CS12: Introduction to Big O Notation

Understanding Algorithm Efficiency

Mr. Gullo

Key Terms for This Lesson

Vocabulary

Algorithm: A step-by-step set of instructions to solve a problem

Key Terms for This Lesson

Vocabulary

Algorithm: A step-by-step set of instructions to solve a problem

Efficiency: How fast and how little memory an algorithm uses

Key Terms for This Lesson

Vocabulary

Algorithm: A step-by-step set of instructions to solve a problem

Efficiency: How fast and how little memory an algorithm uses

Input size (n): The amount of data the algorithm works with

Key Terms for This Lesson

Vocabulary

Algorithm: A step-by-step set of instructions to solve a problem

Efficiency: How fast and how little memory an algorithm uses

Input size (n): The amount of data the algorithm works with

Complexity: How the time or space grows as n gets bigger

Learning Objectives

By the end of this lesson, you will be able to:

- Define algorithm efficiency in your own words

Learning Objectives

By the end of this lesson, you will be able to:

- Define algorithm efficiency in your own words
- Identify and explain common Big O notations

Learning Objectives

By the end of this lesson, you will be able to:

- Define algorithm efficiency in your own words
- Identify and explain common Big O notations
- Analyze simple algorithms to determine their time complexity

Learning Objectives

By the end of this lesson, you will be able to:

- Define algorithm efficiency in your own words
- Identify and explain common Big O notations
- Analyze simple algorithms to determine their time complexity
- Compare different algorithms based on their efficiency

What is Big O Notation?

Definition

Big O notation measures how long an algorithm takes or how much memory it uses as the input size grows.

What is Big O Notation?

Definition

Big O notation measures how long an algorithm takes or how much memory it uses as the input size grows.

- Think of it as a way to measure an algorithm's speed

What is Big O Notation?

Definition

Big O notation measures how long an algorithm takes or how much memory it uses as the input size grows.

- Think of it as a way to measure an algorithm's speed
- Helps us compare different solutions

What is Big O Notation?

Definition

Big O notation measures how long an algorithm takes or how much memory it uses as the input size grows.

- Think of it as a way to measure an algorithm's speed
- Helps us compare different solutions
- Focuses on the slowest possible situation

What is Big O Notation?

Definition

Big O notation measures how long an algorithm takes or how much memory it uses as the input size grows.

- Think of it as a way to measure an algorithm's speed
- Helps us compare different solutions
- Focuses on the slowest possible situation
- Ignores smaller details and focuses on the main pattern

What is Worst Case?

Why do we care about worst case?

Big O describes the **maximum** time an algorithm could take.

What is Worst Case?

Why do we care about worst case?

Big O describes the **maximum** time an algorithm could take.

Example: Searching for 99 in this array:

12	5	23	7	42	15	8	31	99
0	1	2	3	4	5	6	7	8

What is Worst Case?

Why do we care about worst case?

Big O describes the **maximum** time an algorithm could take.

Example: Searching for 99 in this array:

0	1	2	3	4	5	6	7	8
12	5	23	7	42	15	8	31	99

- Linear search checks: 12, 5, 23, 7, 42, 15, 8, 31, **99**

What is Worst Case?

Why do we care about worst case?

Big O describes the **maximum** time an algorithm could take.

Example: Searching for 99 in this array:

0	1	2	3	4	5	6	7	8
12	5	23	7	42	15	8	31	99

- Linear search checks: 12, 5, 23, 7, 42, 15, 8, 31, **99**
- **Worst case**: Target is at the END (or not there at all)

What is Worst Case?

Why do we care about worst case?

Big O describes the **maximum** time an algorithm could take.

Example: Searching for 99 in this array:

0	1	2	3	4	5	6	7	8
12	5	23	7	42	15	8	31	99

- Linear search checks: 12, 5, 23, 7, 42, 15, 8, 31, **99**
- **Worst case**: Target is at the END (or not there at all)
- We had to check **all 9 elements** = O(n) operations

Best Case vs Worst Case

12	5	23	7	42	15	8	31	99

Best Case: O(1)

- Target = 12 (first element)
- Found in 1 check!

Best Case vs Worst Case

12	5	23	7	42	15	8	31	99

Best Case: O(1)

- Target = 12 (first element)
- Found in 1 check!

Worst Case: O(n)

- Target = 99 (last element)
- Takes 9 checks
- Or target not in array!

Best Case vs Worst Case

12	5	23	7	42	15	8	31	99

Best Case: O(1)

- Target = 12 (first element)
- Found in 1 check!

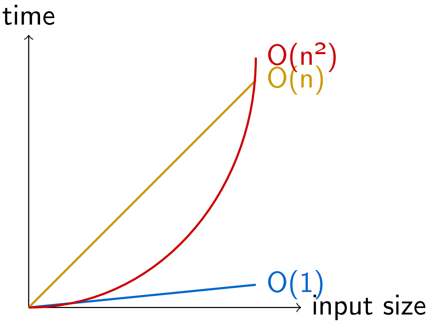
Worst Case: O(n)

- Target = 99 (last element)
- Takes 9 checks
- Or target not in array!

Why Worst Case?

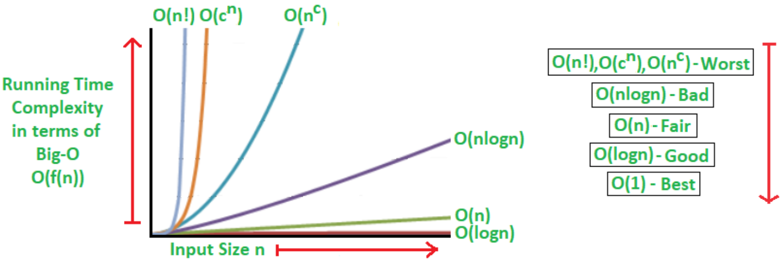
We plan for the worst so our program never surprises us with slow performance.

How Different Algorithms Grow

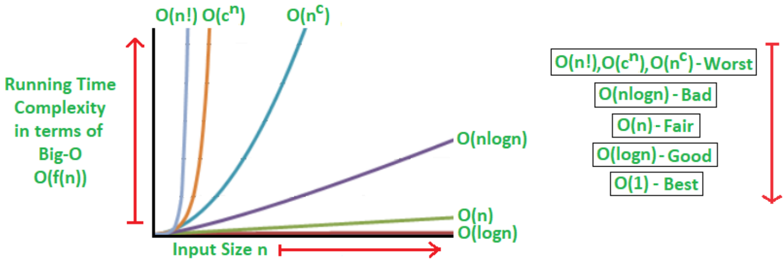


As input grows...

Big O Complexity Comparison



Big O Complexity Comparison



Key insight: As input grows, the gap between O(1) and O(n!) becomes enormous.

O(1) - Constant Time

- Finding a book on your desk
- Looking up array element by index

Understanding Through Real Examples

O(1) - **Constant Time**

- Finding a book on your desk
- Looking up array element by index

O(n) - **Linear Time**

- Searching through a line of books
- Finding maximum in unsorted array

Understanding Through Real Examples

O(1) - Constant Time

- Finding a book on your desk
- Looking up array element by index

O(n) - Linear Time

- Searching through a line of books
- Finding maximum in unsorted array

O(n^2) - Quadratic Time

- Everyone shakes hands with everyone
- Bubble sort algorithm

Understanding Through Real Examples

O(1) - Constant Time

- Finding a book on your desk
- Looking up array element by index

O(n) - Linear Time

- Searching through a line of books
- Finding maximum in unsorted array

O(n^2) - Quadratic Time

- Everyone shakes hands with everyone
- Bubble sort algorithm

O(log n) - Logarithmic Time

- Guessing a number 1-100 by halving
- Binary search (sorted data only!)

Linear Search: Check Every Element

Searching for 42:

Found!

12	5	23	7	42	15	8	31	99

Linear Search: Check Every Element

Searching for 42:

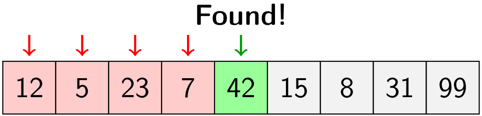
Found!

12	5	23	7	42	15	8	31	99

- Check index 0, 1, 2, 3... until we find 42

Linear Search: Check Every Element

Searching for 42:



- Check index 0, 1, 2, 3... until we find 42
- **5 checks** to find element at index 4

Linear Search: Check Every Element

Searching for 42:

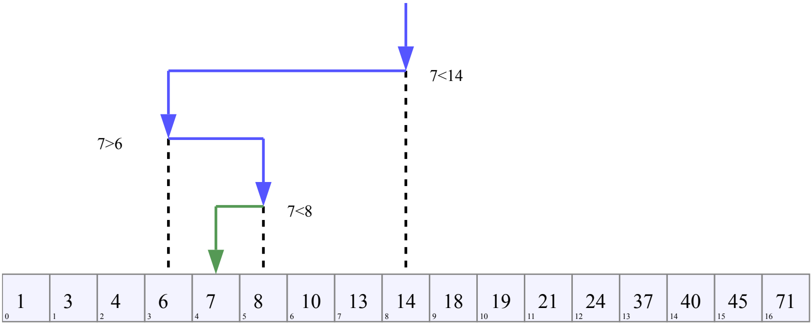
Found!

12	5	23	7	42	15	8	31	99

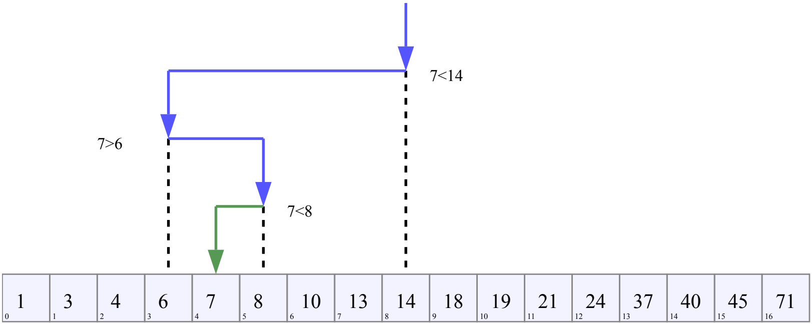
- Check index 0, 1, 2, 3... until we find 42
- **5 checks** to find element at index 4

O(n): In worst case, we check every single element.

Binary Search: Divide and Conquer



Binary Search: Divide and Conquer



O(log n): Each step eliminates **half** the remaining elements.

Requirement

Binary search only works on **sorted** data!

Image: Wikimedia Commons

What is a Logarithm?

Simple Definition

$\log_2(n) =$ "How many times can we divide n by 2?"

What is a Logarithm?

Simple Definition

$\log_2(n) =$ "How many times can we divide n by 2?"

$$\boxed{\text{8 items}} \xrightarrow[\nabla \cdot 2]{} \boxed{4} \xrightarrow[\nabla \cdot 2]{} \boxed{2} \xrightarrow[\nabla \cdot 2]{} \boxed{1}$$

3 divisions $\rightarrow \log_2(8) = 3$

What is a Logarithm?

Simple Definition

$\log_2(n) =$ "How many times can we divide n by 2?"

$$\boxed{\text{8 items}} \xrightarrow[\div 2]{} \boxed{4} \xrightarrow[\div 2]{} \boxed{2} \xrightarrow[\div 2]{} \boxed{1}$$

3 divisions $\rightarrow \log_2(8) = 3$

The Pattern

$$\begin{aligned} 2^3 &= 8 &\Leftrightarrow\quad& \log_2(8) = 3 \\ 2^4 &= 16 &\Leftrightarrow\quad& \log_2(16) = 4 \\ 2^{10} &= 1024 &\Leftrightarrow\quad& \log_2(1024) = 10 \end{aligned}$$

Why Logarithms are Amazing

n (items)	O(n) checks	O(log n) checks
8	8	3
16	16	4
256	256	8
1,024	1,024	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

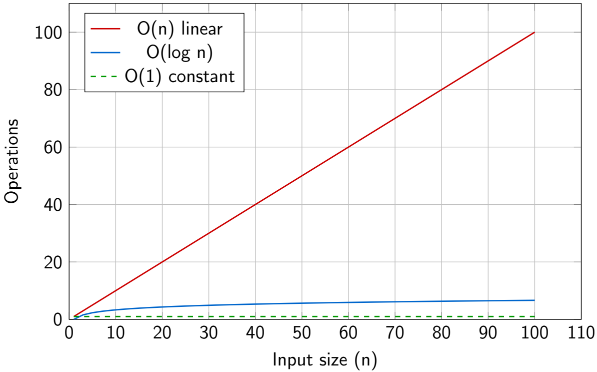
Why Logarithms are Amazing

n (items)	O(n) checks	O(log n) checks
8	8	3
16	16	4
256	256	8
1,024	1,024	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

Mind-Blowing

1 billion items with binary search = only **30 checks!**

Graph: O(log n) vs O(n)



Binary Search: Halving in Action

Find 44 in sorted array of 16 elements:

Step 1:	3	7	12	19	23	31	38	42	44	51	56	62	71	78	85	99	check 42, go RIGHT

Binary Search: Halving in Action

Find 44 in sorted array of 16 elements:

Step 1:	3	7	12	19	23	31	38	42	44	51	56	62	71	78	85	99	check 42, go RIGHT
Step 2:	44	51	56	62	71	78	85	99									check 56, go LEFT

Binary Search: Halving in Action

Find 44 in sorted array of 16 elements:

Step 1:

3	7	12	19	23	31	38	42	44	51	56	62	71	78	85	99

check 42, go RIGHT

Step 2:

44	51	56	62	71	78	85	99

check 56, go LEFT

Step 3:

44	51	56	62

check 51, go LEFT

Binary Search: Halving in Action

Find 44 in sorted array of 16 elements:

Step 1:	3 7 12 19 23 31 38 42 44 51 56 62 71 78 85 99	check 42, go RIGHT
Step 2:	44 51 56 62 71 78 85 99	check 56, go LEFT
Step 3:	44 51 56 62	check 51, go LEFT
Step 4:	44	FOUND! Only 4 checks for 16 items

Binary Search: Halving in Action

Find 44 in sorted array of 16 elements:

Step 1: 3 | 7 | 12 | 19 | 23 | 31 | 38 | 42 | 44 | 51 | 56 | 62 | 71 | 78 | 85 | 99 — check 42, go RIGHT

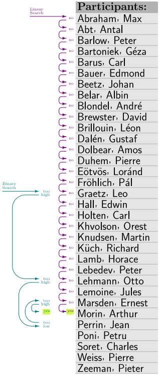
Step 2: 44 | 51 | 56 | 62 | 71 | 78 | 85 | 99 — check 56, go LEFT

Step 3: 44 | 51 | 56 | 62 — check 51, go LEFT

Step 4: 44 — FOUND! Only 4 checks for 16 items

$\log_2(16) = 4$ checks vs Linear: up to 16 checks

Linear vs Binary Search



Linear Search

- Works on any data
- O(n) time

Image: Wikimedia Commons

Binary Search

- Requires sorted data
- O(log n) time

I Do: Analyzing Linear Search

Problem

Let's analyze this linear search algorithm:

I Do: Analyzing Linear Search

Problem

Let's analyze this linear search algorithm:

```
int linearSearch(int arr[], int n, int x) {
    for(int i = 0; i < n; i++) {
        if(arr[i] == x) {
            return i;  // Found it!
        }
    }
    return -1;  // Not found
}
```

I Do: Analyzing Linear Search

Problem

Let's analyze this linear search algorithm:

```
int linearSearch(int arr[], int n, int x) {
    for(int i = 0; i < n; i++) {
        if(arr[i] == x) {
            return i;  // Found it!
        }
    }
    return -1;  // Not found
}
```

- Time Complexity: O(n)

I Do: Analyzing Linear Search

Problem

Let's analyze this linear search algorithm:

```
int linearSearch(int arr[], int n, int x) {
    for(int i = 0; i < n; i++) {
        if(arr[i] == x) {
            return i;  // Found it!
        }
    }
    return -1;  // Not found
}
```

- Time Complexity: O(n)
- Why? In worst case, we check every element

We Do: Let's Analyze Together

What's the time complexity?

What's the time complexity?

```
void printPairs(int arr[], int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            cout << arr[i] << ","
                 << arr[j] << endl;
        }
    }
}
```

We Do: Let's Analyze Together

What's the time complexity?

```
void printPairs(int arr[], int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            cout << arr[i] << ","
                 << arr[j] << endl;
        }
    }
}
```

- Let's count the operations...

We Do: Let's Analyze Together

What's the time complexity?

```cpp
void printPairs(int arr[], int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            cout << arr[i] << ","
                 << arr[j] << endl;
        }
    }
}
```

- Let's count the operations...
- Outer loop runs n times

We Do: Let's Analyze Together

What's the time complexity?

```cpp
void printPairs(int arr[], int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            cout << arr[i] << ","
                 << arr[j] << endl;
        }
    }
}
```

- Let's count the operations...
- Outer loop runs n times
- For each outer loop, inner loop runs n times

We Do: Let's Analyze Together

What's the time complexity?

```
void printPairs(int arr[], int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            cout << arr[i] << ","
                 << arr[j] << endl;
        }
    }
}
```

- Let's count the operations...
- Outer loop runs n times
- For each outer loop, inner loop runs n times
- Total operations: n times n = n squared

You Do: Practice Time!

Analyze These Operations

Determine the Big O notation for:

You Do: Practice Time!

Analyze These Operations

Determine the Big O notation for:

1. Getting the first element of an array

You Do: Practice Time!

Analyze These Operations

Determine the Big O notation for:

1. Getting the first element of an array
2. Finding the maximum value in an unsorted array

You Do: Practice Time!

Analyze These Operations

Determine the Big O notation for:

1. Getting the first element of an array
2. Finding the maximum value in an unsorted array
3. Checking if a number is even or odd

You Do: Practice Time!

Analyze These Operations

Determine the Big O notation for:

1. Getting the first element of an array
2. Finding the maximum value in an unsorted array
3. Checking if a number is even or odd

Solutions

1. O(1) - Direct access, no matter the size

You Do: Practice Time!

Analyze These Operations

Determine the Big O notation for:

1. Getting the first element of an array
2. Finding the maximum value in an unsorted array
3. Checking if a number is even or odd

Solutions

1. O(1) - Direct access, no matter the size
2. O(n) - Must check every element once

You Do: Practice Time!

Analyze These Operations

Determine the Big O notation for:

1. Getting the first element of an array
2. Finding the maximum value in an unsorted array
3. Checking if a number is even or odd

Solutions

1. O(1) - Direct access, no matter the size
2. O(n) - Must check every element once
3. O(1) - Single operation, size independent

Big O Quick Reference

Notation	Name	If n = 1000
O(1)	Constant	1 operation
O(log n)	Logarithmic	$\sim$10 operations
O(n)	Linear	1,000 operations
O(n log n)	Linearithmic	$\sim$10,000 operations
O(n^2)	Quadratic	1,000,000 operations
O(2^n)	Exponential	More than atoms in universe!

Big O Quick Reference

Notation	**Name**	**If n = 1000**
O(1)	Constant	1 operation
O(log n)	Logarithmic	∼10 operations
O(n)	Linear	1,000 operations
O(n log n)	Linearithmic	∼10,000 operations
O(n^2)	Quadratic	1,000,000 operations
O(2^n)	Exponential	More than atoms in universe!

Rule of thumb: Anything slower than O(n^2) is usually too slow for large data.

Key Takeaways

Remember These Points

- Big O notation helps us measure efficiency

Key Takeaways

Remember These Points

- Big O notation helps us measure efficiency
- Most common notations: O(1), O(n), O(n squared)

Key Takeaways

Remember These Points

- Big O notation helps us measure efficiency
- Most common notations: O(1), O(n), O(n squared)
- Consider how performance changes with input size

Key Takeaways

Remember These Points

- Big O notation helps us measure efficiency
- Most common notations: O(1), O(n), O(n squared)
- Consider how performance changes with input size
- Different problems require different solutions

Key Takeaways

Remember These Points

- Big O notation helps us measure efficiency
- Most common notations: O(1), O(n), O(n squared)
- Consider how performance changes with input size
- Different problems require different solutions

Practice Makes Perfect

Try analyzing algorithms you write in your own code!