O(1) - CONSTANT TIME

The Lightning Fast Algorithm - Always the Same Speed!

CS 101 - Fall 2025

What is O(1) - Constant Time?

The Superpower of Algorithms

O(1) means the algorithm takes the **same amount of time** no matter how much data you give it!
Real-World Analogy: * Like a **valet parking service** - you hand over your ticket and get your car back instantly * Whether there are 10 cars or 10,000 cars in the lot, it takes the same time! * The valet has a **direct system** to find your exact car

Performance Guarantee

- 10 items → 1 step
- 100 items → 1 step

- 1,000,000 items → 1 step
- Same speed **forever**!

Key Insight

The algorithm has a **"shortcut"** that goes directly to the answer without checking other data!

Magic Question:

"Can I get the answer without looking at most of the data?"

What Makes O(1) So Fast?

The Secret Ingredients

O(1) algorithms use **smart data organization** and **direct access patterns**

Hash Tables (Dictionaries/Sets)

```python
# Python dictionary uses hash function
student_grades = {
    "Alice": 95,
    "Bob": 87,
    "Charlie": 92
}

# Hash function calculates EXACTLY where
# "Alice" is stored in memory
grade = student_grades["Alice"]  # O(1)!
```

How it works:

1. Hash function: `"Alice"` → memory location 147
2. Go directly to location 147
3. Get the value (95)
4. Done! No searching needed!

Array Indexing

```python
# Arrays store data in consecutive memory
scores = [95, 87, 92, 78, 85]
#         0   1   2   3   4

# Computer calculates:
# Address = start_address + (index × item_size)
first_score = scores[0]    # O(1)
third_score = scores[2]    # O(1)
```

Mathematical Magic:

- Memory address = Base + (2×4 bytes) = Direct jump!
- No need to check scores[0] or scores[1]
- Jump straight to the answer!

Interactive O(1) Dictionary Demo

> **ℹ See O(1) in Action!**
>
> Try looking up different students' grades. Notice how it's always instant, no matter which student you choose!

Python O(1) Examples - The Fast Ones!

Dictionary Operations

```python
# Creating a gradebook
gradebook = {
    "Alice": 95,
    "Bob": 87,
    "Charlie": 92,
    "Diana": 88
}

# All of these are O(1) - instant!
alice_grade = gradebook["Alice"]      # Lookup: O(1)
gradebook["Eve"] = 90                 # Insert: O(1)
gradebook["Bob"] = 89                 # Update: O(1)
del gradebook["Charlie"]              # Delete: O(1)

# Check if student exists
if "Diana" in gradebook:              # Membership: O(1)
    print("Diana is in the class!")

print(f"Class size: {len(gradebook)}")  # Length: O(1)
```

List Operations (by index)

```python
# Student scores list
scores = [95, 87, 92, 78, 85, 91, 88]

# These are O(1) - direct access
first_score = scores[0]               # Get first: O(1)
last_score = scores[-1]               # Get last: O(1)
middle_score = scores[3]              # Get by index: O(1)
```

```python
print(f" Scores BEFORE update = {scores}")

scores[2] = 94                        # Update by index: O(1)
                                      # overwrite previous value
print(f" Scores AFTER update = {scores}")

# Stack operations (end of list)
scores.append(96)                     # Add to end: O(1)
final_score = scores.pop()            # Remove from end: O(1)

print(f"Total scores: {len(scores)}") # Length: O(1)
```

> ⚠ Important Note
>
> **Not all list operations are O(1)!**
>
> - `scores.insert(0, 100)` is O(n) - inserting at beginning
> - `scores.remove(87)` is O(n) - searching for value
> - `87 in scores` is O(n) - searching through list

Set Operations - Another O(1) Champion!

> ℹ Note
>
> Are there differences in search times?

Set Basics

```python
# Create a set of student IDs
enrolled_students = {101, 102, 103, 104, 105}

# All O(1) operations
enrolled_students.add(106)            # Add: O(1)
enrolled_students.remove(103)         # Remove: O(1)

# The magic of O(1) membership testing!
if 104 in enrolled_students:          # Check: O(1)
    print("Student 104 is enrolled!")

# Compare with list - this would be O(n)
```

```python
student_list = [101, 102, 103, 104, 105]
if 104 in student_list:               # O(n) - slow!
    print("Found in list (but slowly)")
```

Real Performance Difference

```python
import time

# Large dataset
large_list = list(range(100000))        # 100k numbers
large_set = set(range(100000))          # Same 100k numbers

target = 99999  # Last item (worst case)

# Timing list search - O(n)
start = time.time()
found = target in large_list            # Checks all 100k!
list_time = time.time() - start

# Timing set search - O(1)
start = time.time()
found = target in large_set             # Instant!
set_time = time.time() - start + 0.001  # Add error time


print(f"List search: {list_time:.6f} seconds")
print(f"Set search: {set_time:.6f} seconds")
print(f"Set is {list_time/set_time:.0f}x faster!")
```

Partner Activity: O(1) Experiments!

> **Instructions**: Find a partner and complete these experiments together. Discuss your observations!

Experiment 1: Dictionary vs List Race

```python
import time
import random

# Create test data with your partner
```

```python
def create_test_data(size):
    data = [f"student_{i}" for i in range(size)]
    data_dict = {name: f"grade_{i}" for i, name in enumerate(data)}
    return data, data_dict

# Partner A: Test small dataset (100 items)
# Partner B: Test large dataset (10,000 items)
sizes = [100, 10000]  # Choose one each

for size in sizes:
    data_list, data_dict = create_test_data(size)
    target = f"student_{size-1}"  # Last item (worst case)

    # Time list search (O(n))
    start = time.time()
    found = target in data_list
    list_time = time.time() - start

    # Time dict search (O(1))
    start = time.time()
    found = target in data_dict
    dict_time = time.time() - start

    print(f"Size {size}:")
    print(f"  List search: {list_time:.6f} seconds")
    print(f"  Dict search: {dict_time:.6f} seconds")
    # zero division precaution: add 0.001 to denominator
    print(f"  Dict is {list_time/(dict_time+0.001):.0f}x faster!")
```

Discussion Questions:

- What happened to the performance difference as data size increased?
- Why doesn't dictionary search time change much?

Experiment 2: Hash Table Investigation

```python
# Build a student grade system together
student_grades = {}

# Partner A: Add students with IDs 1-1000
# Partner B: Add students with IDs 1001-2000
def add_students(start_id, end_id, grades_dict):
```

```
    import time
    start = time.time()

    for i in range(start_id, end_id + 1):
        grades_dict[f"student_{i}"] = random.randint(70, 100)

    end = time.time()
    return end - start

# Time your additions
partner_a_time = add_students(1, 1000, student_grades)
partner_b_time = add_students(1001, 2000, student_grades)

# Now test random lookups
def test_lookups(grades_dict, num_tests=100):
    import time
    start = time.time()

    for _ in range(num_tests):
        random_id = random.randint(1, 2000)
        grade = grades_dict.get(f"student_{random_id}", "Not found")

    return time.time() - start

lookup_time = test_lookups(student_grades)
print(f"Adding 1000 students: ~{partner_a_time:.4f} seconds each")
print(f"100 lookups in {len(student_grades)} students: {lookup_time:.6f} seconds")
```

Partner Discussion:

- Did adding more students slow down individual lookups?
- How would this compare with a simple list of 2000 students?

O(1) Scenario Analysis: Group Discussion Prompts

Class Discussion Questions

Work in groups of 3-4. Discuss these scenarios and be ready to share insights!

Important

Discuss with your group:

1. **Social Media Apps**: When you open Instagram/TikTok, your profile loads instantly regardless of how many users the app has. What O(1) operations make this possible?
2. **Online Shopping**: Amazon can instantly tell you if an item is in stock, even with millions of products. How might they achieve O(1) inventory checks?
3. **Video Games**: In a multiplayer game with thousands of players, you can instantly see your own stats. What data structure enables O(1) player data retrieval?
4. **School Systems**: Your student portal shows your GPA instantly, even though the school has thousands of students. What makes this O(1)?

Group Challenge: Design a simple system for one of these scenarios using Python dictionaries. How would you organize the data for O(1) access?

O(1) Scenario Analysis: Critical Thinking

Class Discussion Questions

Work in groups of 3-4. Discuss these scenarios and be ready to share insights!

Important

Discuss with your group:

5. **Trade-off Analysis**: O(1) operations often require extra memory (like hash tables). When is this trade-off worth it? When might it not be?
6. **Failure Cases**: Can you think of situations where a dictionary lookup might NOT be O(1)? (Hint: Think about hash collisions)
7. **Design Decisions**: You're building an app that needs to store user preferences. Would you use a list or dictionary? Why? How would your choice affect performance with 1 user vs 1 million users?
8. **Performance Prediction**: If a dictionary lookup takes 0.001 seconds with 1,000

items, how long should it take with 1,000,000 items? Why?

Your Turn: Practice O(1) Operations!

Individual Coding Exercises

After your group discussions, try these hands-on exercises to solidify your understanding.

Exercise 1: Phone Book

```
# Create your own phone book
phone_book = {}

# Add contacts (O(1) each)
phone_book["Mom"] = "555-0123"
phone_book["Pizza Place"] = "555-PIZZA"
phone_book["Best Friend"] = "555-9999"

# Look up numbers instantly
print(f"Mom's number: {phone_book['Mom']}")

# Your task: Add 5 more contacts and
# time how long it takes to look them up!

import time
start = time.time()
# Add your lookups here
end = time.time()
print(f"Lookup time: {end - start} seconds")
```

Exercise 2: Student Checker

```
# Create sets for different classes
math_class = {"Alice", "Bob", "Charlie"}
science_class = {"Bob", "Diana", "Eve"}
history_class = {"Alice", "Eve", "Frank"}

# Check enrollment instantly (O(1))
student = "Alice"
enrolled_classes = []
```

```python
# Check to see whether item is in dictionary
# add subject name to new list if student is enrolled

if student in math_class:
    enrolled_classes.append("Math")
if student in science_class:
    enrolled_classes.append("Science")
if student in history_class:
    enrolled_classes.append("History")

print(f"{student} is in: {enrolled_classes}")

# Your task: Check enrollment for all students
# and see how fast it is!
```

When NOT to Use O(1) Approaches

> ⚠ O(1) Has Limitations!
>
> While O(1) is amazing, it's not always possible or practical.

When O(1) Won't Work

```python
# These operations CANNOT be O(1):

scores = [95, 87, 92, 78, 85]

# Finding maximum - must check all values
max_score = max(scores)           # O(n) - no shortcut!

# Counting specific values
count_90s = scores.count(90)      # O(n) - must check all

# Finding an item in unsorted list
position = scores.index(92)       # O(n) - no direct path

# Sorting data
scores.sort()                     # O(n log n) - complex!
```

Why?

No way to know the answer without examining the data!

Trade-offs to Consider

```
# Memory vs Speed Trade-off
students = ["Alice", "Bob", "Charlie", "Diana"]

# Option 1: List (less memory, slower search)
if "Bob" in students:              # O(n) - slow search
    print("Found Bob!")

# Option 2: Set (more memory, faster search)
# Note: Sets contain only unique items.
# Although time and energy is required to complete
# operations involved with creating sets ...

student_set = set(students)        # Uses more memory
if "Bob" in student_set:           # O(1) - fast search!
    print("Found Bob!")

# Choice depends on:
# - How often do you search?
# - How much memory do you have?
# - How large is your data?
```

Comparing O(1) to Other Complexities

> See the Dramatic Difference!
>
> Let's compare O(1) to other complexities with real numbers.

Data Size	O(1)	O(log n)	O(n)	O(n^2)	Real-World Impact
10 items	1 step	~3 steps	10 steps	100 steps	All feel instant
100 items	1 step	~7 steps	100 steps	10,000 steps	O(1) still instant
1,000 items	1 step	~10 steps	1,000 steps	1,000,000 steps	Only O(1) stays fast
1,000,000 items	1 step	~20 steps	1,000,000 steps	1,000,000,000,000 steps	O(1) is superhuman!

The O(1) Advantage

Dictionary lookup with 1 million entries = Same speed as with 10 entries! This is why Google can search billions of web pages instantly - they use hash tables and other O(1) techniques!

Summary: O(1) - The Algorithm Superhero!

Key Takeaways

O(1) - Constant Time is the gold standard of algorithm efficiency!

What Makes O(1) Special

- **Always the same speed** - no matter how much data
- **Direct access patterns** - no searching required

- **Smart data structures** - hash tables, arrays
- **Real-world applications** - Google search, databases, caches

Python O(1) Champions:

- Dictionary operations: `dict[key]`, `dict[key] = value`
- Set operations: `item in set`, `set.add(item)`
- List indexing: `list[0]`, `list[-1]`
- Stack operations: `list.append()`, `list.pop()`

Programming Wisdom

```
# Choose your data structure wisely!

# For frequent lookups
use_dict = {"key": "value"}        # O(1) lookup
# not_list = ["key", "value"]      # O(n) lookup

# For membership testing
use_set = {1, 2, 3, 4, 5}         # O(1) testing
# not_list = [1, 2, 3, 4, 5]      # O(n) testing

# For indexed access
use_list = [1, 2, 3, 4, 5]        # O(1) by index
# Perfect for stacks and queues
```

Remember: O(1) is not always possible, but when it is, it's magical!

Next: Exploring O(log n) - The Smart Searcher!

Coming Up Next

O(log n) - Logarithmic Time * Binary search and divide-and-conquer strategies * Why "halving" is so powerful * Interactive binary search demonstrations * When O(log n) beats O(n) by huge margins
Questions to Think About: * When might you need to give up O(1) for O(log n)? * How can "smart searching" be almost as good as direct access? * What's the trade-off between data organization and search speed?

Ready to see how smart algorithms can be nearly as fast as O(1)?