Introduction to Big-Oh

In O(1) time

Presented by: **Shawn Bullock**
http://algoacad.me

Overview

- Data Structures & Algorithms
- Comparison
- Categories
    - $O(1)$, $O(\log n)$, $O(n)$, $O(n^2)$, $O(n^3)$, $O(m^n)$
- Analysis
- Your own implementation review

Data Structures & Algorithms

- Scary Stuff™
- Voodoo
- Advanced
- Intense

Data Structures & Algorithms

All Myths

Data Structures & Algorithms

- Dream Job
- Prestige
- Seniority

Data Structures & Algorithms

- Data Structures describe how to structure data and organize it in memory
    - Mainly concerned with storage
    - Data Structures don’t `function` (or execute)

- Just a few basic `families` of Data Structures
    - Arrays / Vectors / Lists
    - Stacks / Queues
    - Trees / Graphs
    - Sets / Hash Tables

Data Structures & Algorithms

- A good data structure can store any kind of data

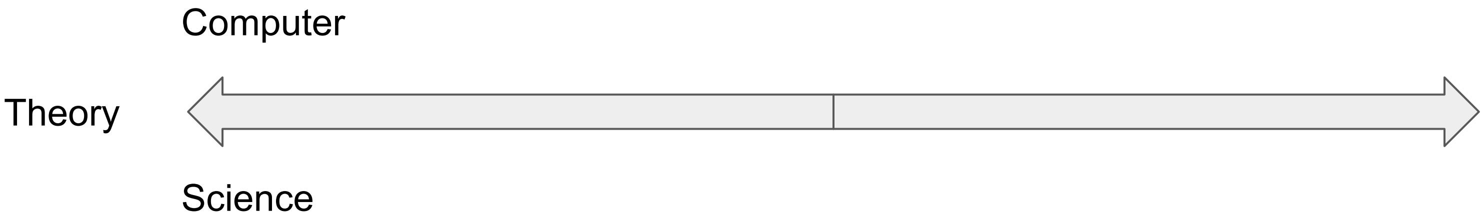
Data Structures & Algorithms

- Primarily concerned with `what` to do with the data stored in a structure
    - And `how` it's done
    - Algorithms do compute (or execute)

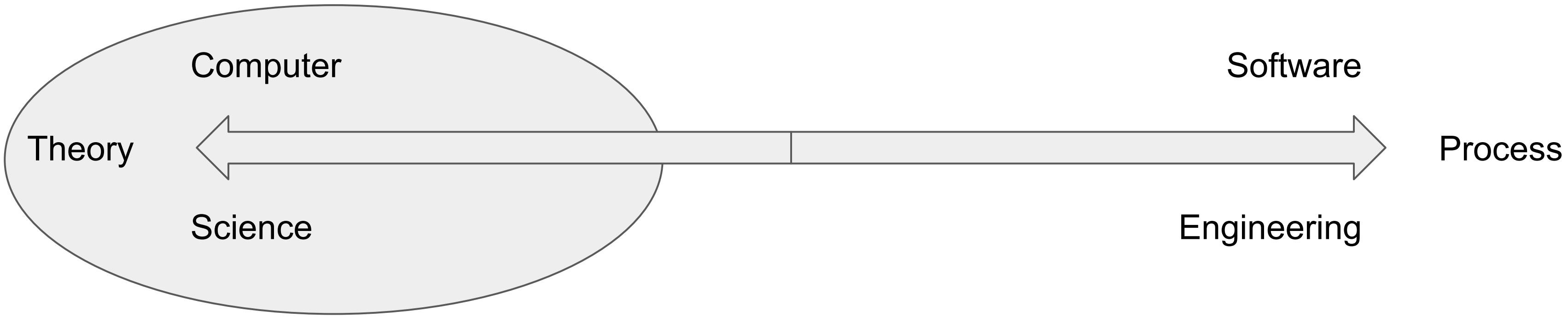
- Step-by-Step instructions

- Infinite number of possible algorithms, but a few universal ones.
    - The behavioral operations of the data structure
        - Searching / Sorting
        - Insert / Retrieve
        - Add / Remove
        - Compare / Compute

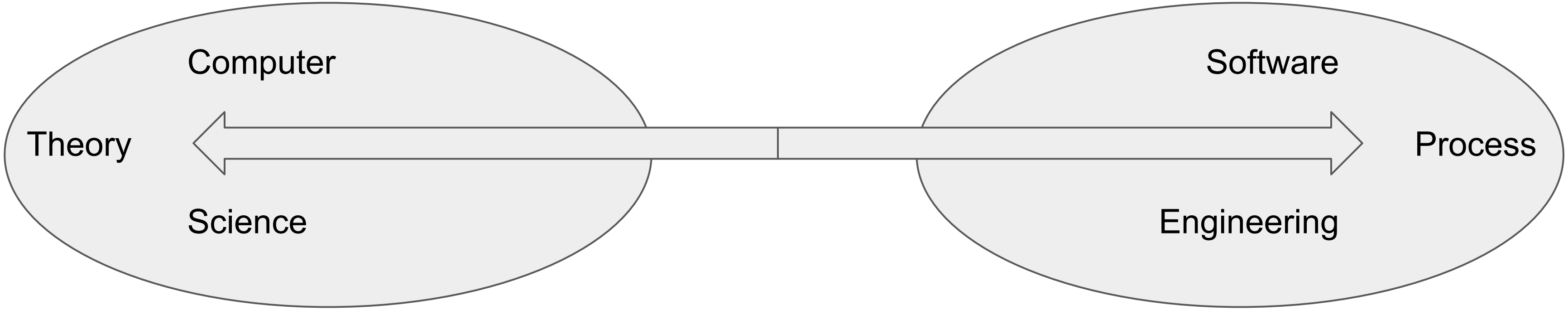
Data Structures & Algorithms



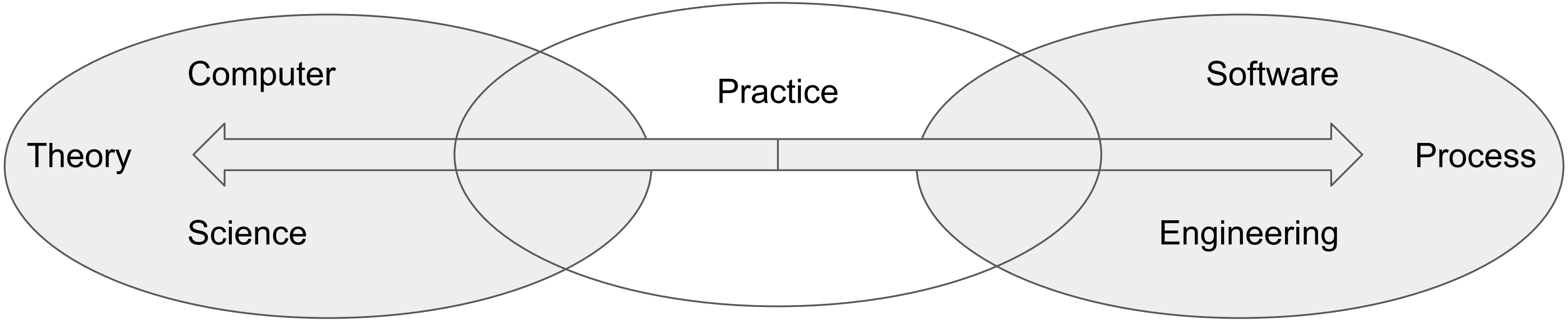
Data Structures & Algorithms



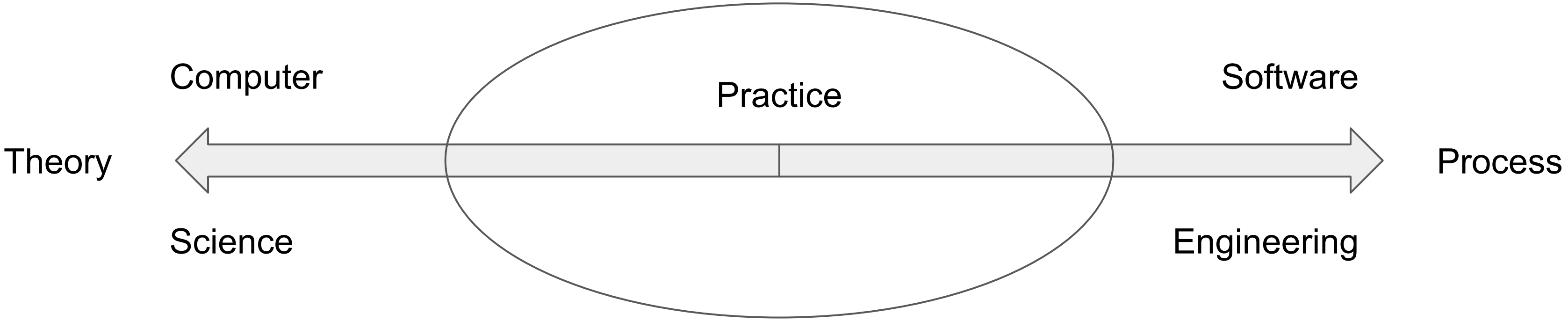
Data Structures & Algorithms



Data Structures & Algorithms



Data Structures & Algorithms



Comparison

- Many ways to perform the same operation
    - Bubble Sort, Insertion Sort, Quick Sort, Radix Sort, Etc.
    - Linear Search, Binary Search, etc.
- How can we know which algorithm is `better` than the other?
    - Analysis
        - Big-Oh

Comparison

- A universal statement of fact: the code we ourselves write is BETTER than the code others write
  - Prove me wrong?

- Big-Oh gives us an objective way to compare the performance between algorithms
  - And are often accompanied by a mathematical proof to prove its advertised performance

Comparison

- It is possible to make an accurate decision about its runtime performance just by looking at a few basic constructs of the code
    - No mathematical proof needed
        - Incredible skill to have during an interview
            - A few weeks more to become very comfy with edge cases

- Using the traditional method, years, usually (I don’t have proof for that statement)

Comparison

- The process of studying an algorithm to determine the Big-Oh category is called: Analysis
    - More specifically: Asymptotic Analysis (google it for more information)

- The result of the analysis indicates which Big-Oh function an algorithm belongs to

    - Though in written and spoken language, is rarely referred to as such
        - We’ll just say: O(n), or O(1), etc. or in English: Linear, Constant, etc.

Comparison

- What we measure:
  - Number of computations as the input size increases
  - How much memory consumption grows as the input size increases (sometimes)

- Performance

Worst case analysis: Big-Oh	$O(1)$, $O(n)$, etc.
Average or exact case analysis: Big-Theta	$\Theta(1)$, $\Theta(n)$, etc.
Best case analysis: Big-Omega	$\Omega(1)$, $\Omega(n)$, etc.

- Big-Oh very useful for comparing which algorithms perform the best
- Big-Theta very useful for actually comparing the average expected performance
- Big-Omega useful for comparing the best performance, not useful otherwise

Categories / Analysis

- Algorithms can belong to any performance category, but there are a few extremely common ones

O(1):	Constant	Best
O(log n):	Logarithmic	
O(n):	Linear	
O(n log n):	Linear-Logarithmic	
$O(n^2)$:	Quadratic	
$O(n^3)$:	Cubic	
$O(m^n)$:	Exponential	Worst

Categories / Analysis

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ to the sorted sequence $A[1..j-1]$	0	$n-1$
$\quad i = j - 1$	c_4	$n-1$
$\quad$ **while** $i > 0$ **and** $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\quad\quad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$\quad\quad i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$

$= O(n^2)$

Analysis

- O(1) : Constant ; fixed no. of operations
    - The amount of time does not change as the input size grows
    - The number of operations do not increase as the input size grows

```
function multiply(n, m) {
  return n * m;                                                // 1
}
```

Analysis

- O(n) : Linear ; loop, iteration, incremental recursion
  - The amount of time grows proportionately as the input size grows
  - The number of operations increase proportionately as the input size grows

```
function count(a) {
  var sum = 0;                                          // 1
  for (var n=0; n<a.length; n++) {                      // n
    sum += a[n];                                        // 1
  }
  return sum;                                           // 1
}
```

Analysis

- O(n) : Linear ; loop, iteration, incremental recursion
  - The amount of time grows proportionately as the input size grows
  - The number of operations increase proportionately as the input size grows

```
function compute(n) {
  var sum = 0;                                          // 1
  for (var i=0; i<n; i++) {                             // n
    sum += 2                                            // 1
  }
  for (var i=0; i<n; i++) {                             // n
    sum += 1;                                           // 1
  }
  return sum;                                           // 1
}
```

Analysis

- $O(n^2)$: Quadratic ; nested loops, inner incremental recursion
  - The amount of time grows as product of the input size
  - The number of operations increase as a product of the input size

```
function product(n) {
  var result = 0;                                          // 1
  for (var i=0; i<n; i++) {                                // n
    for (var j=0; j<n; j++) {                              // . n
      result += 1;                                         // 1
    }
  }
  return result;                                           // 1
}
```

Analysis

- $O(n^2)$: Quadratic ; nested loops, inner incremental recursion
    - The amount of time grows as product of the input size
    - The number of operations increase as a product of the input size

```
function compute(n) {
  var sum = 0;                                          // 1
  for (var i=0; i<n; i++) {                             // n
    sum += 2                                            // 1
  }
  for (var i=0; i<n; i++) {                             // n
    for (var j=0; j<n; j++) {                           // . n
      sum += 1;                                         // 1
    }
  }
  return sum;                                           // 1
}
```

Analysis

- O(log n) : Logarithmic ; Cuts the problem size by a fraction (usually ½)
  - The amount of time grows as a fraction of the input size
  - The number of operations increase as a fraction of the input size

```
function compute(n) {
  var result = 0;                                             // 1
  for (var i=0; i<n; i*=2) {                                  // n * ½
    sum += i;                                                 // 1
  }
  return sum;                                                 // 1
}
```

Analysis

- O(log n) : Logarithmic ; Cuts the problem size by a fraction (usually ½)
  - The amount of time grows as a fraction of the input size
  - The number of operations increase as a fraction of the input size

```
function compute(n) {
  var result = 0;                                   // 1
  for (var i=n; i>0; i/=2) {                        // n * ½
    sum += i;                                       // 1
  }
  return sum;                                       // 1
}
```

Analysis

- $O(n^3)$: Cubic ; triple nested loops, inner incremental recursion
  - The amount of time grows cubic in relation to the input size
  - The number of operations grow cubic in relation to the input size

```
function compute(list) {
  var count = 0;                                              // 1
  for(var i = 0; i < list.length; i++) {                      // n
    for(var j = i+1; j < list.length; j++) {                  // . n
      for(var k = j+1; k < list.length; k++) {                // . . n
        if(list[i] + list[j] + list[k] === 0) {               // 1
          Count++;                                            // 1
        }
      }
    }
  }
  return count;                                               // 1
};
```

Analysis

- $O(m^n)$: Exponential ; Too many nested computations
  - The amount of time grows exponentially as the input size increases
  - The number of operations grow exponentially as the input size increases

```
function compute(n) {                              // = O(n5)
  var sum = 0;                                     // 1
  for (var i=0; i<n; i++) {                        // n
    for (var j=i; j<i*i; j++) {                    // . n*n
      if (j % i === 0) {                           // . .   n
        for (var k=0; k<j; k++) {                  // . .   . n
          sum += 1;                                // 1
        }
      }
    }
  }
  return sum;                                      // 1
}
```

Analysis

- if-then-else statements
    - Whichever of the if-then-else parts is the biggest

```
function compute(a) {
  var sum = 0;
  if(a.length === 0) {
    return 0;
  }
  else {
    for (var n=0; n<a.length; n++) {
      if (a[n] % 2 == 0) {
        sum += 1;
      }
    }
    return sum;
  }
}
```

Analysis

- if-then-else statements
  - Whichever of the if-then-else parts is the biggest

```
function compute(a) {
  var sum = 0;                                          // 1
  if(a.length === 0) {                                  // 1
    return 0;                                           // 1
  }
  else {
    for (var n=0; n<a.length; n++) {                    // n
      if (a[n] % 2 == 0) {                              // 1
        sum += 1;                                       // 1
      }
    }
    return sum;                                         // 1
  }
}
```

Analysis

- Multiples / Repeats
    - Drop the constants, thus O(3n) becomes O(n); O(½n) becomes O(n); O(7) becomes O(1)

```
function compute(n) {
  var sum = 0;
  for (var i=0; i<n; i++) {
    sum += 2;
  }
  for (var i=0; i<n; i++) {
    sum += 1;
  }
  return sum;
}
```

Analysis

- Multiples / Repeats
  - Drop the constants, thus O(3n) becomes O(n); O(½n) becomes O(n); O(7) becomes O(1)

```
function compute(n) {
  var sum = 0;                                               // 1
  for (var i=0; i<n; i++) {                                  // n
    sum += 2;                                                // 1
  }
  for (var i=0; i<n; i++) {                                  // n
    sum += 1;                                                // 1
  }
  return sum;                                                // 1
}

                                    // Looks like O(2n)
                                    // Drop the `2`, is O(n)
```

Analysis

- Bonus
  - What is the Big-Oh of the following example?

```
function compute(n) {
  var sum = 0;
  for (var i=0; i<n; i++) {
    for (var k=n-i; k<i; k++) {
      sum += 1;
    }
  }
  return sum;
}
```

Analysis

- Bonus
  - What is the Big-Oh of the following example?

```
function compute(n) {
  var sum = 0;                                                    // 1
  for (var i=0; i<n; i++) {                                       // n
    for (var k=n-i; k<i; k++) {                                   // . n
      sum += 1;                                                   // 1
    }
  }
  return sum;                                                     // 1
}

// Looks like O(n * ½n), thus is O(n * n), or O(n^2)
```

Study Tools (bigocheatsheet.com)

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Implementation Review

Array

Write code to calculate the average of the following array:

```
var array = [4, 8, 12, 4, 2, 9, 1, 0, 12, 17, 8, 10];
```

Implementation Review

Array

Write code to calculate the average of the following array:

```
var array = [4, 8, 12, 4, 2, 9, 1, 0, 12, 17, 8, 10];
function average(a) {
  var average = 0;
  for (var i=0; i<a.length; i++) {                              // n
    average += a[i];
  }
  return average / a.length;
}
```

Implementation Review

LinkedList get(index)

```
// this._length;
// this._head
//
function get(index) {
  if (index > -1 && index < this._length) {
    var current = this._head,
    i = 0;
    while(i++ < index){
      current = current.next;
    }
    return current.data;
  } else {
    return null;
  }
}
```

Implementation Review

Array get(index)

```
function getValue(index, data) {
  return data[index];
}
```

Implementation Review

LinkedList remove(index)

```
function remove(index) {
  var i = 0;
  var current = first, previous;

  if(index === 0) {
    first = current.next;
  }
  else {
    while(i++ < index) {
      previous = current;
      current = current.next
    }

    previous.next = current.next;
  }
  return current.value;
};
```

Implementation Review

LinkedList append(value)

```
function append(data) {
  const node = {
    data: data,
    next: null
  };

  if(this.count === 0) {
    this.head = node;
  } else {
    this.tail.next = node;
  }

  this.tail = node;
  this.count++;
}
```

Implementation Review

Stack pop()

```
Stack.prototype.pop = function() {
    var size = this._size,
        deletedData;

    if (size) {
        deletedData = this._storage[size];

        delete this._storage[size];
        this._size--;

        return deletedData;
    }
};
```

Implementation Review

BinarySearchTree insert(...)

```
BinarySearchTree.prototype.insert = function (value) {
  var node = BinarySearchTree(value);

  function recurse(bst) {
    if (bst.value > value && bst.left === undefined) {
      bst.left = node;
    } else if (bst.value > value) {
      recurse(bst.left);
    } else if (bst.value < value && bst.right === undefined) {
      bst.right = node;
    } else if (bst.value < value) {
      recurse(bst.right);
    }
  }

  recurse(this);
}
```

Questions?

US

http://algoacad.me

linkedIn: /company/AlgorithmAcademy

Me

linkedIn: /bullockshawn