Unit 3: Algorithms

•••

Time Complexity & Big O Notation

Objectives

In this lecture, we'll look at how to assess time complexity in algorithms, in a general way, using Big O notation.

What is an algorithm?

- An **algorithm** is a step by step set of instructions that provide a solution to a problem.
- Most of the **business logic** in an app is algorithmic.
- **Business Logic:** determines how data can be created, stored, and changed
  - Image Processing
  - Recommendations for users to connect to (LinkedIn, Facebook)
  - File compression
  - Targeted advertising

What is time complexity?

- As developers, we need an easy way to **communicate** about the effectiveness of our algorithms.
- The more computational steps an algorithm performs, the longer it takes to complete.
- **Big O notation** describes the relationship between the size of an algorithm's input and the number of computational steps it takes for the algorithm to complete.
  - O(n) means proportional to input length
  - O(n^2) means proportional to input length squared

Interpretation of Big O notation

- **O(1)** - (**constant**) time complexity
    - Example: Testing to see if a key is in an object

- **O(n)** - (**linear**) time complexity
    - Example: Linear search

- **O(n^2)** - (**quadratic**) time complexity
    - Example: Two-sum (naive)

O(1) - Constant time

```
const objectLookup = (obj, target) => {
  if (obj[target]) return true;
  return false;
}
```

O(n) - Linear time

```
const linearSearch = (num, array) => {
  for (let i = 0; i < array.length; i++) {
    if (array[i] === num) return true;
  }
  return false;
}
```

$O(n^2)$ - Quadratic

```
const twoSum = function (nums, target) {
  for (let i = 0; i < nums.length; i += 1) {
    for (let j = i + 1; j < nums.length; j += 1) {
      if (nums[i] + nums[j] === target) return true;
    }
  }

  return false;
};
```

Time Complexity: Different Approaches

- Sometimes different approaches to the same problem can have different complexities. One approach may be more efficient than the other!
- Example: twoSum & sumNaturals problems

Big O (Further Breakdown)

Spoken	Written	Scenario	Speed
Constant	O(1)	Key lookup	Fastest
Logarithmic	O(log n)	Binary (tree) search	
Linear	O(n)	1 level looping	
Quasilinear	O(n log (n))	Good sorting (e.g. merge sort)	
Quadratic	$O(n^2)$	2 nested loops (e.g. selection sort)	
Exponential	$O(2^n)$	Password guessing	
Factorial	O(n!)	Generating permutations	Slowest

- n represents the size of the input. For functions of 1D arrays, n is the length of the array.
- Big O notation is always based on worst case.

Big O (Further Breakdown)

Big-O	Operations for 10 “things”	Operations for 100 “things”
O(1)	1	1
O(log n)	3	7
O(n)	10	100
O(n log n)	30	700
O(n^2)	100	10000
O(2^n)	1024	2^100
O(n!)	3628800	100!

logarithm

O(log n)

(B) Logarithms for the base 2 and corresponding powers

Log	Number
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	131072
18	262144
19	524288
20	1048576

Demo: Binary Search

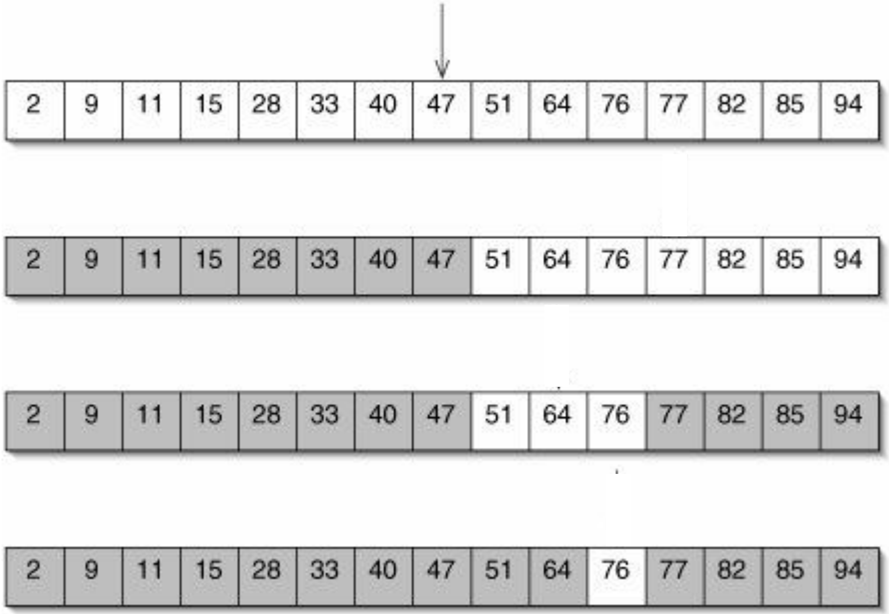
Binary Search

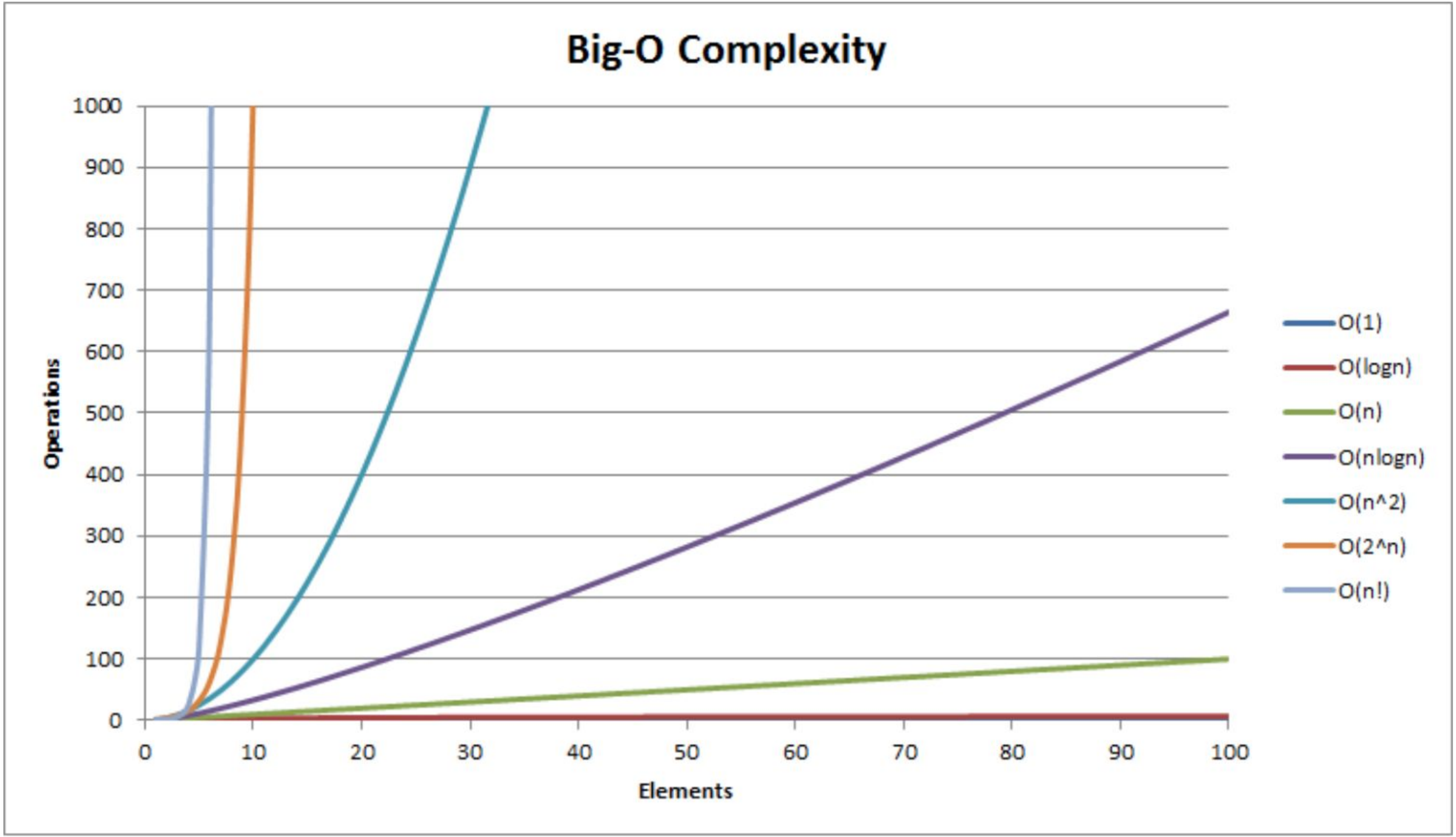
Given a target number, determine if such number exists in an array…

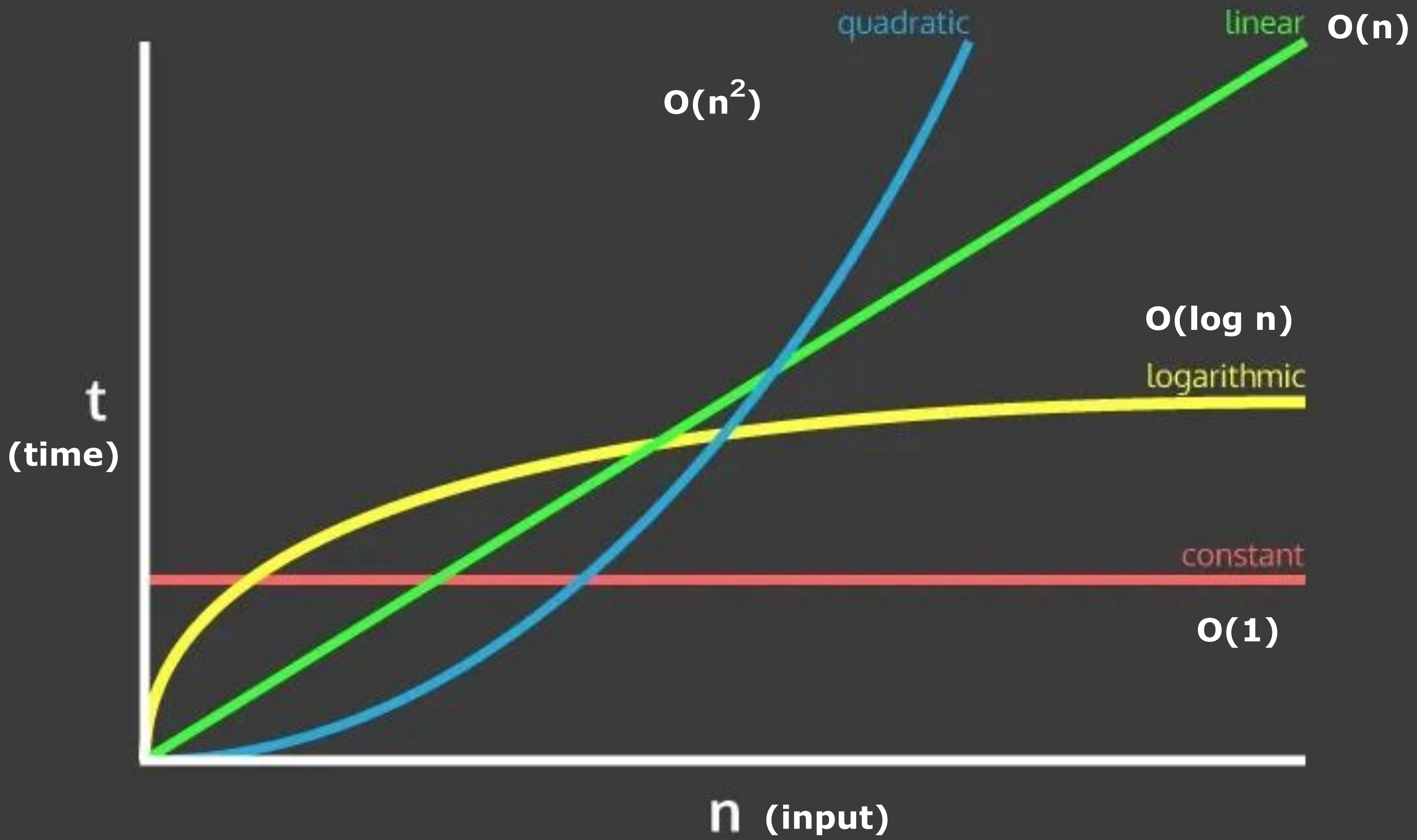
2	9	11	15	28	33	40	47	51	64	76	77	82	85	94

Binary Search

O(log n)




Big-O Complexity
Operations
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)


quadratic
linear
O(n)
$O(n^2)$
O(log n)
logarithmic
t
(time)
constant
O(1)
n (input)

Suppose I have an algorithm with $O(20n^2 + 3n + 10)$ time complexity. Usually, we just call it $O(n^2)$.

Why is this okay?

Big O Generalizes

Difference in time complexity matters only at **big numbers for n**.

Compare **$20n^2$** *to* **3n** *to* **10** when n = 1,000

10	O(1)
3 * 1,000 => 3,000	O(n)
20 * (1,000^2) => 20,000,000	$O(n^2)$

(6600 times bigger)

Constants *and* ***coefficients*** *are dropped to create an approximation.*

Using built-in methods

- While built-in methods make our code more succinct and readable, they often do a lot more under the hood!

- Example: under the hood, the .includes( ) method for arrays will iterate through an array and check if any elements match with the argument

Space Complexity

```
const addNums = (arr) => {
  let sum = 0;
  for (let i = 0; i < arr.length; i += 1) {
    sum += arr[i];
  }
  return sum;
};
```

```
const addOneToEach = (arr) => {
  const output = [];
  for (let i = 0; i < arr.length; i += 1) {
    const newSum = arr[i] + 1;
    output.push(newSum);
  }
  return output;
};
```

Considering Trade-offs when Optimizing

Often when optimizing an algorithm, you will face a tradeoff

- Time ↓, Space ↑ VS. Time ↑, Space ↓

Which should we go with?

- You can buy more memory (servers, RAM), but you **cannot** buy more time
- Time ↓, Space ↑

Why care?

Why does it matter?

Computers have limited resources (space and processing power). Writing algorithms with better time complexity saves time!

Time = Performance, efficiency = money!

Don’t Guess!

- Don’t memorize “code shape”
  - Nested for loops do NOT automatically mean $O(n^2)$
  - Single loops do NOT automatically mean $O(n)$
- If you understand time complexity behavior, you will understand how to approach algorithms quicker
- Focus on understanding WHY an algorithm has a certain time complexity

Summary

- An algorithm is a set of instructions that provides an answer to a problem.
- Time-complexity describes the rate at which the number of computations grows as the input grows.
- Big O Notation provides a way to represent time-complexity in a meaningful, but approximate way.
- Big O notation describes the **worst case scenario**.

Further Reading

Time Complexity (Wikipedia)
Big O Notation (Interview Cake)
Big O Notation (Wikipedia)
Big O Cheat Sheet