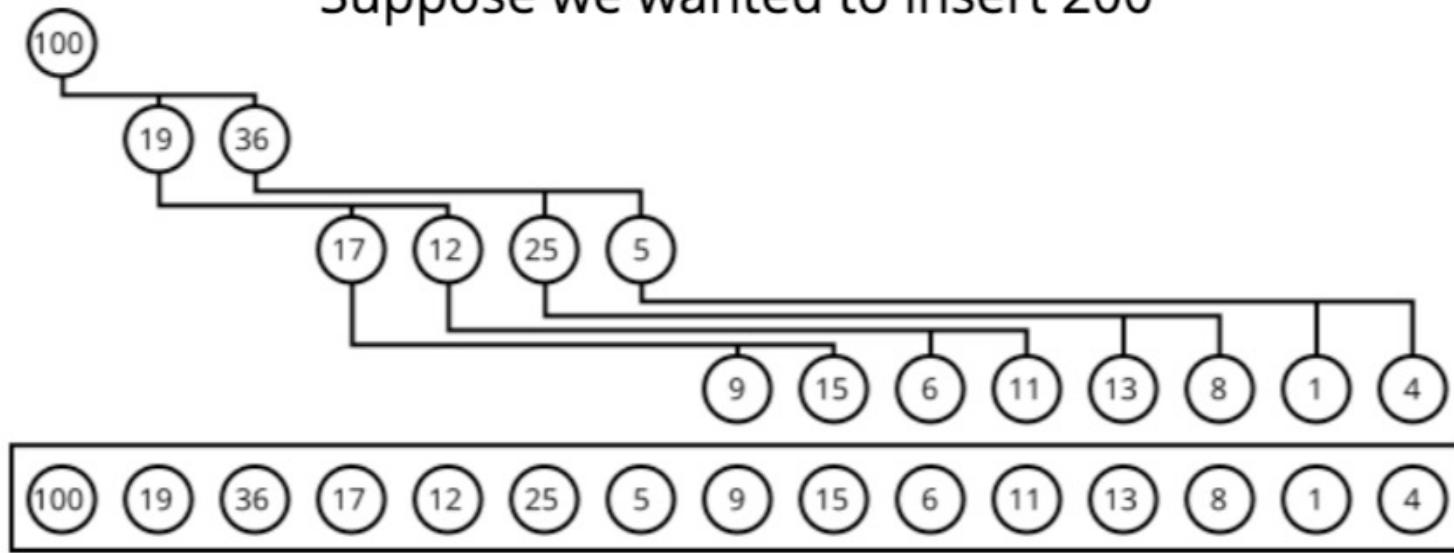
Insertion -  $O(\log N)$

Removal -  $O(\log N)$

Search -  $O(N)$

# WHY LOG(N)?

Suppose we wanted to insert 200



For 16 Elements....4 comparisons

# RECAP

- Binary Heaps are very useful data structures for sorting, and implementing other data structures like priority queues
- Binary Heaps are either MaxBinaryHeaps or MinBinaryHeaps with parents either being smaller or larger than their children
- With just a little bit of math, we can represent heaps using arrays!

# HASH TABLES

# OBJECTIVES

- Explain what a hash table is
- Define what a hashing algorithm
- Discuss what makes a good hashing algorithm
- Understand how collisions occur in a hash table
- Handle collisions using separate chaining or linear probing

# WHAT IS A HASH TABLE?

Hash tables are used to store *key-value* pairs.

They are like arrays, but the keys are not ordered.

Unlike arrays, hash tables are *fast* for all of the following operations:  
finding values, adding new values, and removing values!

# HASH TABLES IN THE WILD

Python has Dictionaries

JS has Objects and Maps\*

Java, Go, & Scala have Maps

Ruby has...Hashes

\* Objects have some restrictions, but are  
basically hash tables

# HASH TABLES

Introductory Example

How can we get human-readability  
*and* computer readability?

Computers don't know how to find an  
element at index *pink*!

Hash tables to the rescue!

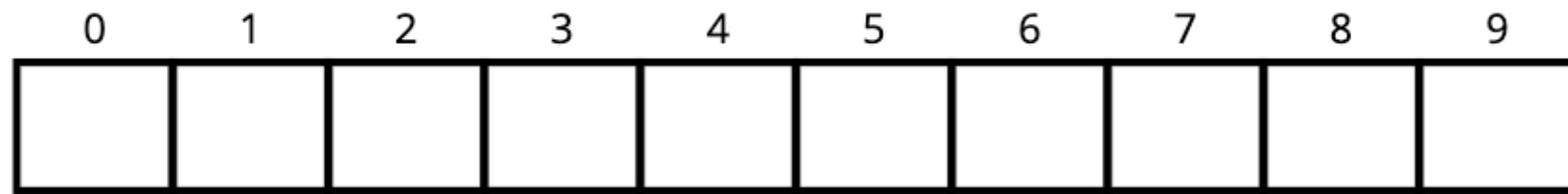
# THE HASH PART

To implement a hash table,  
we'll be using an array.

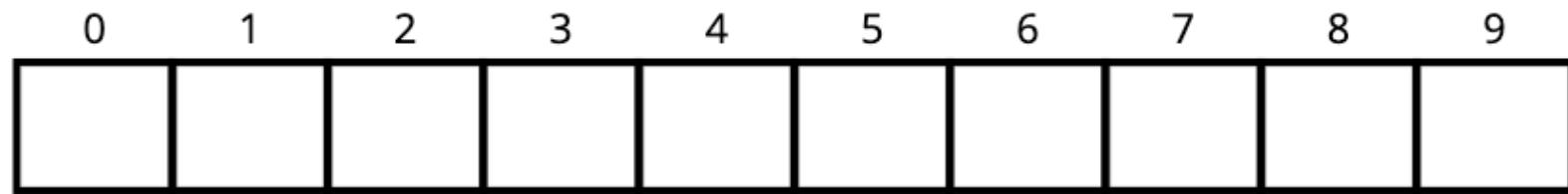
In order to look up values by key,  
we need a way to **convert keys**  
**into valid array indices.**

A function that performs this  
task is called a ***hash function***.

# HASHING CONCEPTUALLY

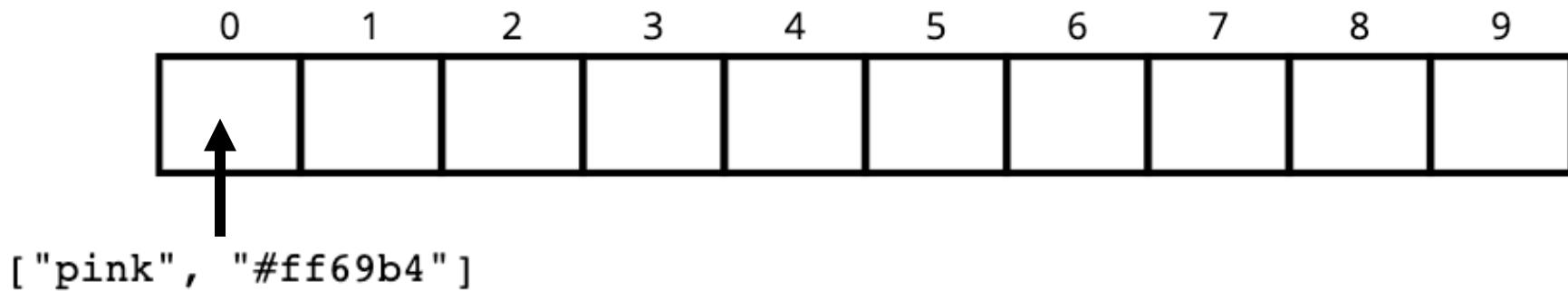


# HASHING CONCEPTUALLY

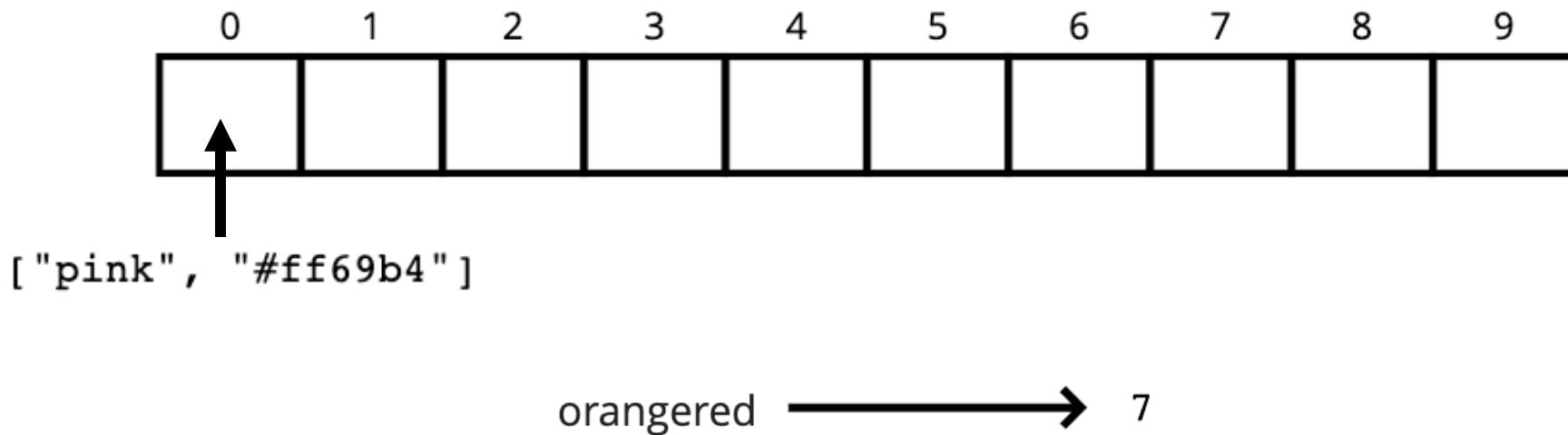


pink → 0

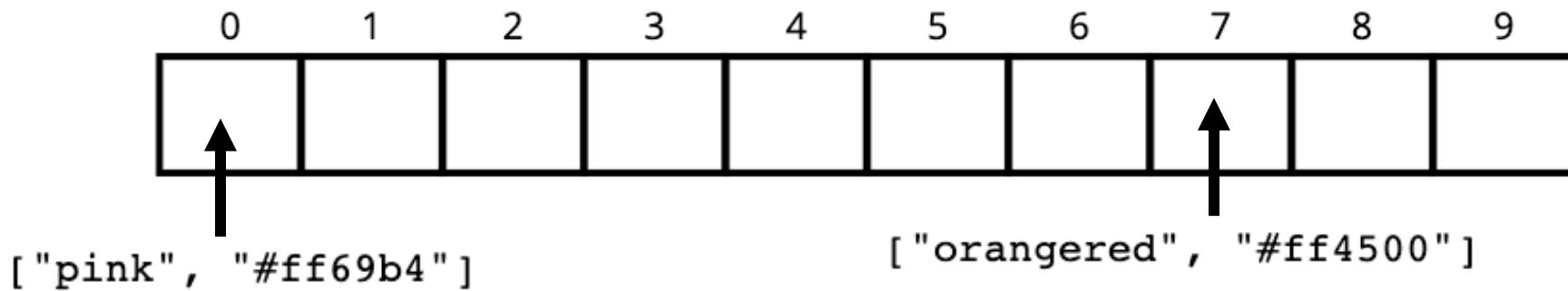
# HASHING CONCEPTUALLY



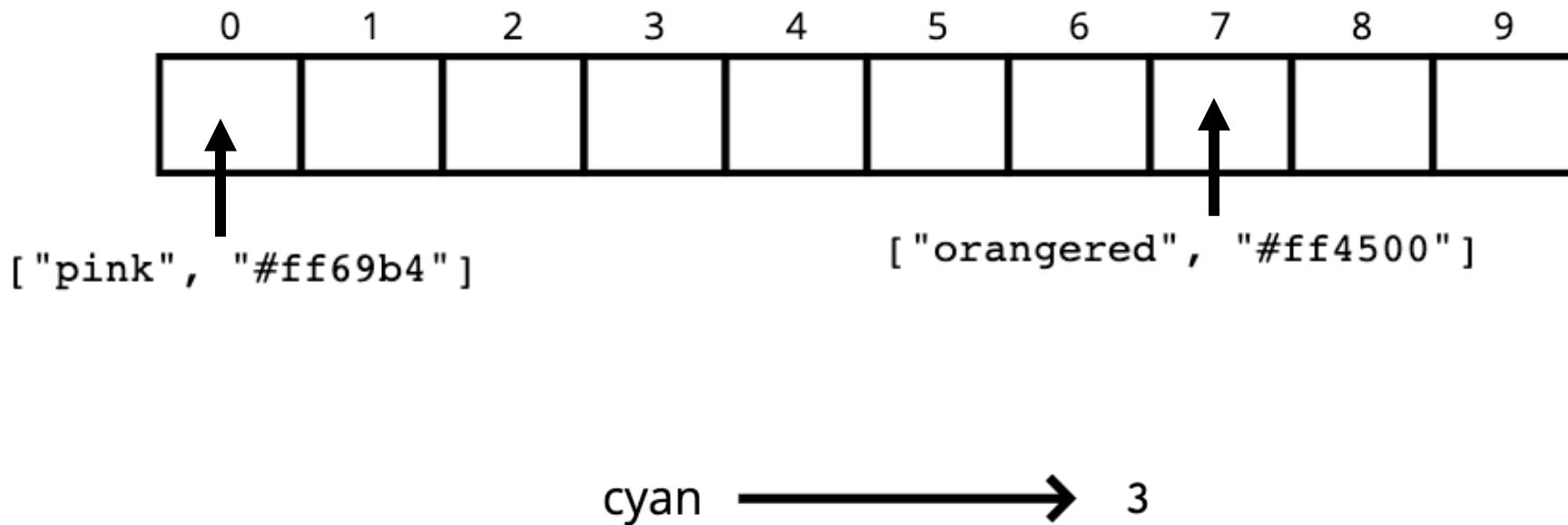
# HASHING CONCEPTUALLY



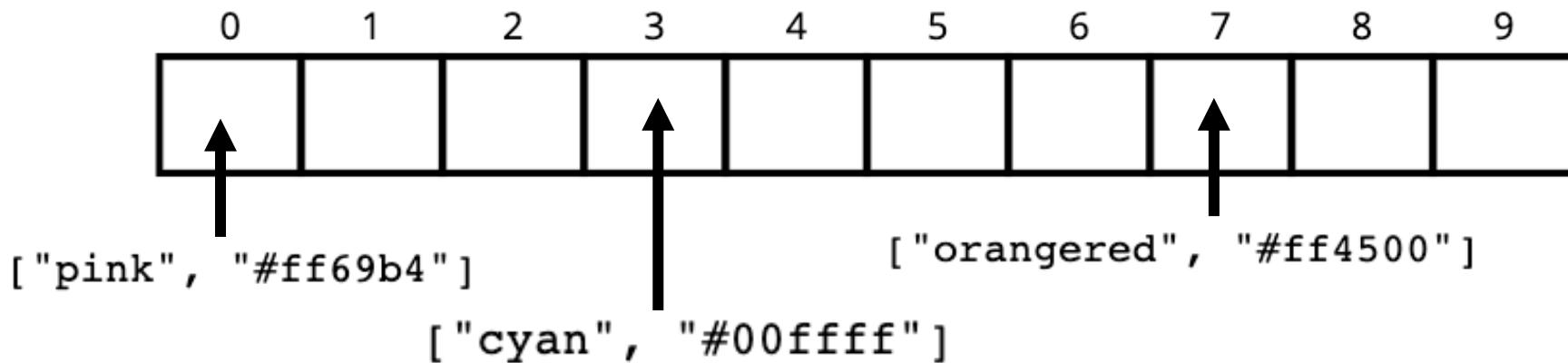
# HASHING CONCEPTUALLY



# HASHING CONCEPTUALLY



# HASHING CONCEPTUALLY



# WHAT MAKES A GOOD HASH?

(not a cryptographically secure one)

1. Fast (i.e. constant time)
2. Doesn't cluster outputs at specific indices, but distributes uniformly
3. Deterministic (same input yields same output)

# WHAT MAKES A GOOD HASH?

Fast

Non-Example

```
function slowHash(key) {  
    for (var i = 0; i < 10000; i++) {  
        console.log("everyday i'm hashing");  
    }  
    return key[0].charCodeAt(0);  
}
```

# WHAT MAKES A GOOD HASH?

Uniformly Distributes Values

Non-Example

```
function sameHashedValue(key) {  
    return 0;  
}
```

# WHAT MAKES A GOOD HASH?

Deterministic

Non-Example

```
function randomHash(key) {  
    return Math.floor(Math.random() * 1000)  
}
```

# WHAT MAKES A GOOD HASH?

## Simple Hash Example

Here's a hash that works on *strings only*:

```
function hash(key, arrayLen) {  
  let total = 0;  
  for (let char of key) {  
    // map "a" to 1, "b" to 2, "c" to 3, etc.  
    let value = char.charCodeAt(0) - 96  
    total = (total + value) % arrayLen;  
  }  
  return total;  
}
```

```
hash("pink", 10); // 0  
hash("orangered", 10); // 7  
hash("cyan", 10); // 3
```

# REFINING OUR HASH

Problems with our current hash

1. Only hashes strings (we won't worry about this)
2. Not constant time - linear in key length
3. Could be a little more random

# Hashing Revisited

```
function hash(key, arrayLen) {
    let total = 0;
    for (let i = 0; i < key.length; i++) {
        let char = key[i];
        let value = char.charCodeAt(0) - 96
        total = (total + value) % arrayLen;
    }
    return total;
}
```

```
function hash(key, arrayLen) {
    let total = 0;
    let WEIRD_PRIME = 31;
    for (let i = 0; i < Math.min(key.length, 100); i++) {
        let char = key[i];
        let value = char.charCodeAt(0) - 96
        total = (total * WEIRD_PRIME + value) % arrayLen;
    }
    return total;
}
```

# Prime numbers? wut.

The prime number in the hash is helpful in spreading out the keys more uniformly.

It's also helpful if the array that you're putting values into has a prime length.

You don't need to know why. (Math is complicated!)  
But here are some links if you're curious.

[Why do hash functions use prime numbers?](#)

[Does making array size a prime number help in hash table implementation?](#)

# Dealing with Collisions

Even with a large array and a great hash function, collisions are inevitable.

There are many strategies for dealing with collisions, but we'll focus on two:

1. Separate Chaining
2. Linear Probing

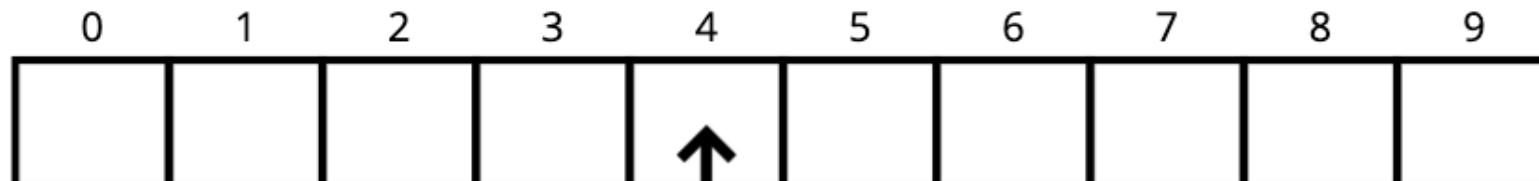
# Separate Chaining

With *separate chaining*, at each index in our array we store values using a more sophisticated data structure (e.g. an array or a linked list).

This allows us to store multiple key-value pairs at the same index.

# Separate Chaining

Example

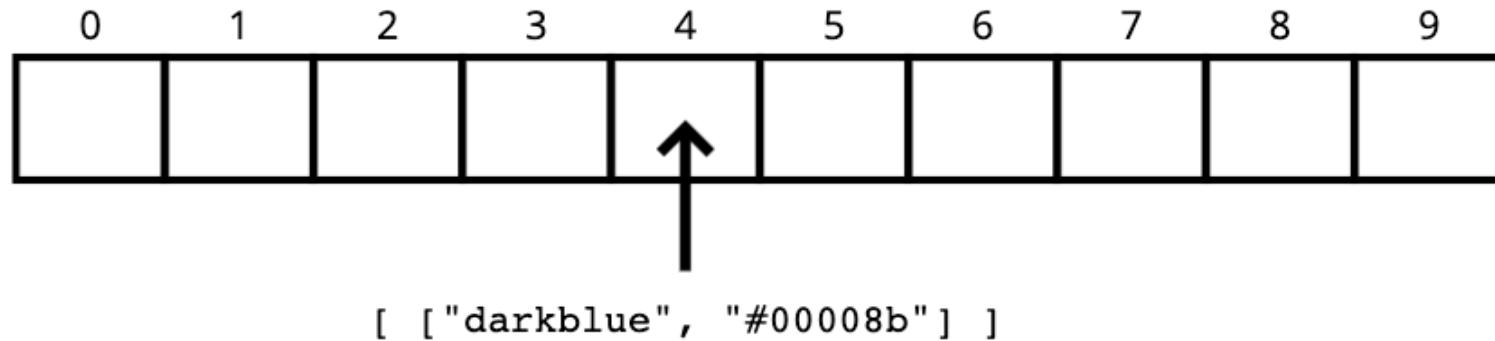


[ [ "darkblue", "#00008b" ] ]

darkblue → 4

# Separate Chaining

Example

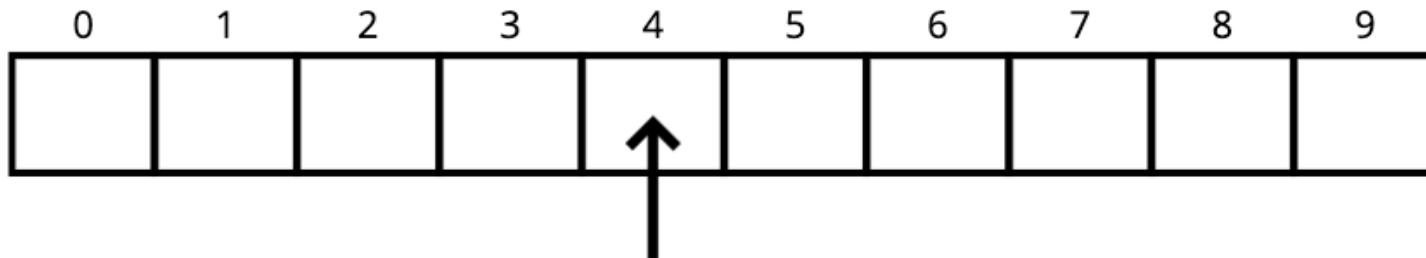


darkblue → 4

salmon → 4

# Separate Chaining

Example



darkblue → 4

salmon → 4

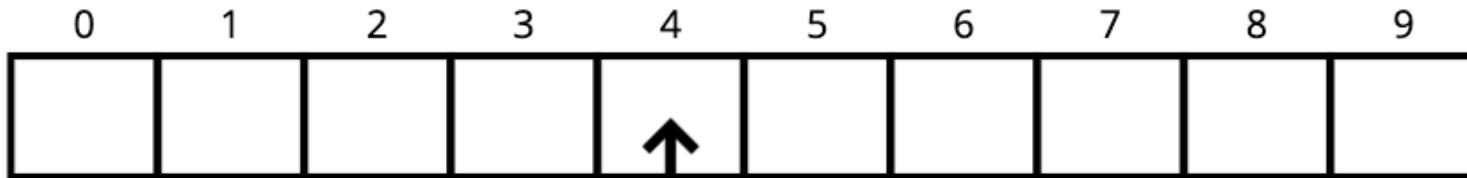
# Linear Probing

With *linear probing*, when we find a collision, we search through the array to find the next empty slot.

Unlike with separate chaining, this allows us to store a single key-value at each index.

# Linear Probing

Example

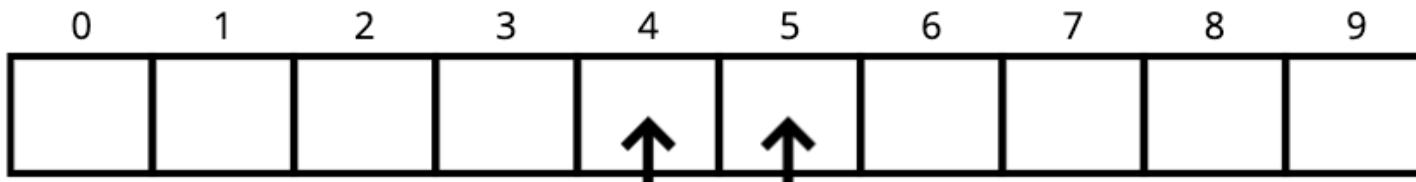


[ "darkblue", "#00008b" ]

darkblue → 4

# Linear Probing

Example



[ "salmon", "#fa8072" ]

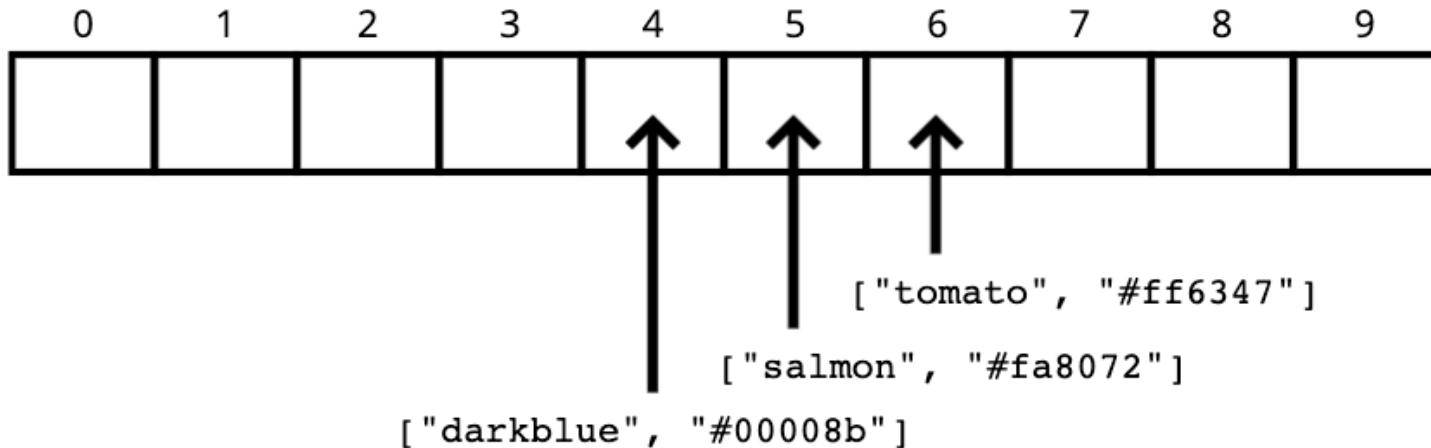
[ "darkblue", "#00008b" ]

darkblue → 4

salmon → 4

# Linear Probing

Example



darkblue → 4

salmon → 4

tomato → 4

# A HashTable Class

```
class HashTable {
    constructor(size=53){
        this.keyMap = new Array(size);
    }

    _hash(key) {
        let total = 0;
        let WEIRD_PRIME = 31;
        for (let i = 0; i < Math.min(key.length, 100); i++) {
            let char = key[i];
            let value = char.charCodeAt(0) - 96
            total = (total * WEIRD_PRIME + value) % this.keyMap.length;
        }
        return total;
    }
}
```

# Set / Get

## set

1. Accepts a key and a value
2. Hashes the key
3. Stores the key-value pair in the hash table array via separate chaining

## get

1. Accepts a key
2. Hashes the key
3. Retrieves the key-value pair in the hash table
4. If the key isn't found, returns `undefined`

# Keys / Values

**keys**

1. Loops through the hash table array and returns an array of keys in the table

**values**

1. Loops through the hash table array and returns an array of values in the table

# BIG O of HASH TABLES

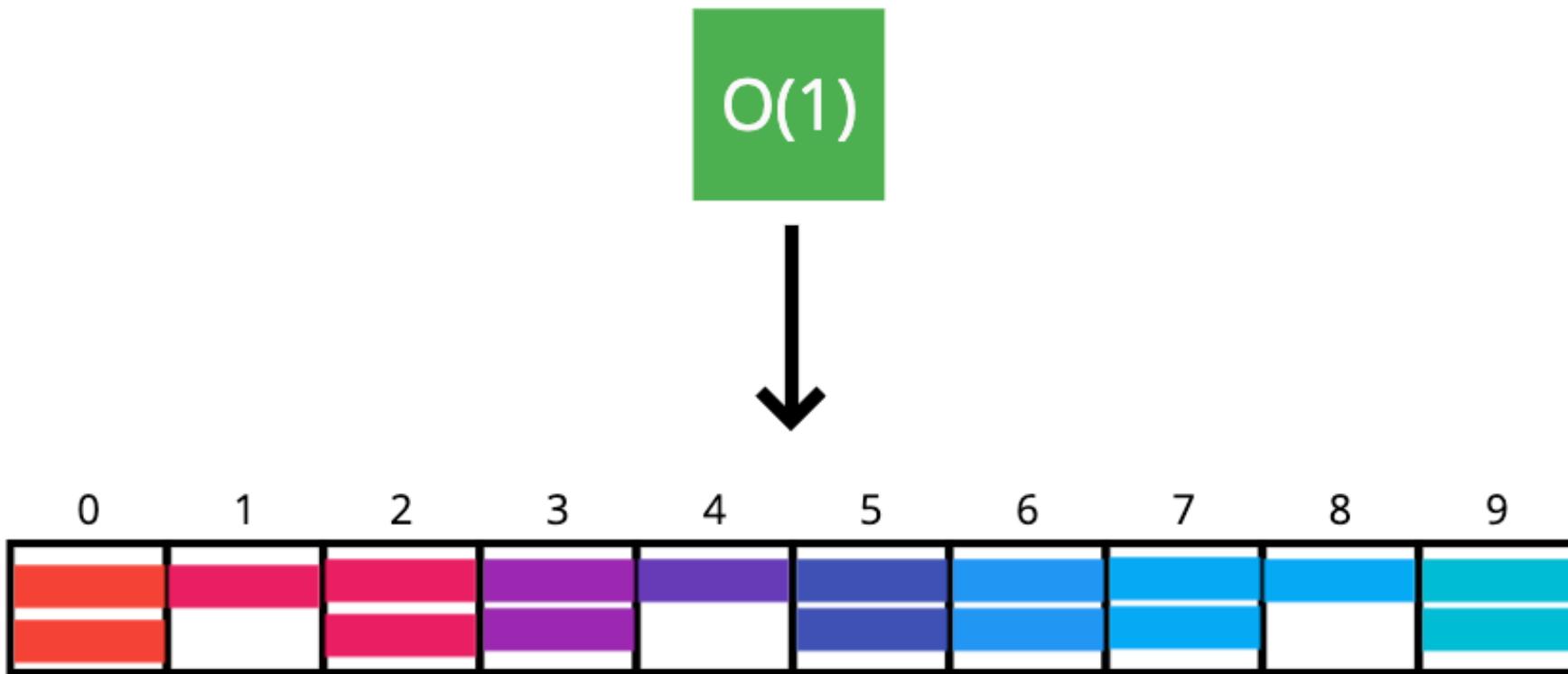
(average case)

- Insert:  $O(1)$
- Deletion:  $O(1)$
- Access:  $O(1)$

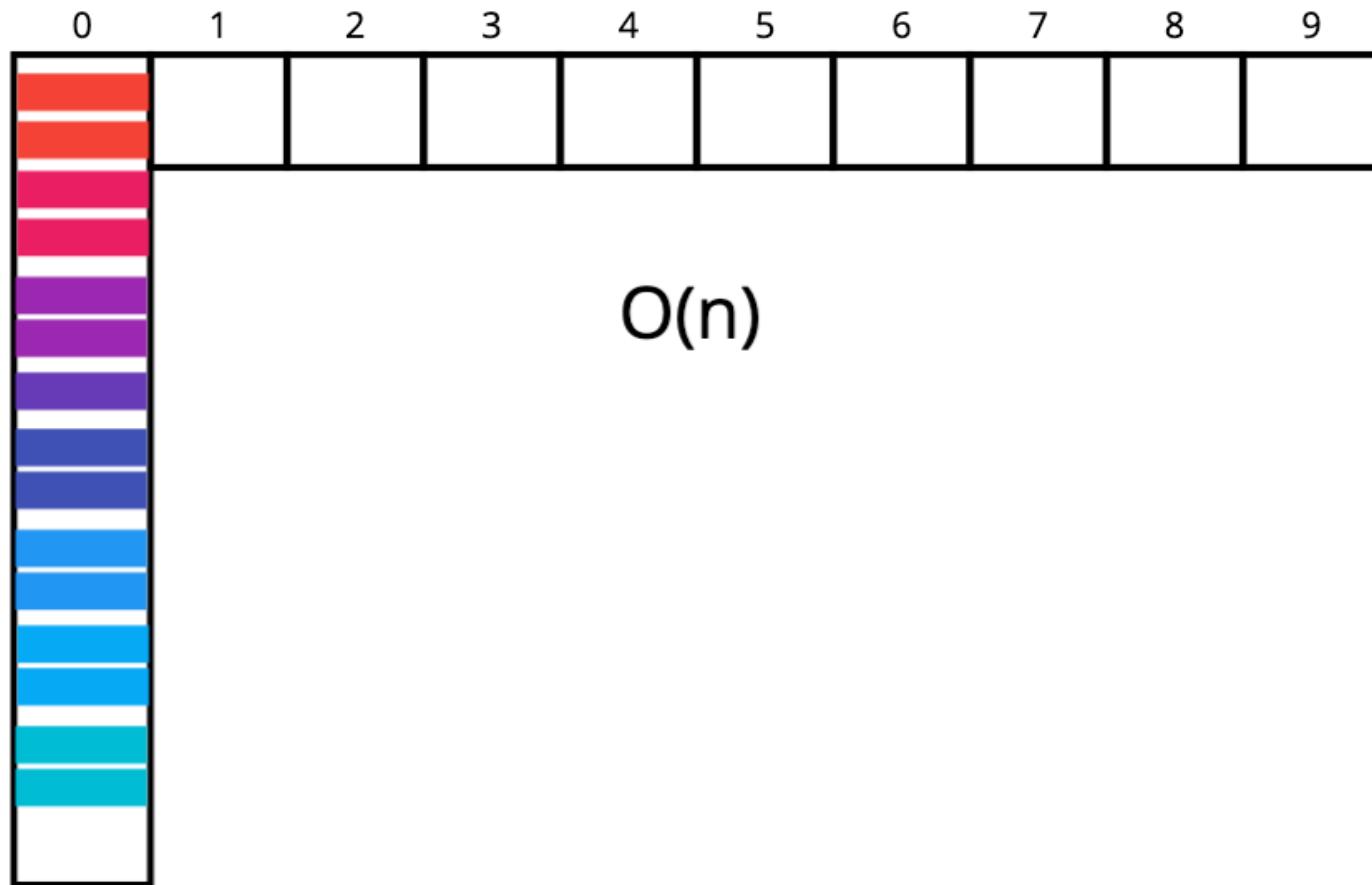
A good hash function



A good hash function



With the world's worst hash function...



# Recap

- Hash tables are collections of key-value pairs
- Hash tables can find values quickly given a key
- Hash tables can add new key-values quickly
- Hash tables store data in a large array, and work by *hashing* the keys
- A good hash should be fast, distribute keys uniformly, and be deterministic
- Separate chaining and linear probing are two strategies used to deal with two keys that hash to the same index
- When in doubt, use a hash table!