Designing Code for Performance

Choosing and using data structures effectively

Session 224

Quinn Taylor
Internal Applications Engineer

Introduction

Introduction

- Huge influx of new developers

Introduction

- Huge influx of new developers
- Diverse backgrounds and experience

Introduction

- Huge influx of new developers
- Diverse backgrounds and experience
- Data structures performance issues are universal

Introduction

- Huge influx of new developers
- Diverse backgrounds and experience
- Data structures performance issues are universal
- Puzzling unless you know what's going on under the hood

“Give a man a fish and you feed him for a day. Teach a man how to fish and you feed him for a lifetime.”

Chinese Proverb

What You Will Learn

What You Will Learn

- When to focus on performance

What You Will Learn

- When to focus on performance
- How to evaluate computational complexity

What You Will Learn

- When to focus on performance
- How to evaluate computational complexity
- How to choose and use data structures

What You Will Learn

- When to focus on performance
- How to evaluate computational complexity
- How to choose and use data structures
- How to design your code for performance

When to Focus on Performance

“We don’t get a chance to do that many things, and every one should be really excellent. …[W]e’ve all chosen to do this with our lives. So it better be damn good. It better be worth it.”

Steve Jobs (2008)

“You have to pick carefully. Innovation is saying no to 1,000 things.”

Steve Jobs (1997)

Premature optimization is the root of all evil.

Donald Knuth (1974)

“We should forget about small efficiencies ... about 97% of the time. Premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”

Donald Knuth (1974)

“Optimize performance when it will make a meaningful difference.”

Me (just now)

Picking Your Battles

Amdahl's Law

Picking Your Battles

Amdahl's Law

- Maximum improvement from speeding up code
  - Depends on percentage of execution time
  - Payoff is bigger for dominant pieces

Picking Your Battles

Amdahl's Law

- Maximum improvement from speeding up code
  - Depends on percentage of execution time
  - Payoff is bigger for dominant pieces
- Will the payoff be worth the effort?

Picking Your Battles

Amdahl's Law

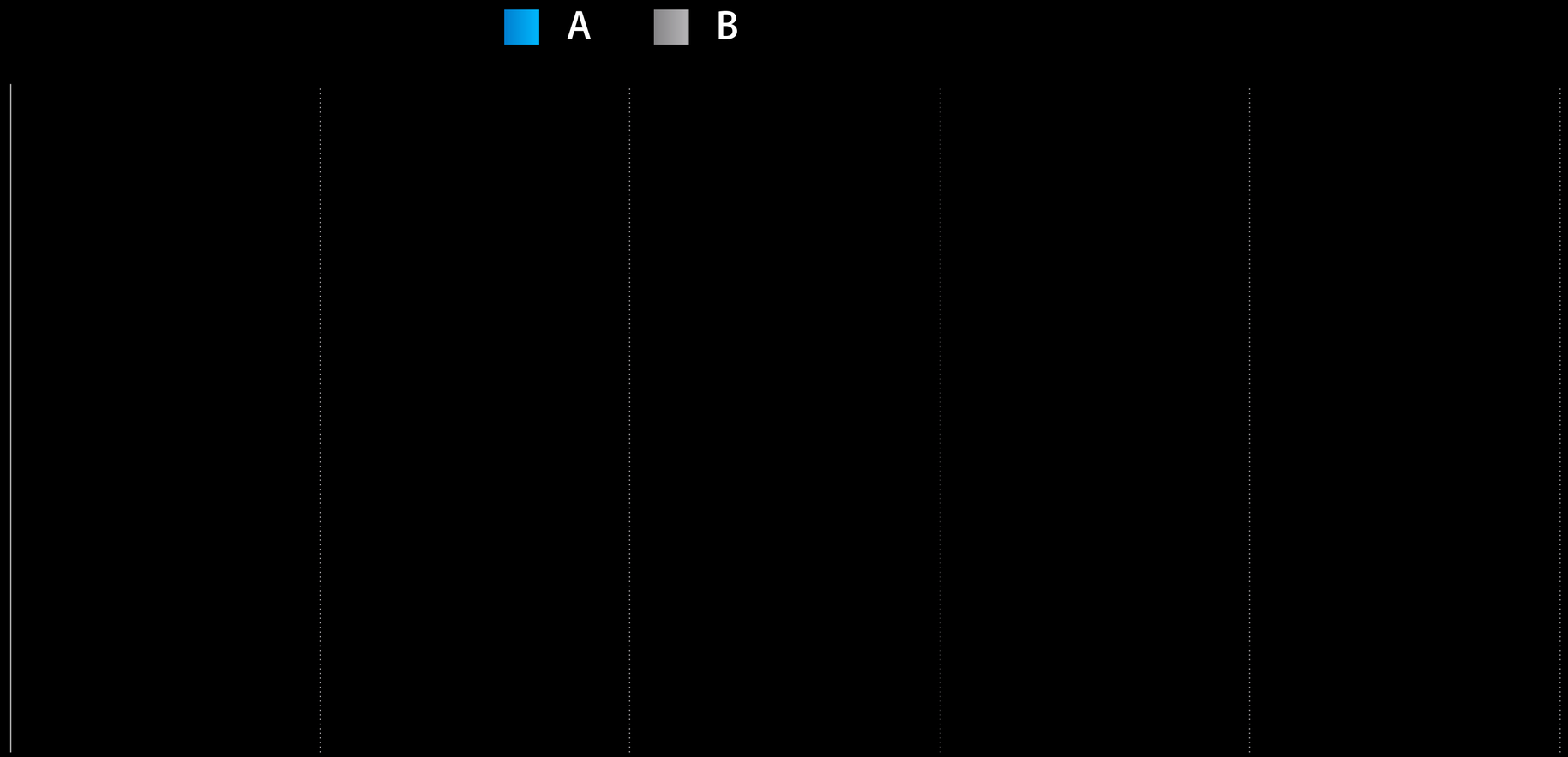
- Maximum improvement from speeding up code
  - Depends on percentage of execution time
  - Payoff is bigger for dominant pieces
- Will the payoff be worth the effort?
- Applies directly to concurrency

Picking Your Battles

Amdahl's Law example

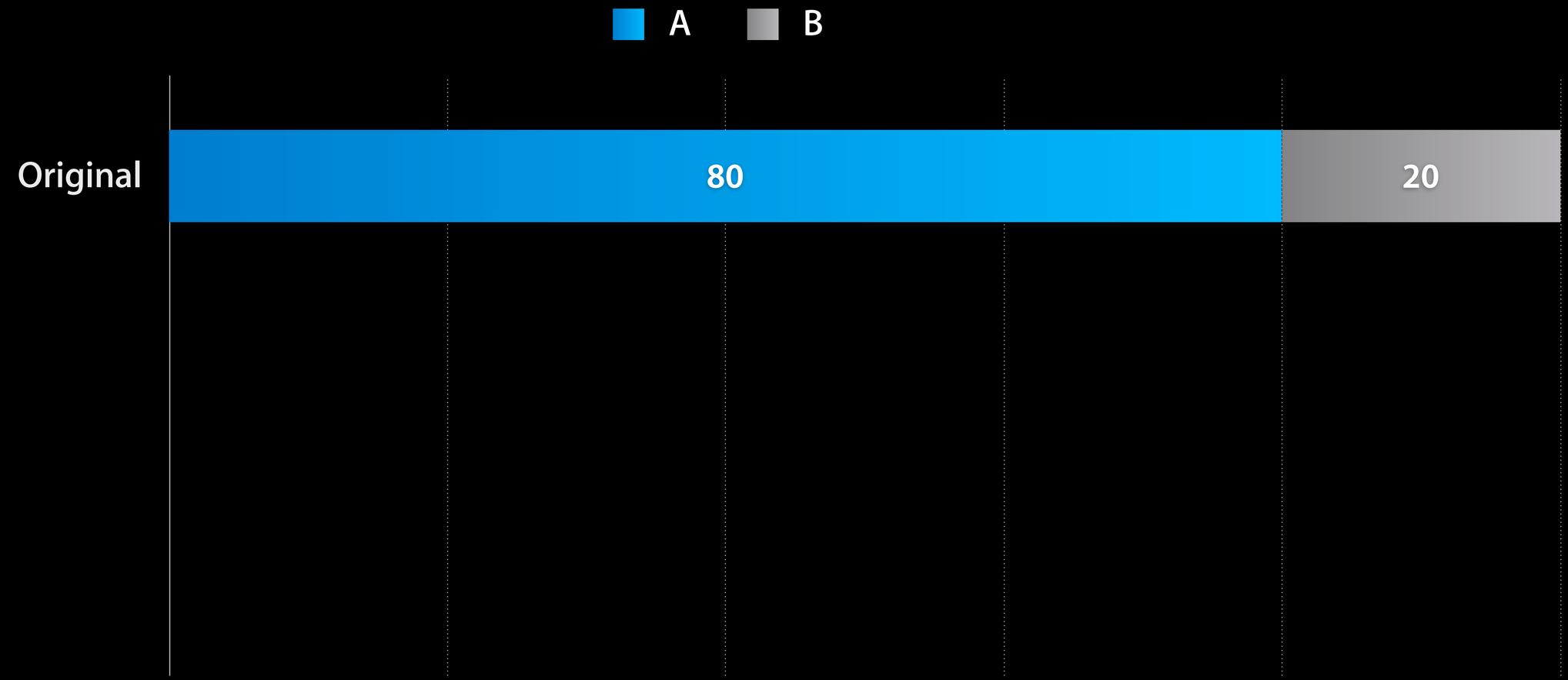
Picking Your Battles

Amdahl's Law example



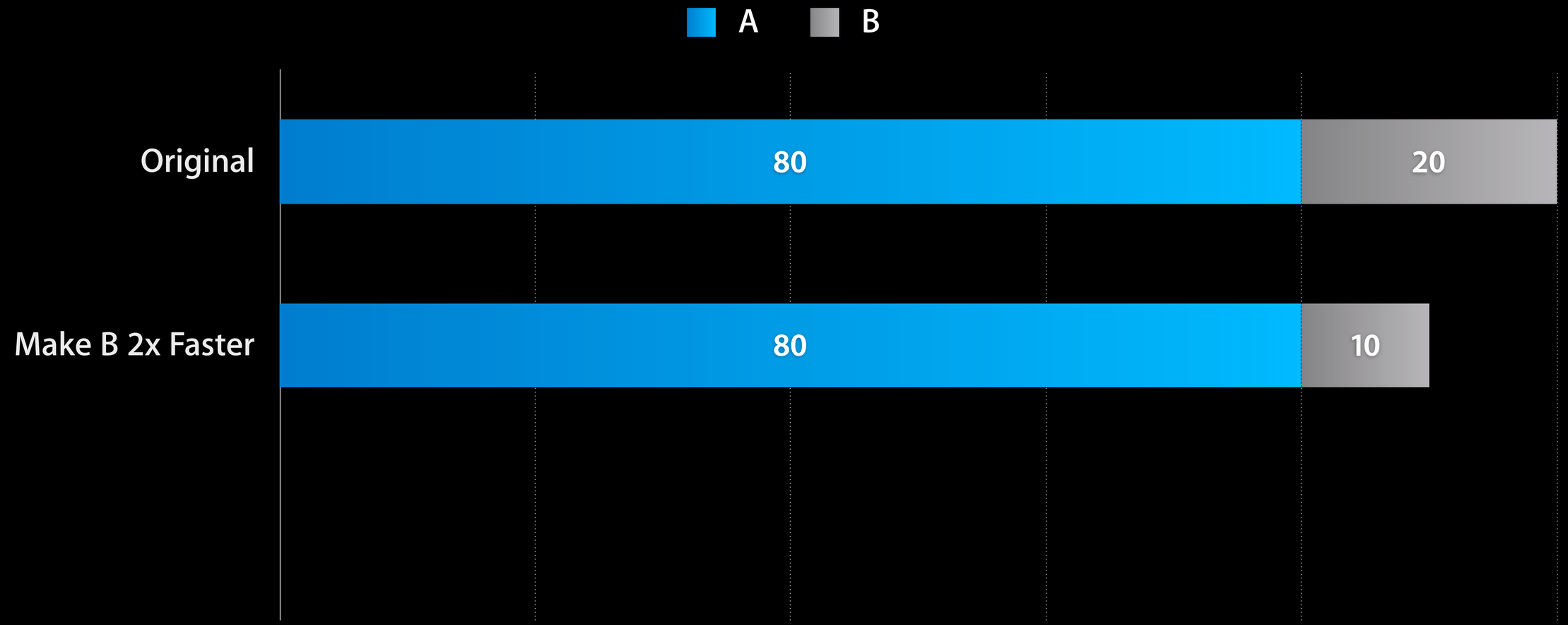
Picking Your Battles

Amdahl's Law example



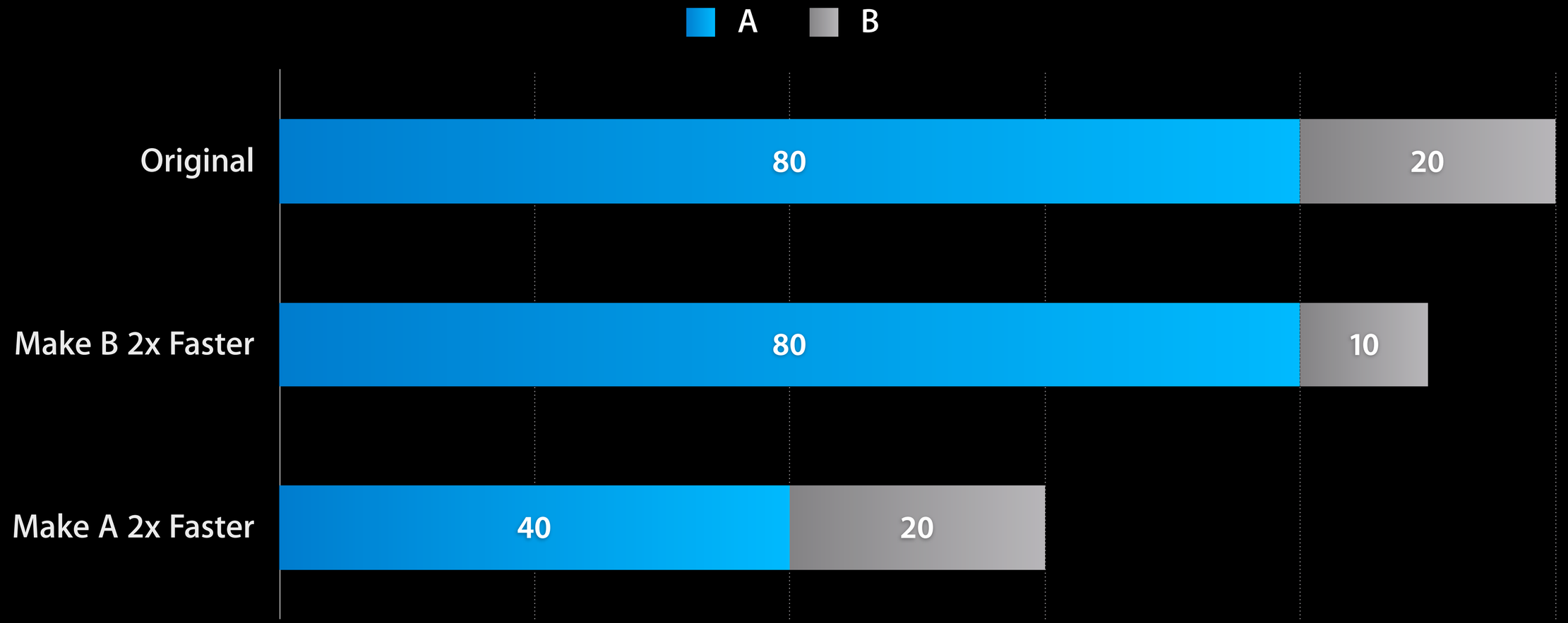
Picking Your Battles

Amdahl's Law example



Picking Your Battles

Amdahl's Law example



Premature Optimization vs. Informed Design

Premature Optimization vs. Informed Design

- Premature optimization leads to unnecessary complexity
  - “If it ain’t broke, don’t fix it.”
  - Use Instruments to focus on bottlenecks

Premature Optimization vs. Informed Design

- Premature optimization leads to unnecessary complexity
  - "If it ain't broke, don't fix it."
  - Use Instruments to focus on bottlenecks
- Informed design leads to elegant, efficient code
  - Consider performance during design
  - Intelligently avoid real performance pitfalls
  - Why design in slowness you can easily avoid?

How to Design for Performance

How to Design for Performance

- Resolving a performance issue
  - Don’t do it
  - Do it as rarely as possible
  - Do it as efficiently as possible

How to Design for Performance

- Resolving a performance issue
  - Don’t do it
  - Do it as rarely as possible
  - Do it as efficiently as possible
- Improvement requires context
  - Is this work necessary?
  - Is redundant work being done?
  - Is there a more efficient way?

How do I know if I can do better?

Computational Complexity and Cost

The Cost of Code

The Cost of Code

- Code takes time to run
  - More work takes more time
  - Short code may be work-intensive

The Cost of Code

- Code takes time to run
  - More work takes more time
  - Short code may be work-intensive
- Effects of data growth can vary dramatically
  - Work may grow disproportionately
  - Small tests won’t reveal scaling problems

The Cost of Code

- Code takes time to run
  - More work takes more time
  - Short code may be work-intensive
- Effects of data growth can vary dramatically
  - Work may grow disproportionately
  - Small tests won’t reveal scaling problems
- Can be analyzed even without running code
  - Understanding algorithm complexity is key
  - Computer Science has entire courses about this

Complexity and “Big O” Notation

Complexity and "Big O" Notation

- Rank algorithms by efficiency (time, memory, etc.)
  - The letter "O" represents the order (growth rate)
  - Performance change as scale increases

Complexity and "Big O" Notation

- Rank algorithms by efficiency (time, memory, etc.)
  - The letter "O" represents the order (growth rate)
  - Performance change as scale increases
- "Big O" approximates worst case
  - Ignore coefficients and lower-order terms (e.g. $6n^2+3n+2 \approx n^2$)
  - Ignore logarithm bases (e.g. $\log_{10}n \approx \log_2 n \approx \log n$)

Complexity and "Big O" Notation

- Rank algorithms by efficiency (time, memory, etc.)
  - The letter "O" represents the order (growth rate)
  - Performance change as scale increases
- "Big O" approximates worst case
  - Ignore coefficients and lower-order terms (e.g. $6n^2+3n+2 \approx n^2$)
  - Ignore logarithm bases (e.g. $\log_{10} n \approx \log_2 n \approx \log n$)
- For any given task, there are inherent limits
  - Some things just take time

“Nine women can’t
make a baby in one month.”

Fred Brooks — *“The Mythical Man-Month”* (1975)

“Big O” Complexity

Order functions

“Big O” Complexity

Order functions

Notation	Name	Examples
O(1)	constant time	simple expressions, indexed/hashed lookup
O(log n)	logarithmic time	search of sorted data
O(n)	linear time	search of unsorted data, enumeration, “for each” loop
O(n log n)	log-linear time	efficient sorting algorithms
$O(n^2)$	quadratic time	nested iteration of data
$O(c^n)$	exponential time	combinatorial explosion, high-dimensional data

“Big O” Complexity

Order functions

- Some are much more common

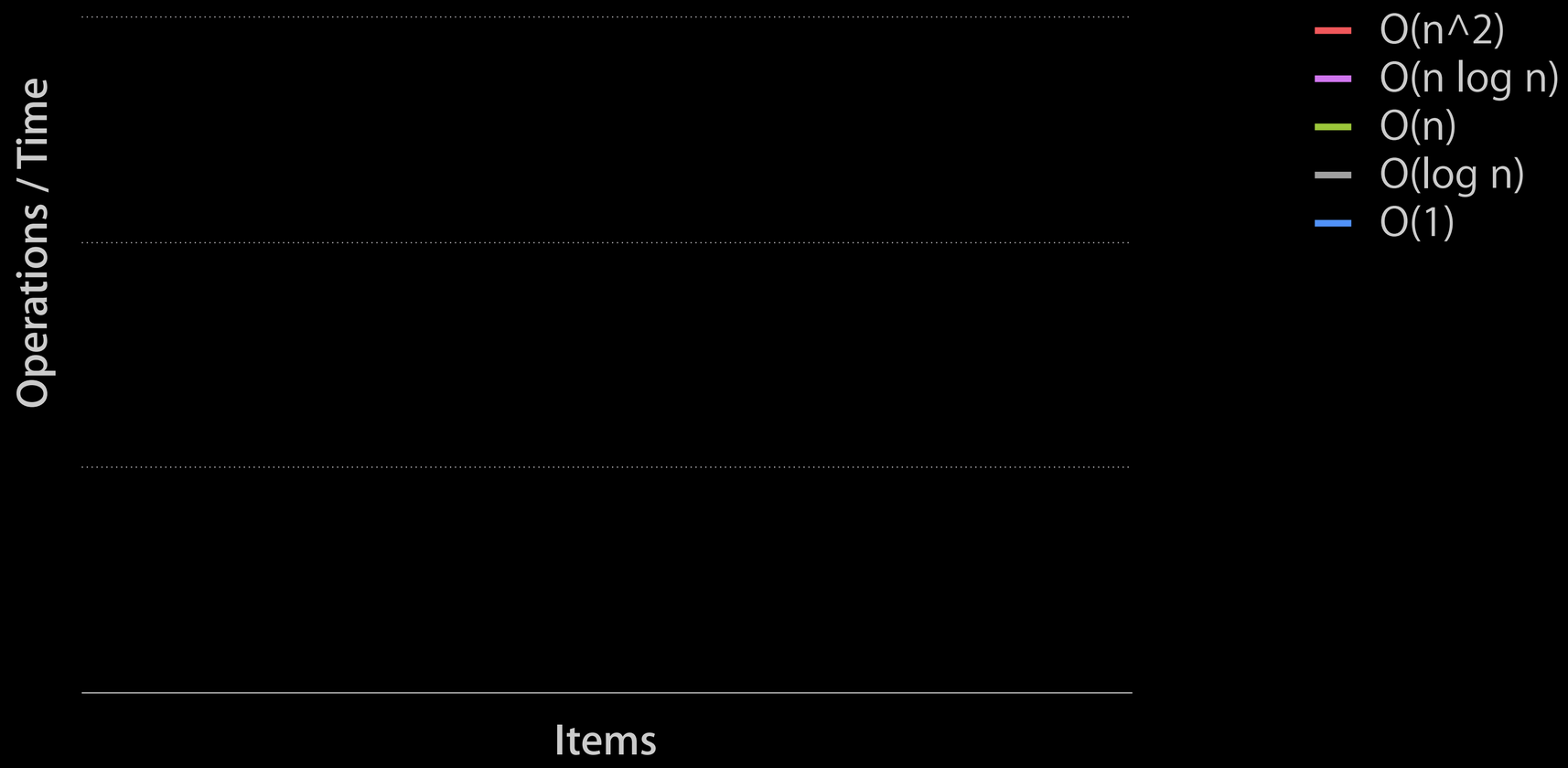
	Notation	Name	Examples
✓	O(1)	constant time	simple expressions, indexed/hashed lookup
	O(log n)	logarithmic time	search of sorted data
✓	O(n)	linear time	search of unsorted data, enumeration, “for each” loop
	O(n log n)	log-linear time	efficient sorting algorithms
✓	$O(n^2)$	quadratic time	nested iteration of data
	$O(c^n)$	exponential time	combinatorial explosion, high-dimensional data

“Big O” Complexity

Order functions comparison

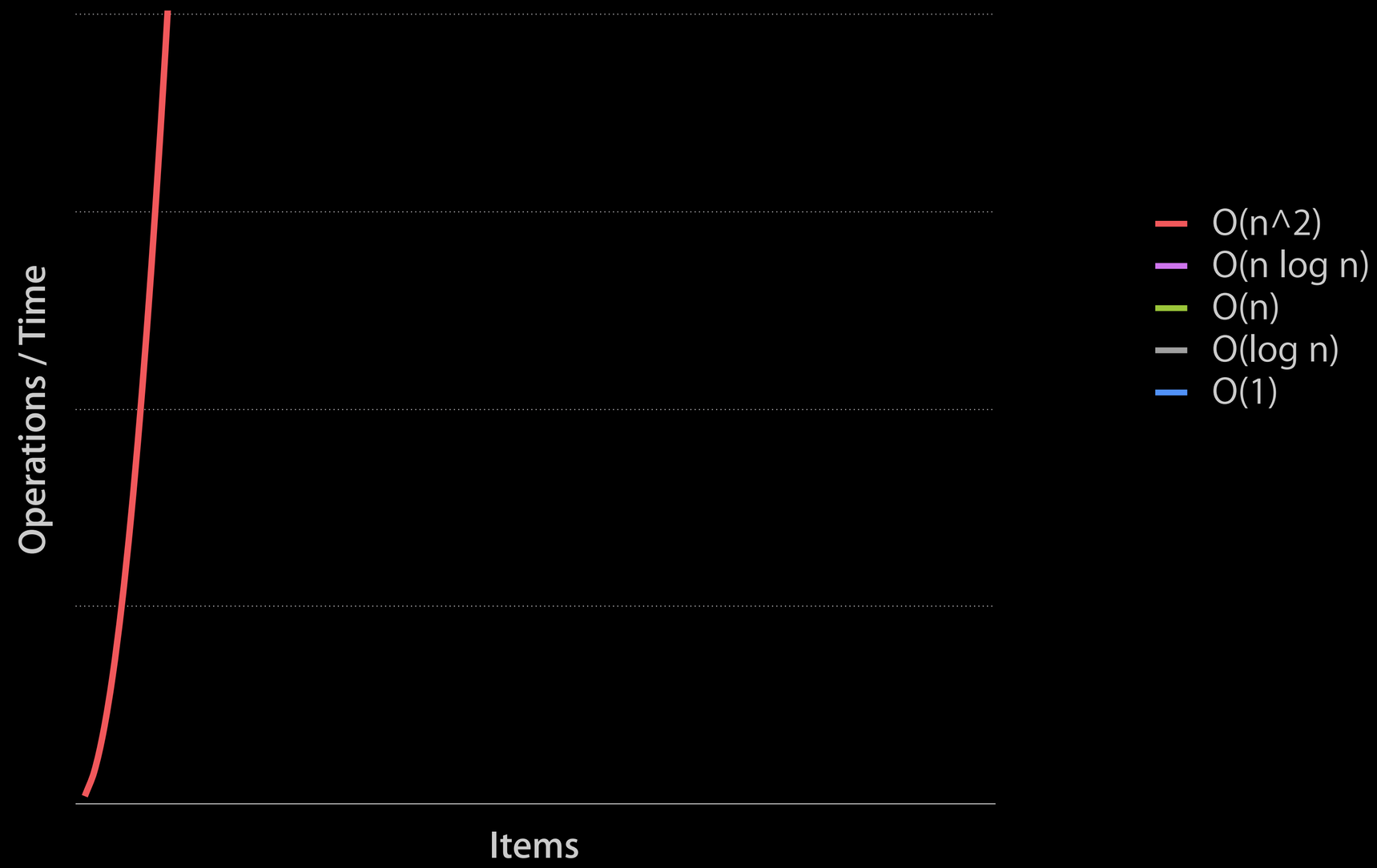
“Big O” Complexity

Order functions comparison



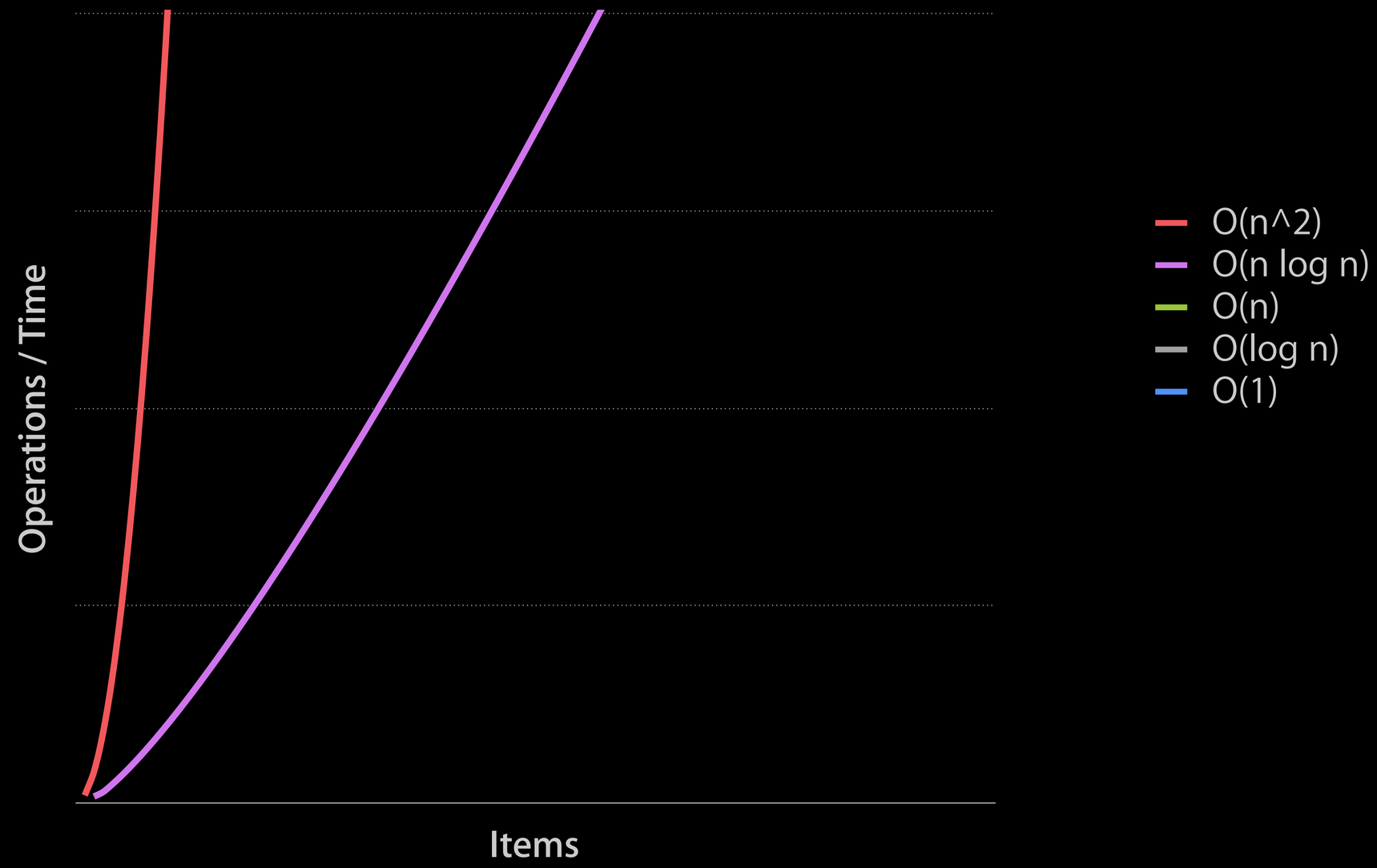
“Big O” Complexity

Order functions comparison



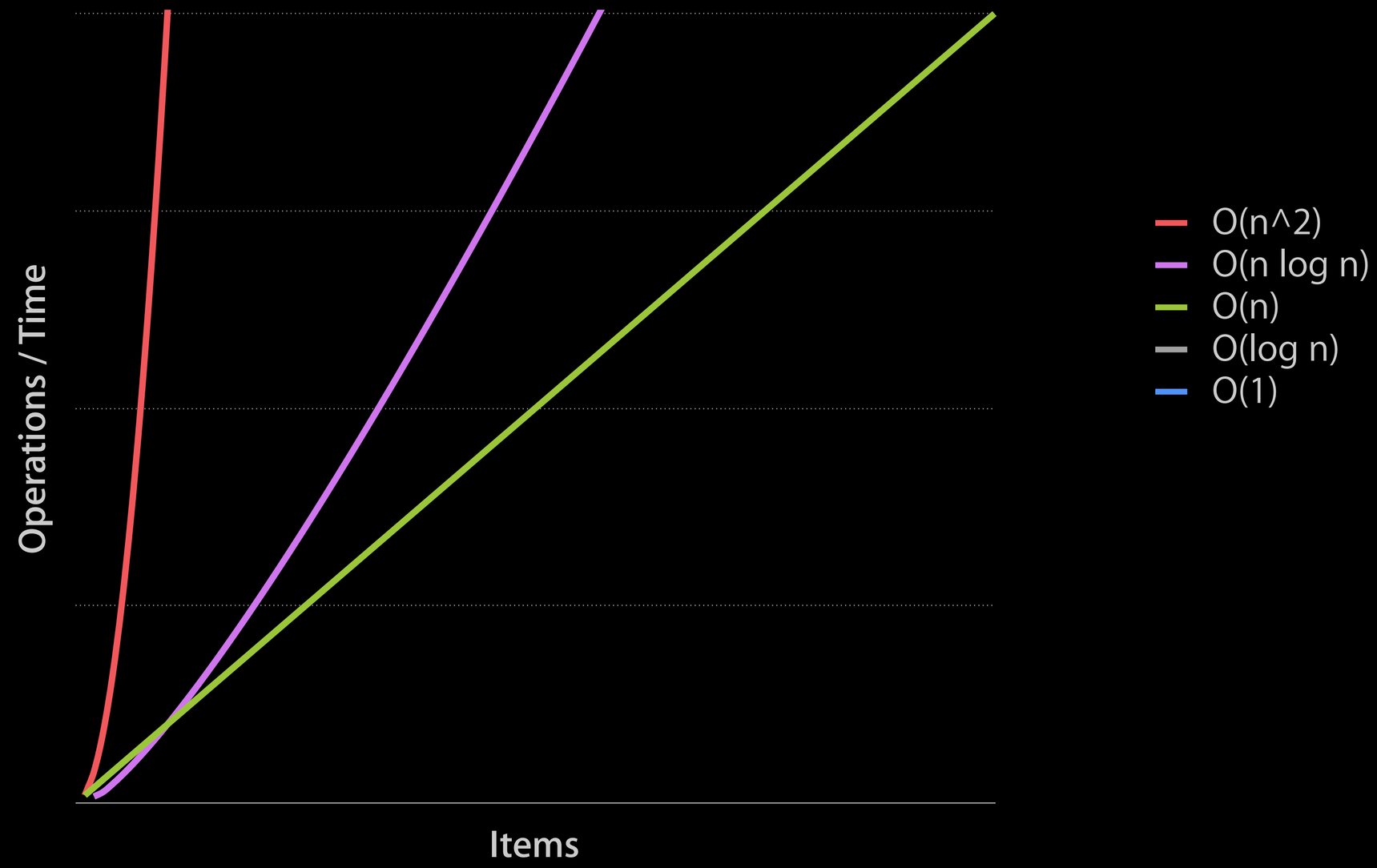
“Big O” Complexity

Order functions comparison



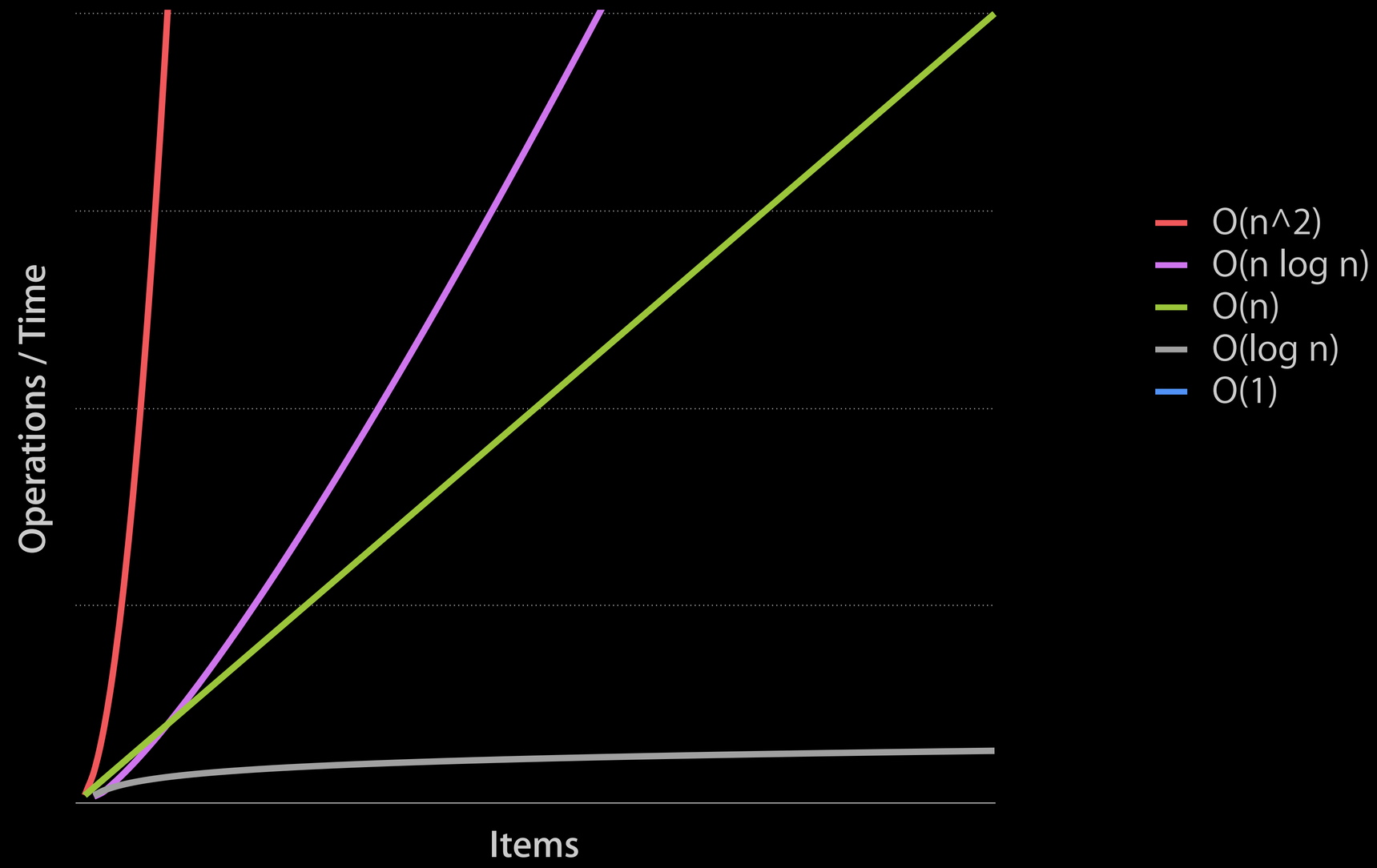
“Big O” Complexity

Order functions comparison



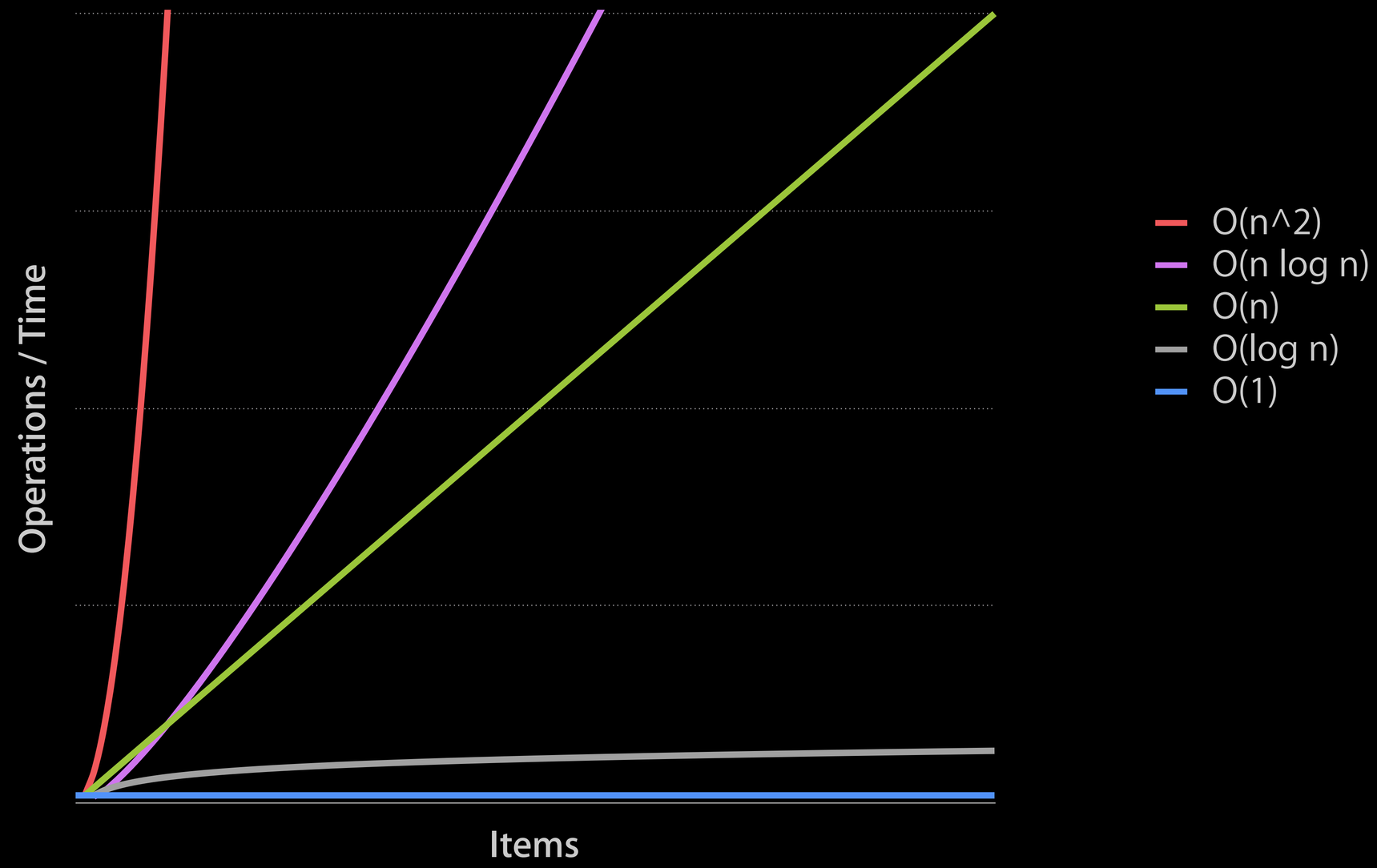
“Big O” Complexity

Order functions comparison



“Big O” Complexity

Order functions comparison



“Big O” Complexity

Examples

- Determine complexity from source code
- Growth of work with size is key

“Big O” Complexity

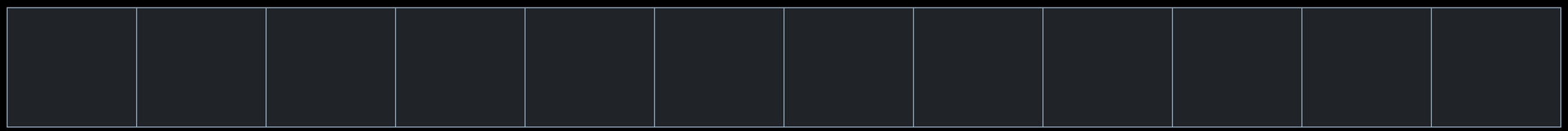
Example: O(1)

```
BOOL ContainsValueAtIndex(int[] array, int count, int value, int idx) {
    return (idx < count && array[idx] == value);
}
```

“Big O” Complexity

Example: O(1)

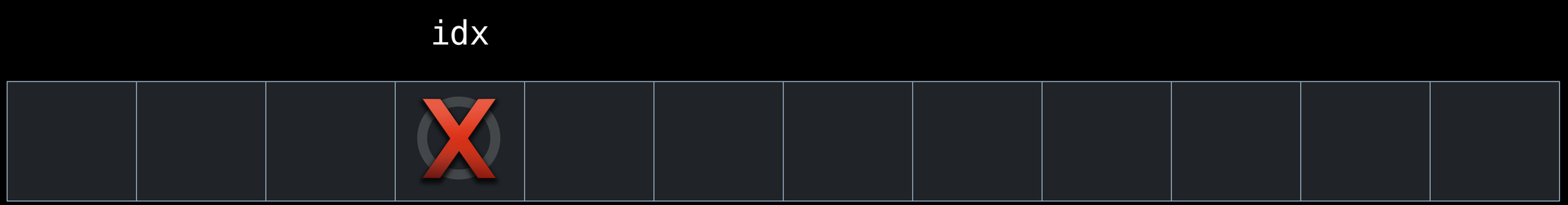
```
BOOL ContainsValueAtIndex(int[] array, int count, int value, int idx) {
    return (idx < count && array[idx] == value);
}
```



“Big O” Complexity

Example: O(1)

```
BOOL ContainsValueAtIndex(int[] array, int count, int value, int idx) {
    return (idx < count && array[idx] == value);
}
```



“Big O” Complexity

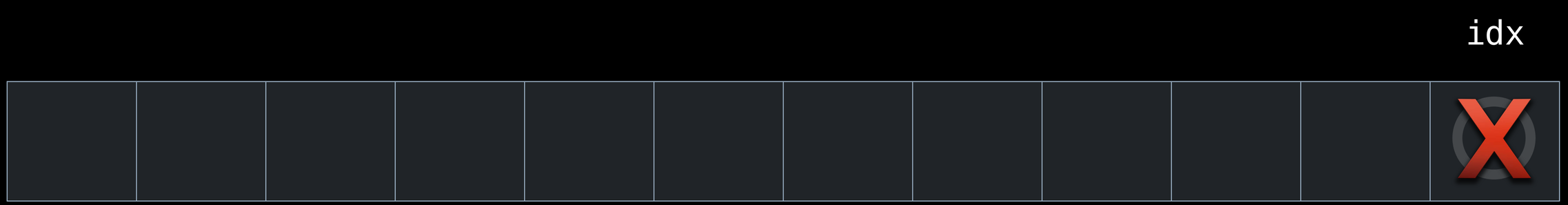
Example: O(1)

```
BOOL ContainsValueAtIndex(int[] array, int count, int value, int idx) {
    return (idx < count && array[idx] == value);
}
```

“Big O” Complexity

Example: O(1)

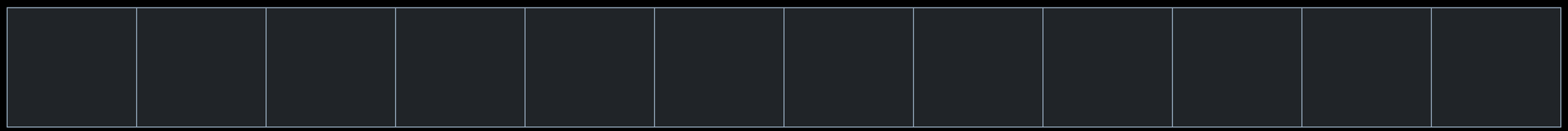
```
BOOL ContainsValueAtIndex(int[] array, int count, int value, int idx) {
    return (idx < count && array[idx] == value);
}
```



“Big O” Complexity

Example: O(1)

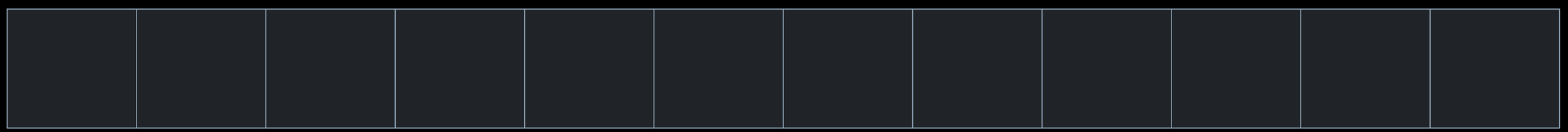
```
BOOL ContainsValueAtIndex(int[] array, int count, int value, int idx) {
    return (idx < count && array[idx] == value);
}
```



“Big O” Complexity

Example: O(n)

```
BOOL ContainsValue(int[] array, int count, int value) {
    for (int i = 0; i < count; i++) {      // O(n)
        if (array[i] == value) {
            return YES;
        }
    }
    return NO;
}
```



“Big O” Complexity

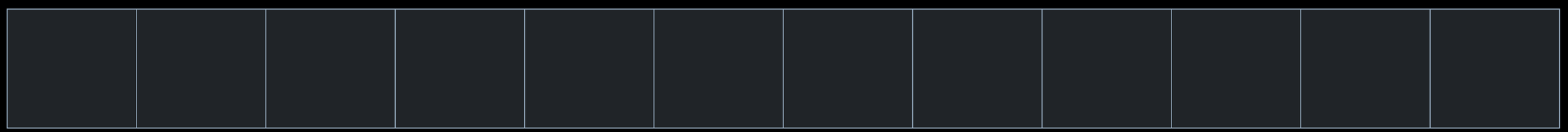
Example: O(n)

```
BOOL ContainsValue(int[] array, int count, int value) {
    for (int i = 0; i < count; i++) {      // O(n)
        if (array[i] == value) {
            return YES;
        }
    }
    return NO;
}
```

“Big O” Complexity

Example: $O(n^2)$

```
BOOL ContainsDuplicateValues(int[] array, int count) {
    for (int i = 0; i < count; i++) {       // O(n)
        for (int j = 0; j < count; j++) {   // O(n)
            if (i != j && array[i] == array[j]) {
                return YES;
            }
        }
    }
    return NO;
}
```



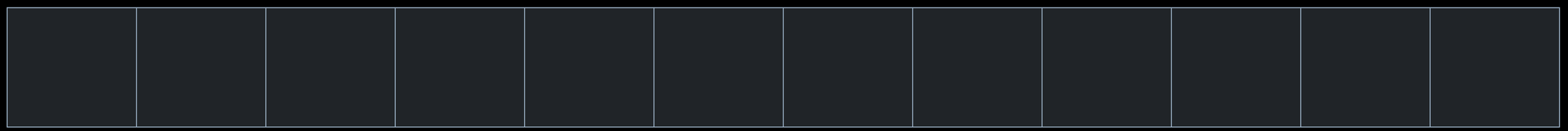
“Big O” Complexity

Example: $O(n^2)$

```
BOOL ContainsDuplicateValues(int[] array, int count) {
    for (int i = 0; i < count; i++) {       // O(n)
        for (int j = 0; j < count; j++) {  // O(n)
            if (i != j && array[i] == array[j]) {
                return YES;
            }
        }
    }
    return NO;
}
```

i

j



“Big O” Complexity

Example: $O(n^2)$

```
BOOL ContainsDuplicateValues(int[] array, int count) {
    for (int i = 0; i < count; i++) {       // O(n)
        for (int j = 0; j < count; j++) {   // O(n)
            if (i != j && array[i] == array[j]) {
                return YES;
            }
        }
    }
    return NO;
}
```

Calculating Complexity

Calculating Complexity

- Order functions can be combined
  - Multiply nested complexities
  - Add sequential complexities

Calculating Complexity

- Order functions can be combined
  - Multiply nested complexities
  - Add sequential complexities
- This function reduces to $O(n^2)$

```
void AnalyzeArray(int[] array, int count) {
    ContainsDuplicateValues(array, count);    // O(n^2)
    ContainsValue(array, count, 0);           // O(n)
    ContainsValue(array, count, 1);           // O(n)
    ContainsValueAtIndex(array, count, 1, 0); // O(1)
    ContainsValueAtIndex(array, count, 0, 1); // O(1)
    ContainsValueAtIndex(array, count, 1, 1); // O(1)
}
```

Estimating Complexity

Estimating Complexity

- When you don’t know, estimate
  - Consider what the code does
  - Profile with Instruments

Estimating Complexity

- When you don’t know, estimate
  - Consider what the code does
  - Profile with Instruments
- Some APIs appear similar, but don’t assume
  - For example, `-containsObject:` on NSArray and NSSet

Estimating Complexity

Reading between the lines

Estimating Complexity

Reading between the lines

- Consider `-[NSArray containsObject:]`
  - What does it do?
  - Documentation says it sends `-isEqual:` to each object

Estimating Complexity

Reading between the lines

- Consider `-[NSArray containsObject:]`
  - What does it do?
  - Documentation says it sends `-isEqual:` to each object
- This sounds like O(n)

Estimating Complexity

Reading between the lines

- Consider `-[NSArray containsObject:]`
  - What does it do?
  - Documentation says it sends `-isEqual:` to each object
- This sounds like O(n)
- Concurrency would only reduce by some factor

Estimating Complexity

Jumping to the wrong conclusion

Estimating Complexity

Jumping to the wrong conclusion

- Consider `-[NSSet containsObject:]`

Estimating Complexity

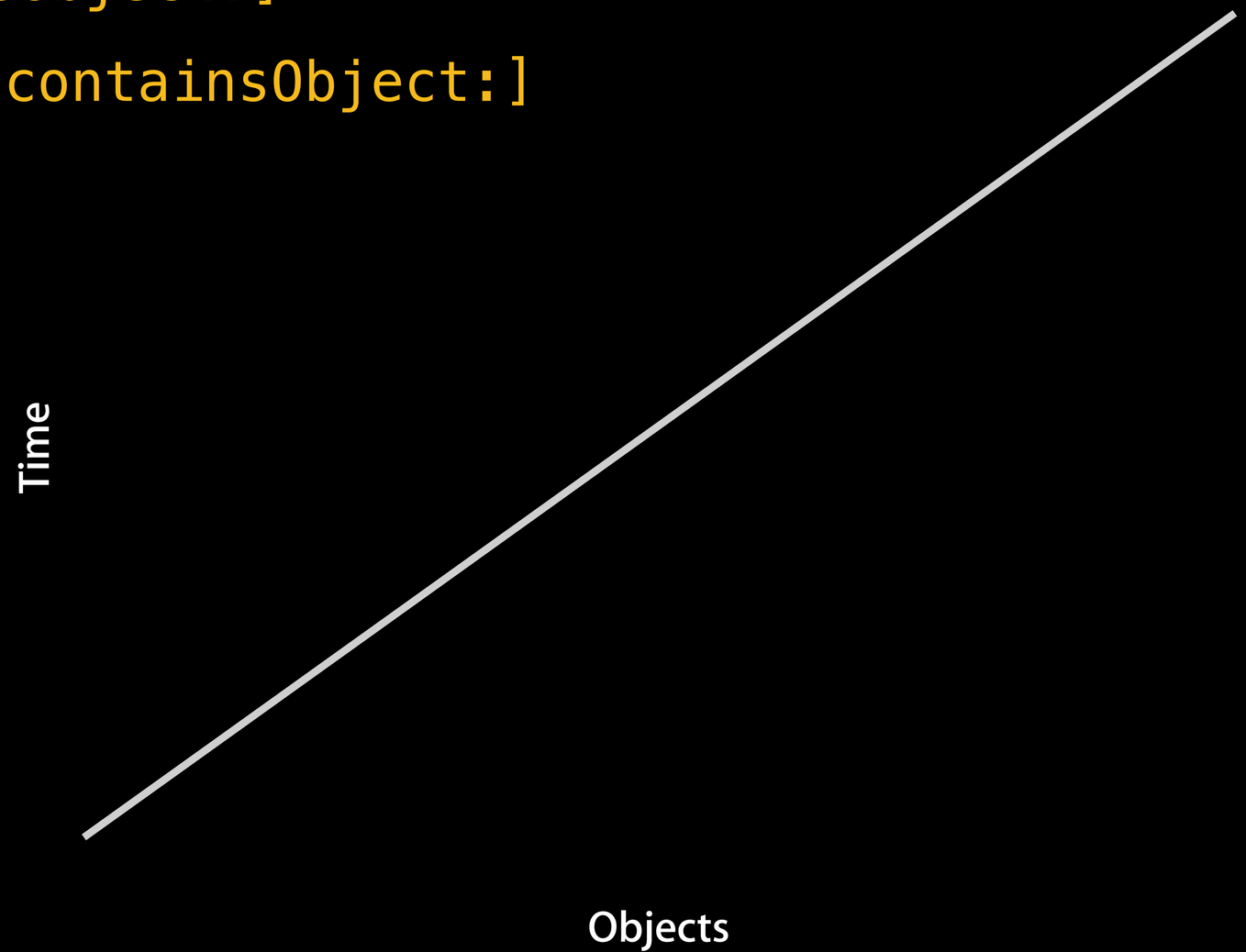
Jumping to the wrong conclusion

- Consider `-[NSSet containsObject:]`
- It looks just like `-[NSArray containsObject:]`

Estimating Complexity

Jumping to the wrong conclusion

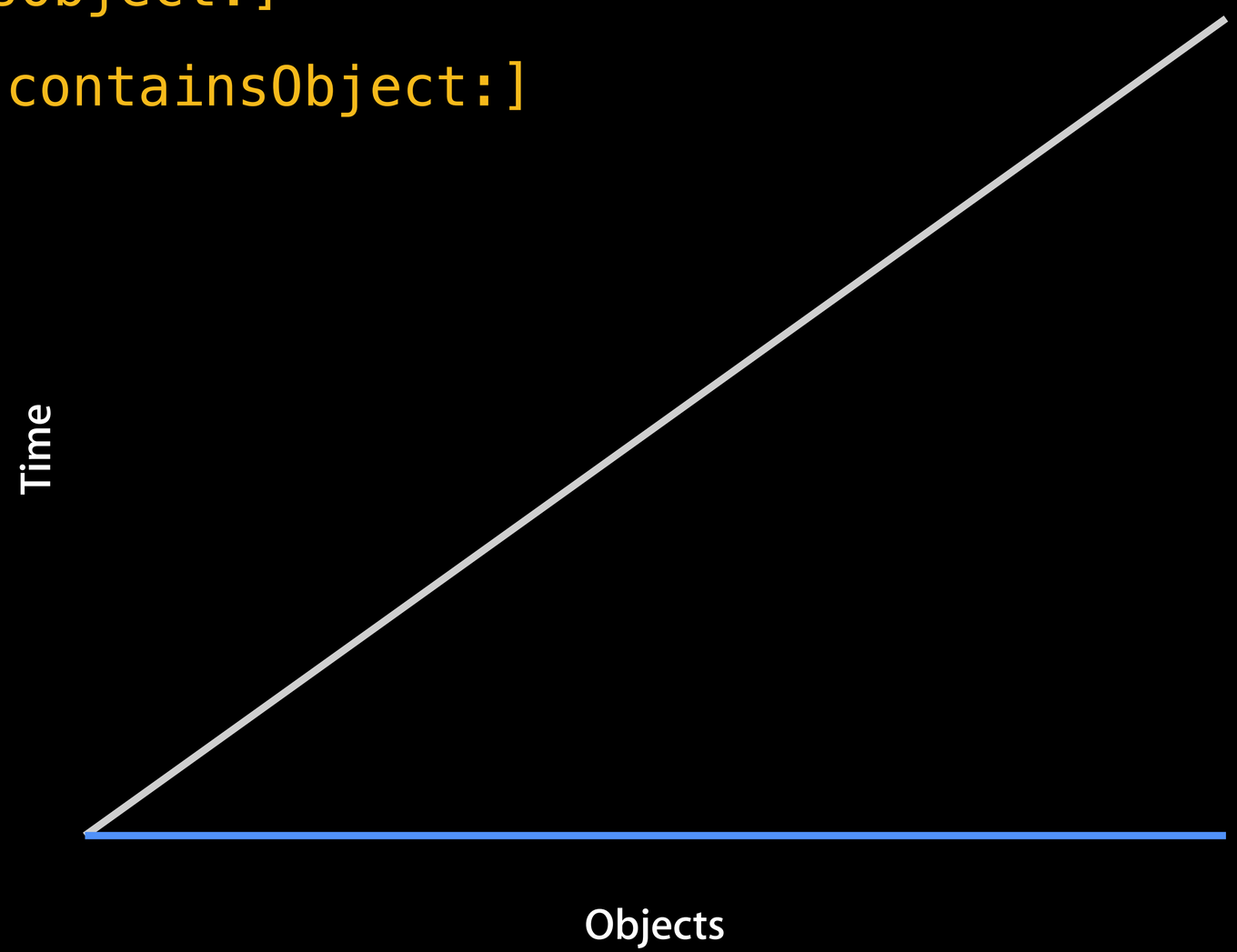
- Consider `-[NSSet containsObject:]`
- It looks just like `-[NSArray containsObject:]`
- Is it also O(n)?



Estimating Complexity

Jumping to the wrong conclusion

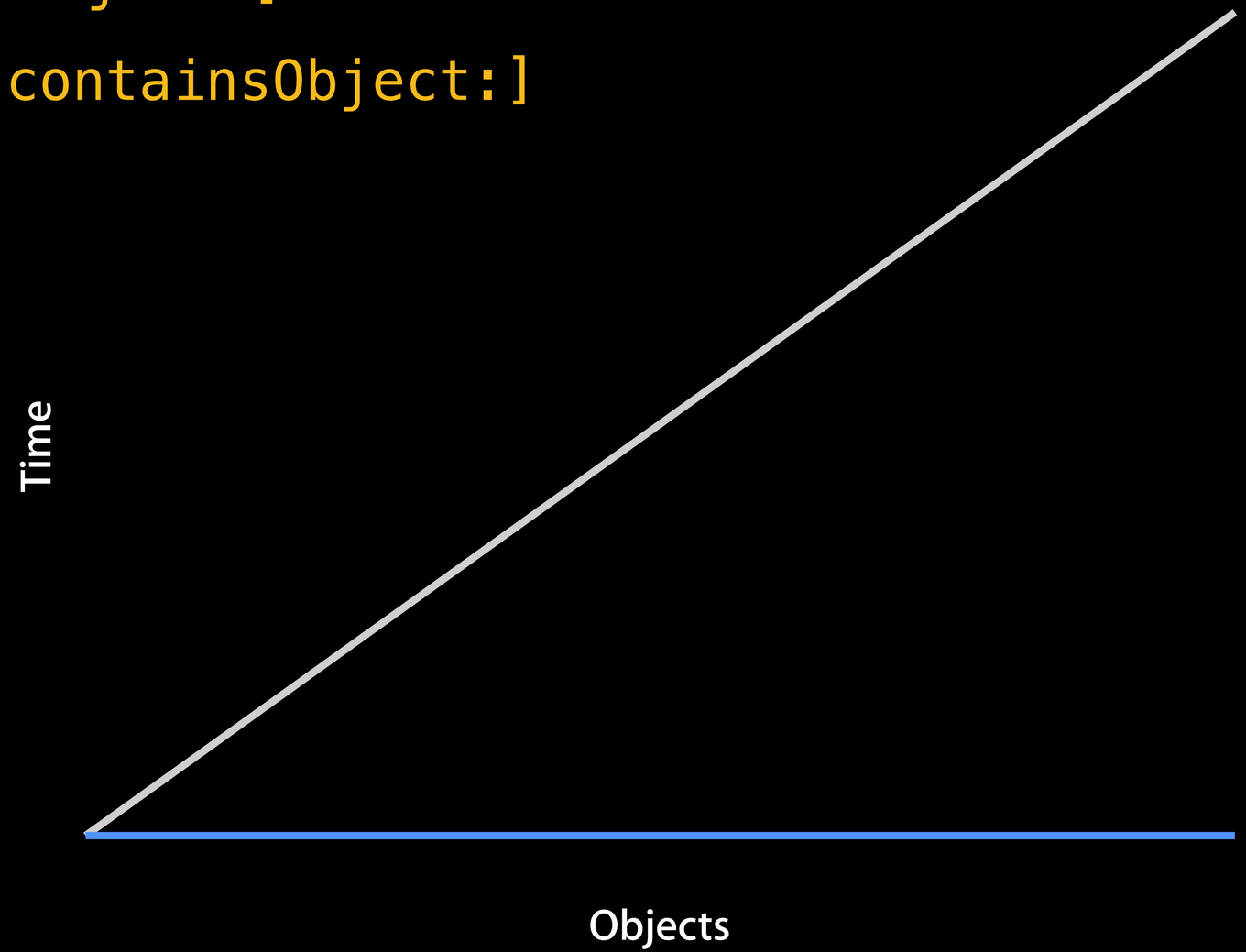
- Consider `–[NSSet containsObject:]`
- It looks just like `–[NSArray containsObject:]`
- Is it also O(n)?
- It's actually O(1)



Estimating Complexity

Jumping to the wrong conclusion

- Consider `-[NSSet containsObject:]`
- It looks just like `-[NSArray containsObject:]`
- Is it also O(n)?
- It's actually O(1)
- It must do less work



Hash-Based Organization

Hash-Based Organization

- NSSet uses a hash table for storage

Hash-Based Organization

- NSSet uses a hash table for storage
- Objects have a deterministic hash value
  - –hash returns an NSUInteger
  - Equal objects have the same hash

Hash-Based Organization

- NSSet uses a hash table for storage
- Objects have a deterministic hash value
  - –hash returns an NSUInteger
  - Equal objects have the same hash
- Objects are grouped in “buckets”
  - A hash function maps hashes to buckets
  - Goal is uniform distribution

Hash-Based Organization

- NSSet uses a hash table for storage
- Objects have a deterministic hash value
  - `–hash` returns an NSUInteger
  - Equal objects have the same hash
- Objects are grouped in “buckets”
  - A hash function maps hashes to buckets
  - Goal is uniform distribution
- Lookup only considers objects in one bucket
  - Check `–isEqual:` for very few objects (or none)

Hash-Based Organization

Hash Function

Buckets

0	
1	
2	
3	
4	
5	

Hash-Based Organization

@"Tim Cook"

Hash Function

Buckets

0	
1	
2	
3	
4	
5	

Hash-Based Organization

@"Tim Cook"

Hash Function

Buckets

0	
1	
2	@"Tim Cook"
3	
4	
5	

Hash-Based Organization

@"Tim Cook"

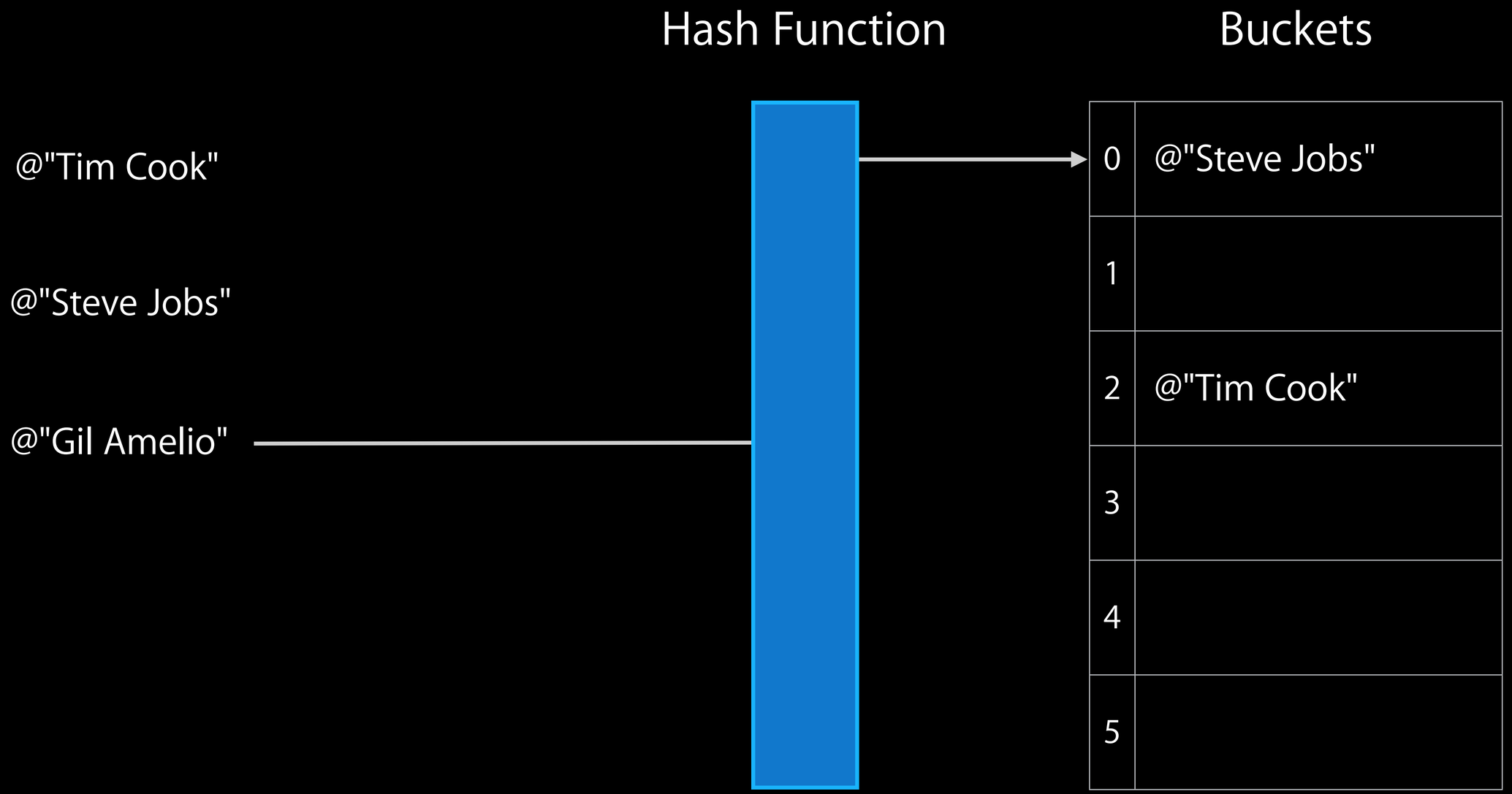
@"Steve Jobs"

Hash Function

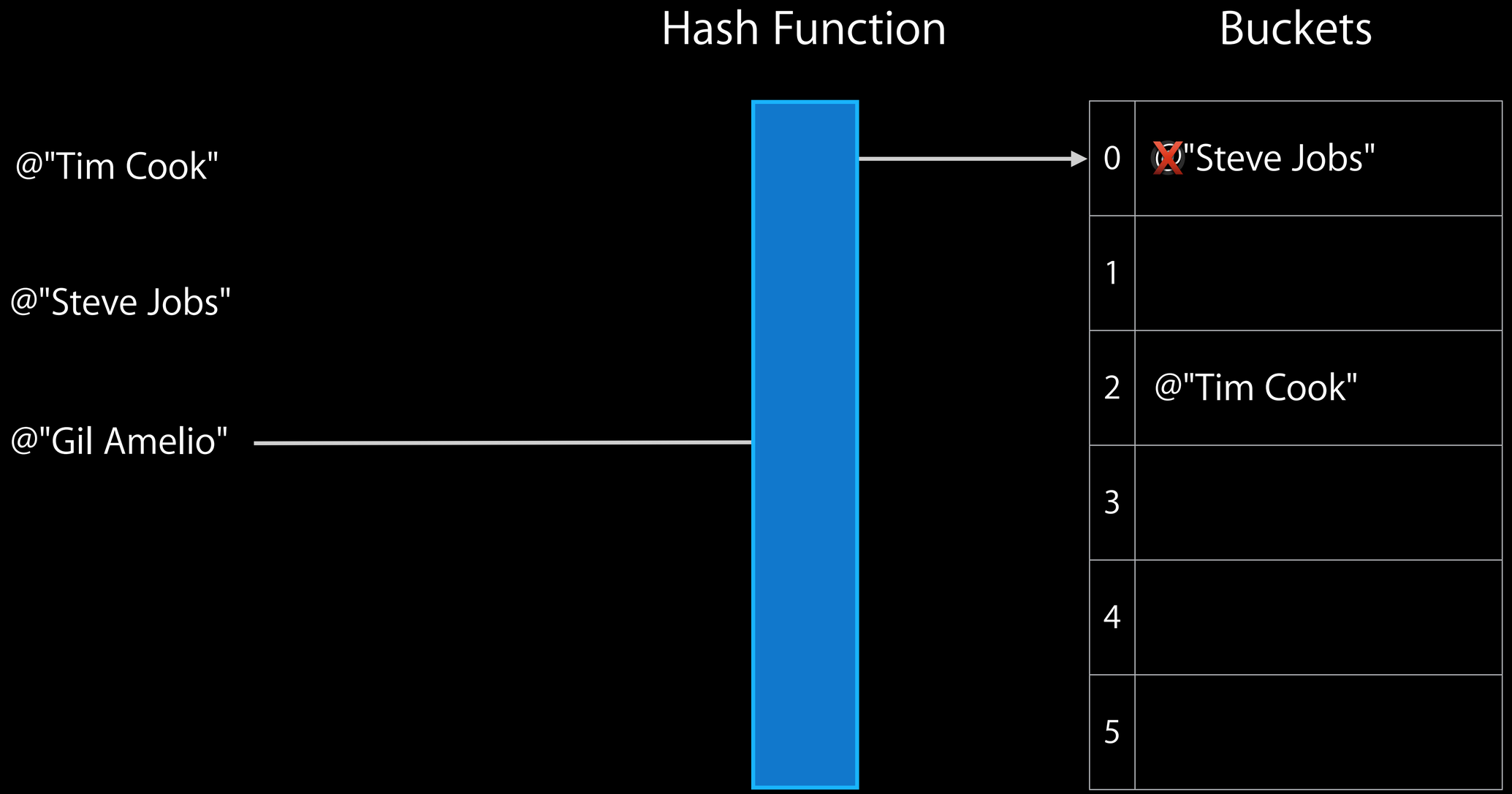
Buckets

0	@"Steve Jobs"
1	
2	@"Tim Cook"
3	
4	
5	

Hash-Based Organization



Hash-Based Organization



Hash-Based Organization

@"Tim Cook"

@"Steve Jobs"

@"Gil Amelio"

Hash Function

Buckets

0	@"Steve Jobs" @"Gil Amelio"
1	
2	@"Tim Cook"
3	
4	
5	

Hash-Based Organization

@"Tim Cook"

@"Steve Jobs"

@"Gil Amelio"

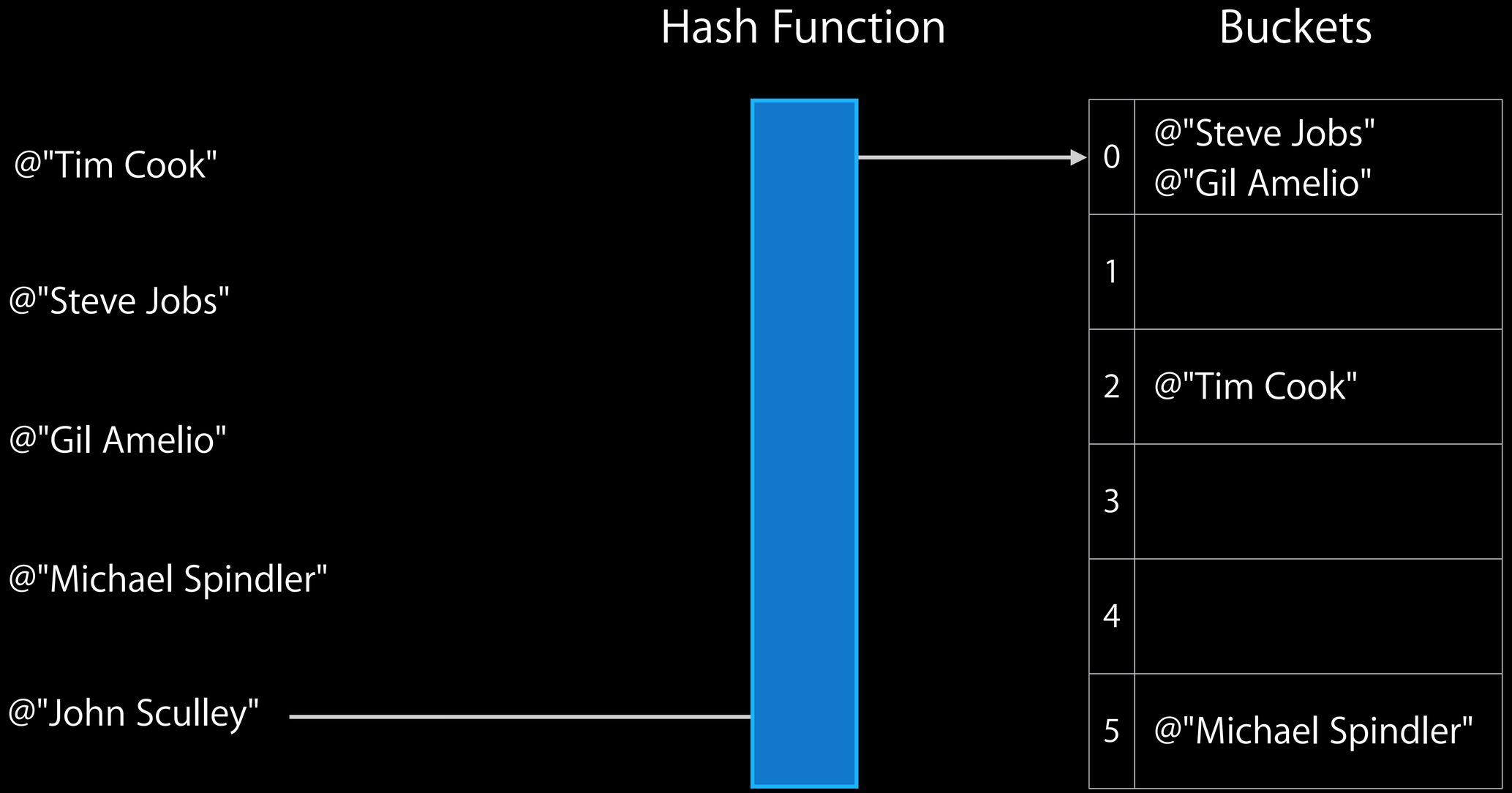
@"Michael Spindler"

Hash Function

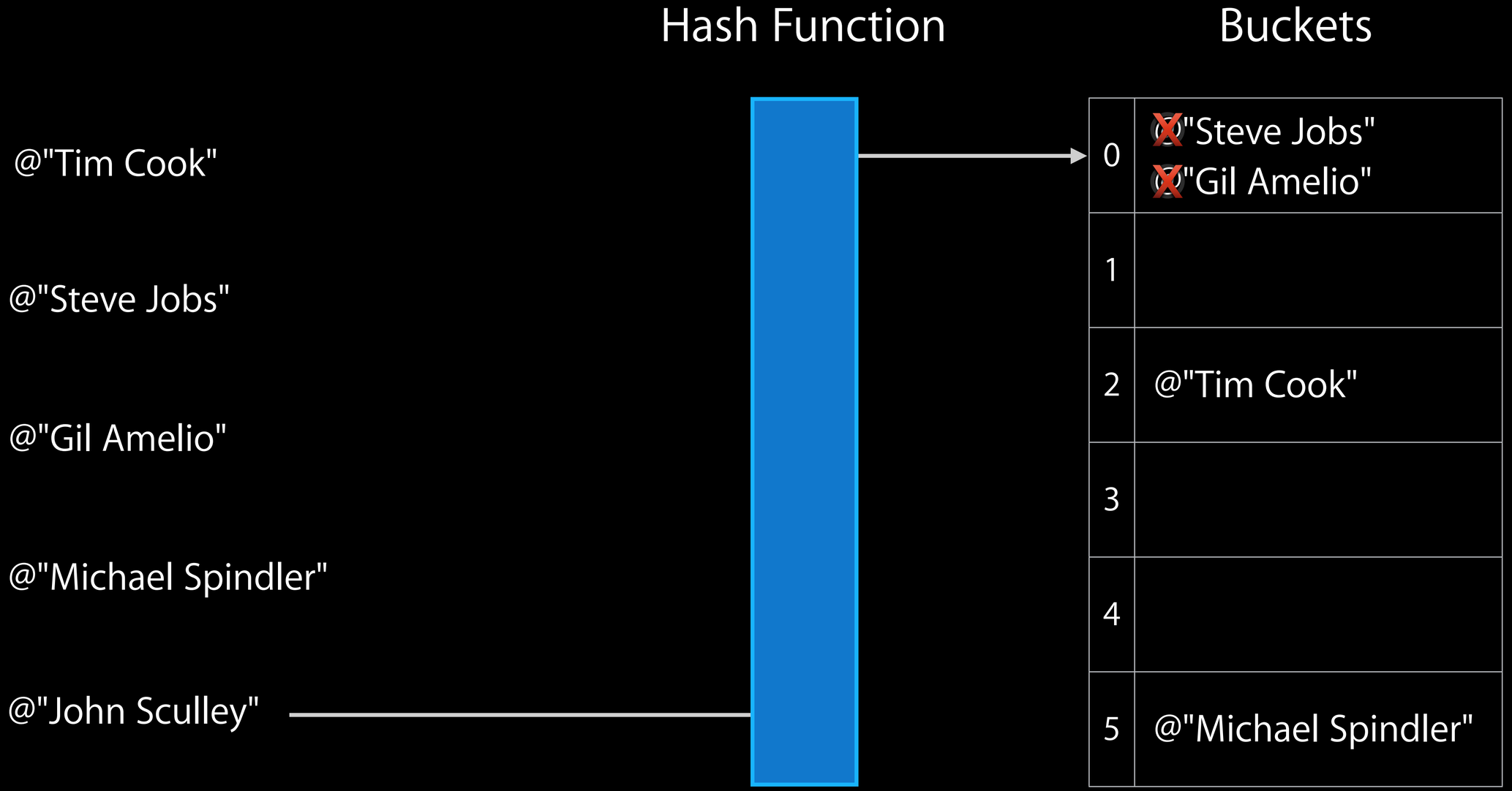
Buckets

0	@"Steve Jobs" @"Gil Amelio"
1	
2	@"Tim Cook"
3	
4	
5	@"Michael Spindler"

Hash-Based Organization



Hash-Based Organization



Hash-Based Organization

@"Tim Cook"

@"Steve Jobs"

@"Gil Amelio"

@"Michael Spindler"

@"John Sculley"

Hash Function

Buckets

0	@"Steve Jobs" @"Gil Amelio" @"John Sculley"
1	
2	@"Tim Cook"
3	
4	
5	@"Michael Spindler"

Hash-Based Organization

@"Tim Cook"

@"Steve Jobs"

@"Gil Amelio"

@"Michael Spindler"

@"John Sculley"

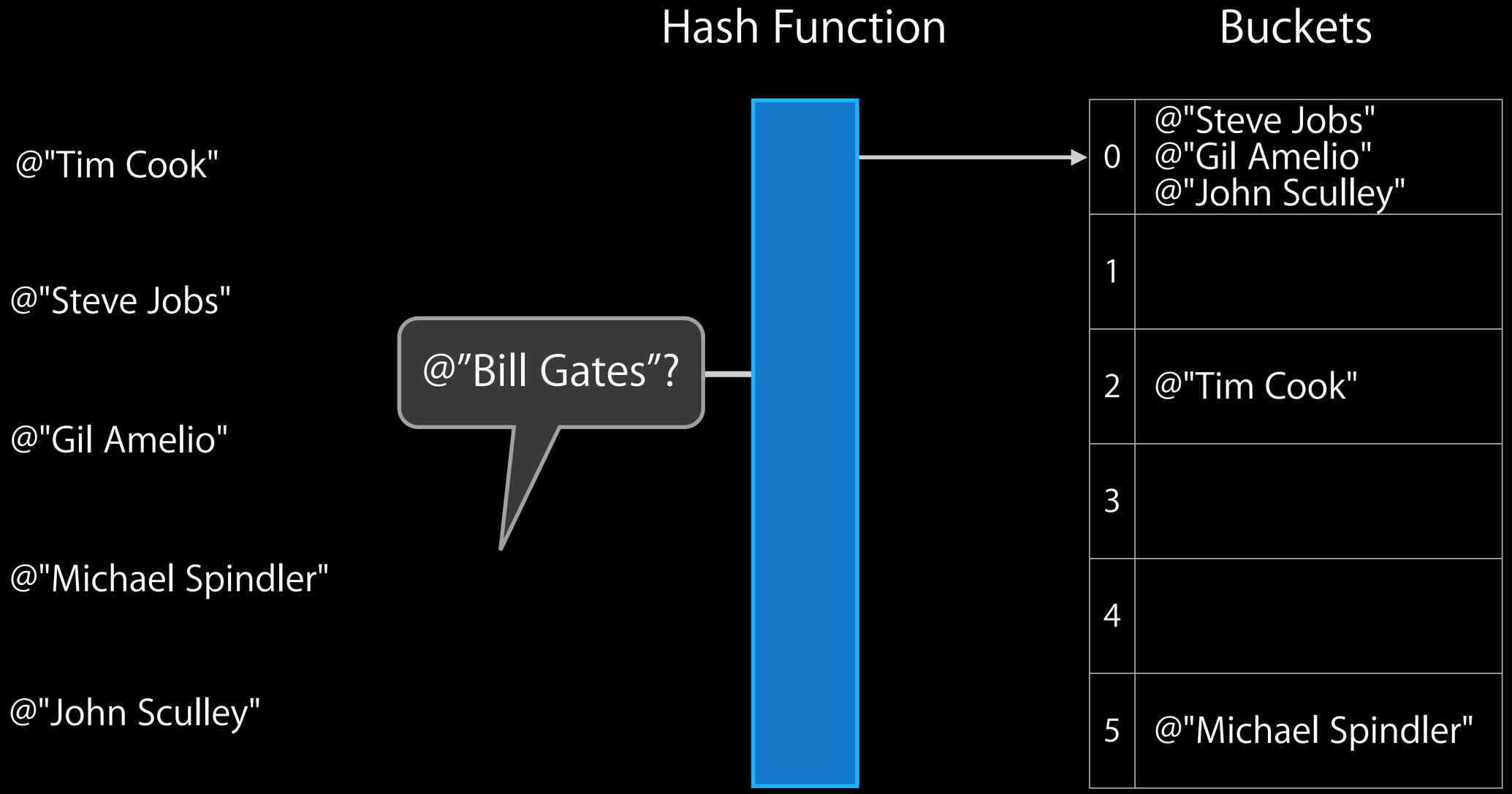
@”Bill Gates”?

Hash Function

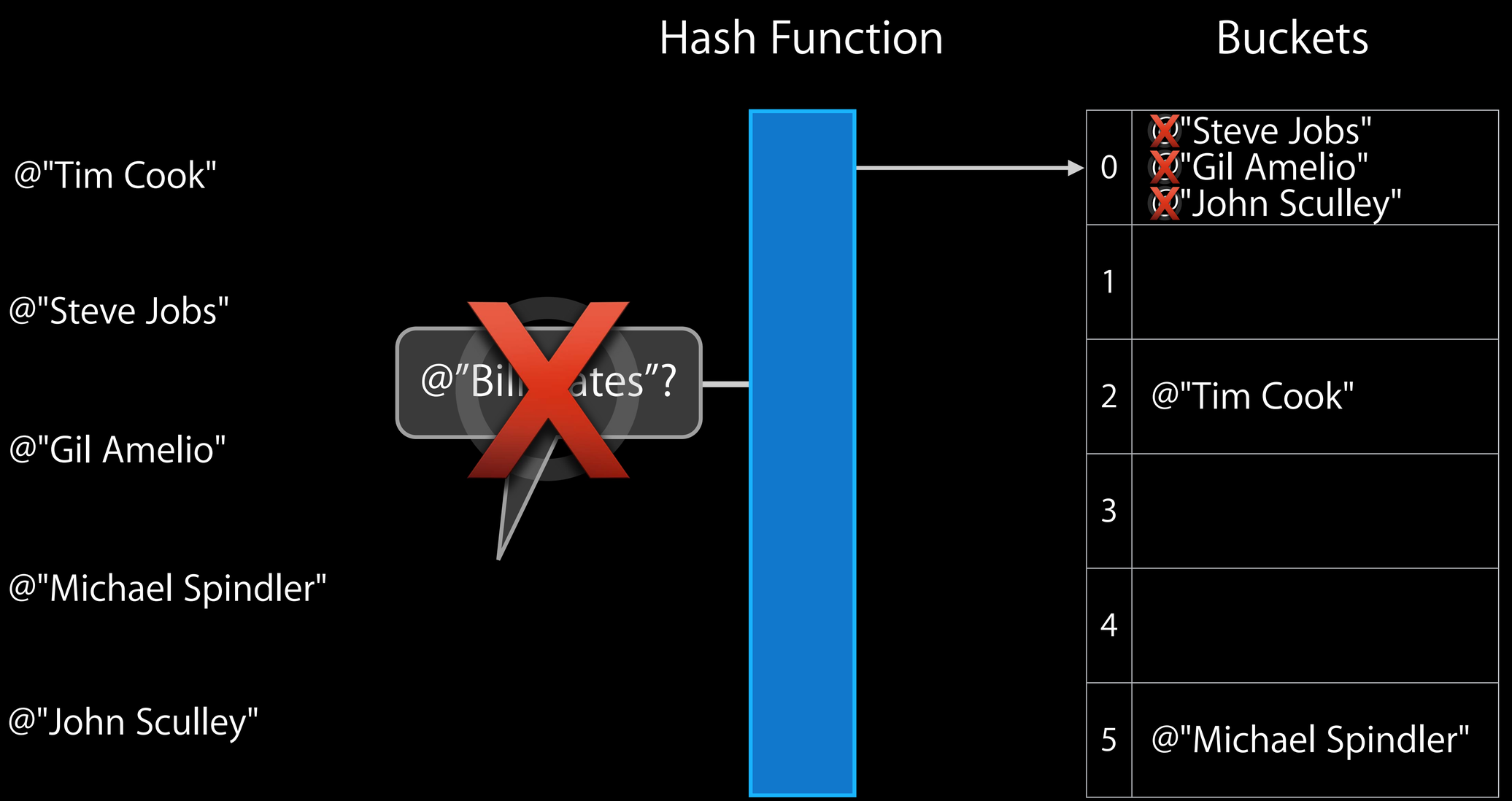
Buckets

0	@"Steve Jobs" @"Gil Amelio" @"John Sculley"
1	
2	@"Tim Cook"
3	
4	
5	@"Michael Spindler"

Hash-Based Organization



Hash-Based Organization



Hash-Based Organization

@"Tim Cook"

@"Steve Jobs"

@"Gil Amelio"

@"Michael Spindler"

@"John Sculley"

Hash Function

Buckets

0	@"Steve Jobs" @"Gil Amelio" @"John Sculley"
1	
2	@"Tim Cook"
3	
4	
5	@"Michael Spindler"

Hash-Based Organization

@"Tim Cook"

@"Steve Jobs"

@"Gil Amelio"

@"Michael Spindler"

@"John Sculley"

Hash Function

Buckets

0	@"John Sculley"
1	
2	@"Steve Jobs"
3	@"Michael Spindler"
4	@"Tim Cook"
5	@"Gil Amelio"

Hash-Based Organization

Define your identity

Hash-Based Organization

Define your identity

- NSObject's `-isEqual:` and `-hash` are functionally equivalent to:

```
- (BOOL) isEqual:(id)other
{
    return self == other;
}

- (NSUInteger) hash
{
    return (NSUInteger)self;
}
```

Hash-Based Organization

Define your identity

- NSObject's `-isEqual:` and `-hash` are functionally equivalent to:

```
- (BOOL) isEqual:(id)other
{
    return self == other;
}

- (NSUInteger) hash
{
    return (NSUInteger)self;
}
```

- Apple-provided subclasses override as needed

Hash-Based Organization

Define your identity

- NSObject's `-isEqual:` and `-hash` are functionally equivalent to:

```
- (BOOL) isEqual:(id)other
{
    return self == other;
}

- (NSUInteger) hash
{
    return (NSUInteger)self;
}
```

- Apple-provided subclasses override as needed
- Custom objects should override if pointer equality is not enough

Hash-Based Organization

Define your identity

- NSObject's `-isEqual:` and `-hash` are functionally equivalent to:

```
- (BOOL) isEqual:(id)other
{
    return self == other;
}

- (NSUInteger) hash
{
    return (NSUInteger)self;
}
```

- Apple-provided subclasses override as needed
- Custom objects should override if pointer equality is not enough
- http://developer.apple.com/documentation/General/Conceptual/DevPedia-CocoaCore/ObjectComparison.html

Hash-Based Organization

Rules of the road

Hash-Based Organization

Rules of the road

- If `-isEqual:` returns true, `-hash` must be equal for both objects
  - The same hash value does not imply equality

Hash-Based Organization

Rules of the road

- If `-isEqual:` returns true, `-hash` must be equal for both objects
  - The same hash value does not imply equality
- If you define `-isEqual:`, also define `-hash`

Hash-Based Organization

Rules of the road

- If `-isEqual:` returns true, `-hash` must be equal for both objects
  - The same hash value does not imply equality
- If you define `-isEqual:`, also define `-hash`
- A good hash should minimize collisions
  - Poor hashes will degrade performance

Hash-Based Organization

Rules of the road

- If `-isEqual:` returns true, `-hash` must be equal for both objects
  - The same hash value does not imply equality
- If you define `-isEqual:`, also define `-hash`
- A good hash should minimize collisions
  - Poor hashes will degrade performance
- Hash lookup + unpredictable hash = disaster
  - Option 1: Don't modify object in a collection
  - Option 2: Don't base hash on mutable state

Hash-Based Organization

Sample implementation

```
@interface WWDCNews : NSObject <NSCopying>
@property (readonly, copy) NSString *title;
@property (readonly, copy) NSDate *timestamp;
@end

@implementation WWDCNews
- (NSUInteger) hash {
    return [self.title hash];
}

- (BOOL) isEqual:(id)object {
    return ([object isKindOfClass:[WWDCNews class]]
         && [self.title isEqual:[object title]]
         && [self.timestamp isEqual:[object timestamp]]);
}
...
@end
```

Hash-Based Organization

Sample implementation

```
@interface WWDCNews : NSObject <NSCopying>
@property (readonly, copy) NSString *title;
@property (readonly, copy) NSDate *timestamp;
@end

@implementation WWDCNews
- (NSUInteger) hash {
    return [self.title hash];
}

- (BOOL) isEqual:(id)object {
    return ([object isKindOfClass:[WWDCNews class]]
         && [self.title isEqual:[object title]]
         && [self.timestamp isEqual:[object timestamp]]);
}
...
@end
```

Data Structures Performance

Example: Organizing Books

Choosing a strategy

Example: Organizing Books

Choosing a strategy

- Sort by topic, author, title, size, color, etc.

Example: Organizing Books

Choosing a strategy

- Sort by topic, author, title, size, color, etc.
- Each option makes some tasks easy, other tasks hard

Example: Organizing Books

Choosing a strategy

- Sort by topic, author, title, size, color, etc.
- Each option makes some tasks easy, other tasks hard
- Plan for scale when appropriate

Example: Organizing Books

Choosing a strategy

- Sort by topic, author, title, size, color, etc.
- Each option makes some tasks easy, other tasks hard
- Plan for scale when appropriate

Example: Organizing Books

Choosing a strategy

- Sort by topic, author, title, size, color, etc.
- Each option makes some tasks easy, other tasks hard
- Plan for scale when appropriate

Data Structures Performance

Choosing a strategy

Data Structures Performance

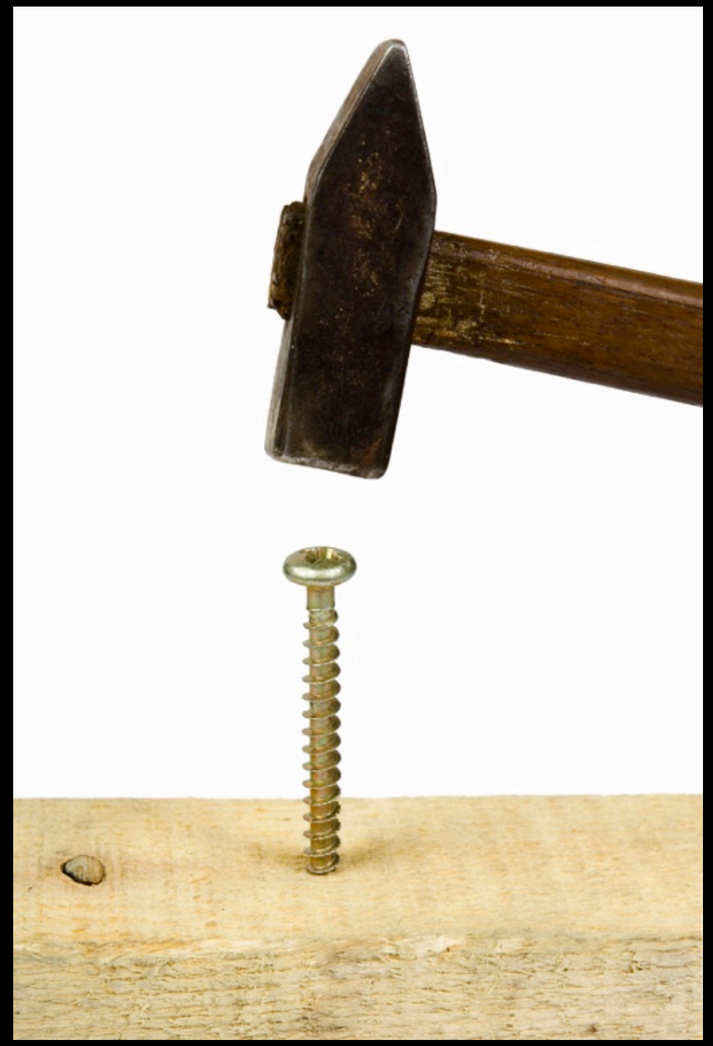
Choosing a strategy

- All data structures have tradeoffs
  - Use one that best fits your needs

Data Structures Performance

Choosing a strategy

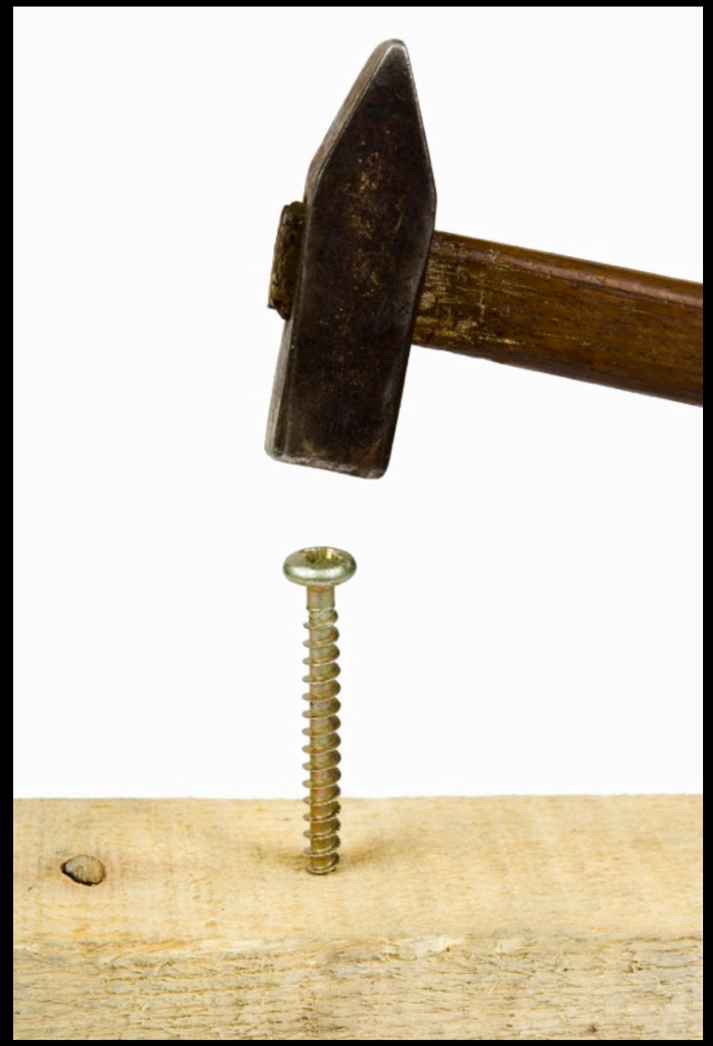
- All data structures have tradeoffs
  - Use one that best fits your needs
- A bad fit hurts performance
  - The wrong tool
  - The wrong approach



Data Structures Performance

Choosing a strategy

- All data structures have tradeoffs
  - Use one that best fits your needs
- A bad fit hurts performance
  - The wrong tool
  - The wrong approach
- Always prefer to use the built-in API
  - Extensively tested and optimized
  - Automatic future improvements



Data Structures

Context is key

Data Structures

Context is key

- Know what you need
  - Is order important?
  - Will I have duplicates?
  - Is mutability required?
  - What operations are most critical?
  - Where will data come from and go to?

Data Structures

Context is key

- Know what you need
  - Is order important?
  - Will I have duplicates?
  - Is mutability required?
  - What operations are most critical?
  - Where will data come from and go to?
- Know what to expect
  - Getting the count is O(1)
  - Enumerating all objects is O(n)
  - Other operations vary by collection

Mutability

Mutability

- Use it when you need it

Mutability

- Use it when you need it
- Immutable collections have benefits
  - Thread safety
  - Memory and speed optimizations

Mutability

- Use it when you need it
- Immutable collections have benefits
  - Thread safety
  - Memory and speed optimizations
- Make immutable copy afterward?

Mutability

- Use it when you need it
- Immutable collections have benefits
  - Thread safety
  - Memory and speed optimizations
- Make immutable copy afterward?
- Help us help you
  - You know how it will be used
  - Hints can enhance performance
  - Consider using `-initWithCapacity:`

Foundation Collection MVPs

- NSArray / NSMutableArray
- NSSet / NSMutableSet / NSCountedSet
- NSDictionary / NSMutableDictionary
- NSOrderedSet / NSMutableOrderedSet
- NSIndexSet / NSMutableIndexSet
- NSMapTable
- NSHashTable
- NSCache

NSArray / NSMutableArray

NSArray / NSMutableArray

- Ordered, indexed, allows duplicates

NSArray / NSMutableArray

- Ordered, indexed, allows duplicates
- Fast operations
  - Indexed access (e.g. `-objectAtIndex:`, `-firstObject`, `-lastObject`)
  - Add / remove at either end (e.g. `-addObject:`, `-removeLastObject:`)

NSArray / NSMutableArray

- Ordered, indexed, allows duplicates
- Fast operations
  - Indexed access (e.g. `-objectAtIndex:`, `-firstObject`, `-lastObject`)
  - Add / remove at either end (e.g. `-addObject:`, `-removeLastObject:`)
- Slower operations
  - Search (e.g. `-containsObject:`, `-indexOfObject*`, `-removeObject:`)
  - Add / remove, arbitrary index (e.g. `-insertObject:atIndex:`)

NSArray / NSMutableArray

- Ordered, indexed, allows duplicates
- Fast operations
  - Indexed access (e.g. `-objectAtIndex:`, `-firstObject`, `-lastObject`)
  - Add / remove at either end (e.g. `-addObject:`, `-removeLastObject:`)
- Slower operations
  - Search (e.g. `-containsObject:`, `-indexOfObject*`, `-removeObject:`)
  - Add / remove, arbitrary index (e.g. `-insertObject:atIndex:`)
- Specialty operations
  - Binary search (requires a sorted range of an array)
    - `-indexOfObject:inSortedRange:options:usingComparator:`

NSSet / NSMutableSet

NSSet / NSMutableSet

- Unordered, no duplicates, hash lookup

NSSet / NSMutableSet

- Unordered, no duplicates, hash lookup
- Add, remove, and search are fast
  - (e.g. `-addObject:`, `-removeObject:`, `-containsObject:`)

NSSet / NSMutableSet

- Unordered, no duplicates, hash lookup
- Add, remove, and search are fast
  - (e.g. `-addObject:`, `-removeObject:`, `-containsObject:`)
- Specialty operations
  - Set math: test overlap (e.g. `-intersectsSet:`, `-isSubsetOfSet:`)
  - Set math: modify (e.g. `-intersectSet:`, `-minusSet:`, `-unionSet:`)

NSSet / NSMutableSet

- Unordered, no duplicates, hash lookup
- Add, remove, and search are fast
  - (e.g. `-addObject:`, `-removeObject:`, `-containsObject:`)
- Specialty operations
  - Set math: test overlap (e.g. `-intersectsSet:`, `-isSubsetOfSet:`)
  - Set math: modify (e.g. `-intersectSet:`, `-minusSet:`, `-unionSet:`)
- Caveats
  - Converting array to set loses ordering and duplicates
  - Cannot be stored in a property list or JSON

NSCountedSet

NSCountedSet

- Unordered, no duplicates, hash lookup

NSCountedSet

- Unordered, no duplicates, hash lookup
- Subclass of NSMutableSet, same operations and caveats

NSCountedSet

- Unordered, no duplicates, hash lookup
- Subclass of NSMutableSet, same operations and caveats
- Tracks net insertion count for each object
  - Incremented on insert, decremented on remove
  - `-countForObject:` returns individual count
  - `-count` still returns number of objects, not sum of insertions

NSDictionary / NSMutableDictionary

NSDictionary / NSMutableDictionary

- Unordered, key-value entries, unique keys, hash lookup

NSDictionary / NSMutableDictionary

- Unordered, key-value entries, unique keys, hash lookup
- Add, remove, and search are fast
  - (e.g. `-objectForKey:`, `-setObject:forKey:`, `-removeObjectForKey:`)

NSDictionary / NSMutableDictionary

- Unordered, key-value entries, unique keys, hash lookup
- Add, remove, and search are fast
  - (e.g. `-objectForKey:`, `-setObject:forKey:`, `-removeObjectForKey:`)
- Specialty operations
  - Property list file I/O
  - `+sharedKeySetForKeys:`, `+dictionaryWithSharedKeySet:` (10.8, iOS 6)

NSDictionary / NSMutableDictionary

- Unordered, key-value entries, unique keys, hash lookup
- Add, remove, and search are fast
  - (e.g. `-objectForKey:`, `-setObject:forKey:`, `-removeObjectForKey:`)
- Specialty operations
  - Property list file I/O
  - `+sharedKeySetForKeys:`, `+dictionaryWithSharedKeySet:` (10.8, iOS 6)
- Caveats
  - Keys must conform to NSCopying (“copy in”)
  - NEVER mutate an object that is a dictionary key

NSOrderedSet / NSMutableOrderedSet

NSOrderedSet / NSMutableOrderedSet

- Ordered, no duplicates, index / hash lookup (10.7, iOS 5)

NSOrderedSet / NSMutableOrderedSet

- Ordered, no duplicates, index / hash lookup (10.7, iOS 5)
- Effectively a cross of NSArray and NSSet
  - Not a subclass of either one
  - Call `-array` or `-set` for immutable, live-updating representations

NSOrderedSet / NSMutableOrderedSet

- Ordered, no duplicates, index / hash lookup (10.7, iOS 5)
- Effectively a cross of NSArray and NSSet
  - Not a subclass of either one
  - Call `-array` or `-set` for immutable, live-updating representations
- Caveats
  - Increased memory usage
  - Property list support requires conversions

NSIndexSet / NSMutableIndexSet

NSIndexSet / NSMutableIndexSet

- Collection of unique NSUInteger values

NSIndexSet / NSMutableIndexSet

- Collection of unique NSUInteger values
- Reference a subset of objects in NSArray
  - Avoid memory overhead of array copies

NSIndexSet / NSMutableIndexSet

- Collection of unique NSUInteger values
- Reference a subset of objects in NSArray
  - Avoid memory overhead of array copies
- Efficient storage and coalescing

NSIndexSet / NSMutableIndexSet

- Collection of unique NSUInteger values
- Reference a subset of objects in NSArray
  - Avoid memory overhead of array copies
- Efficient storage and coalescing
- Set arithmetic (intersect, subset, difference)

NSIndexSet / NSMutableIndexSet

- Collection of unique NSUInteger values
- Reference a subset of objects in NSArray
  - Avoid memory overhead of array copies
- Efficient storage and coalescing
- Set arithmetic (intersect, subset, difference)
- Caveats
  - Use caution with indexes for mutable arrays

NSMapTable / NSHashTable

NSMapTable / NSHashTable

- Similar to NSMutableDictionary / NSMutableSet

NSMapTable / NSHashTable

- Similar to NSMutableDictionary / NSMutableSet
- More flexibility via NSMapTableOptions / NSHashTableOptions
  - May use pointer identity for equality and hashing
  - May contain any pointer (not just objects)
  - Optional weak references to keys and/or values (zeroing under ARC)
  - Optional copy on insert

NSMapTable / NSHashTable

- Similar to NSMutableDictionary / NSMutableSet
- More flexibility via NSMapTableOptions / NSHashTableOptions
  - May use pointer identity for equality and hashing
  - May contain any pointer (not just objects)
  - Optional weak references to keys and/or values (zeroing under ARC)
  - Optional copy on insert
- Caveats
  - Can’t convert non-object contents to dictionary/set
  - Beware of premature optimization!

NSCache

NSCache

- Similar to NSMutableDictionary
- Thread-safe
- Doesn’t copy keys
- Auto-removal under memory pressure
- Ideal for objects that can be regenerated on demand

NSCache

- Similar to NSMutableDictionary
- Thread-safe
- Doesn’t copy keys
- Auto-removal under memory pressure
- Ideal for objects that can be regenerated on demand

Brief Aside: Property Lists

Brief Aside: Property Lists

- Files that store simple hierarchies of data
  - XML and binary formats

Brief Aside: Property Lists

- Files that store simple hierarchies of data
  - XML and binary formats
- Supports "property list" objects
  - NSArray, NSData, NSDate, NSDictionary, NSNumber, NSString
  - Others must adopt NSCoding and be archived

Brief Aside: Property Lists

- Files that store simple hierarchies of data
  - XML and binary formats
- Supports "property list" objects
  - NSArray, NSData, NSDate, NSDictionary, NSNumber, NSString
  - Others must adopt NSCoding and be archived
- Mutability is not preserved on read/write

Brief Aside: Property Lists

- Files that store simple hierarchies of data
  - XML and binary formats
- Supports "property list" objects
  - NSArray, NSData, NSDate, NSDictionary, NSNumber, NSString
  - Others must adopt NSCoding and be archived
- Mutability is not preserved on read/write
- Inefficient for lots of binary data, large files

Brief Aside: Property Lists

- Files that store simple hierarchies of data
  - XML and binary formats
- Supports "property list" objects
  - NSArray, NSData, NSDate, NSDictionary, NSNumber, NSString
  - Others must adopt NSCoding and be archived
- Mutability is not preserved on read/write
- Inefficient for lots of binary data, large files
- NSUserDefaults

Brief Aside: Property Lists

- Files that store simple hierarchies of data
  - XML and binary formats
- Supports "property list" objects
  - NSArray, NSData, NSDate, NSDictionary, NSNumber, NSString
  - Others must adopt NSCoding and be archived
- Mutability is not preserved on read/write
- Inefficient for lots of binary data, large files
- NSUserDefaults
- NSPropertyListSerialization

Brief Aside: Property Lists

- Files that store simple hierarchies of data
  - XML and binary formats
- Supports "property list" objects
  - NSArray, NSData, NSDate, NSDictionary, NSNumber, NSString
  - Others must adopt NSCoding and be archived
- Mutability is not preserved on read/write
- Inefficient for lots of binary data, large files
- NSUserDefaults
- NSPropertyListSerialization
- NSKeyedArchiver / NSKeyedUnarchiver

Brief Aside: JSON

Brief Aside: JSON

- JavaScript Object Notation (http://json.org)

Brief Aside: JSON

- JavaScript Object Notation (http://json.org)
- Lightweight data format
  - Commonly used with web services

Brief Aside: JSON

- JavaScript Object Notation (http://json.org)
- Lightweight data format
  - Commonly used with web services
- Works with a few Foundation classes
  - NSArray, NSDictionary, NSNull, NSNumber, NSString
  - Some restrictions on hierarchy and values

Brief Aside: JSON

- JavaScript Object Notation (http://json.org)
- Lightweight data format
  - Commonly used with web services
- Works with a few Foundation classes
  - NSArray, NSDictionary, NSNull, NSNumber, NSString
  - Some restrictions on hierarchy and values
- NSJSONSerialization (10.7 / iOS 5.0)
  - Reading, writing, object validation
  - Optional mutability on read
  - Fast, built-in, dynamically linked

Real-World Applications

WWDC App — Refreshing Sessions

WWDC App — Refreshing Sessions

- Details are stored in Core Data

WWDC App — Refreshing Sessions

- Details are stored in Core Data
- Periodically fetch updates

WWDC App — Refreshing Sessions

- Details are stored in Core Data
- Periodically fetch updates
- Performance issues
  - Great speed at first
  - More sessions cause lag
  - Performance gets bad quickly

WWDC App — Refreshing Sessions

- Details are stored in Core Data
- Periodically fetch updates
- Performance issues
  - Great speed at first
  - More sessions cause lag
  - Performance gets bad quickly
- Profiling reveals non-linear growth

WWDC App — Refreshing Sessions

Approach 1

WWDC App — Refreshing Sessions

Approach 1

```
NSArray *sessionsToImport = ...; // Fetched from server
NSManagedObjectContext *context = ...;

for (WWDCSession *session in sessionsToImport) {
    NSFetchRequest *r = [NSFetchRequest fetchRequestWithEntityName:@"WWDCSession"];
    r.predicate = [NSPredicate predicateWithFormat:@"sessionID = %@", session.sessionID];
    NSArray *results = [context executeFetchRequest:r error:nil];

    WWDCSession *existingSession = [results firstObject];
    if (existingSession != nil) {
        // Merge into existingSession
    } else {
        // Insert into context
    }
}
```

WWDC App — Refreshing Sessions

Approach 1

```
NSArray *sessionsToImport = ...; // Fetched from server
NSManagedObjectContext *context = ...;

for (WWDCSession *session in sessionsToImport) {
    NSFetchRequest *r = [NSFetchRequest fetchRequestWithEntityName:@"WWDCSession"];
    r.predicate = [NSPredicate predicateWithFormat:@"sessionID = %@", session.sessionID];
    NSArray *results = [context executeFetchRequest:r error:nil];

    WWDCSession *existingSession = [results firstObject];
    if (existingSession != nil) {
        // Merge into existingSession
    } else {
        // Insert into context
    }
}
```

WWDC App — Refreshing Sessions

Approach 2

```
NSArray *sessionsToImport = ...; // Fetched from server
NSManagedObjectContext *context = ...;

NSArray *sessionIDs = [sessionsToImport valueForKey:@"sessionID"];
NSFetchRequest *r = [NSFetchRequest fetchRequestWithEntityName:@"WWDCSession"];
r.predicate = [NSPredicate predicateWithFormat:@"sessionID in %@", sessionIDs];
NSArray *results = [context executeFetchRequest:r error:nil];

for (WWDCSession *session in sessionsToImport) {
    NSInteger existingSessionIndex = [results indexOfObject:session];
    if (existingSessionIndex != NSNotFound) {
        WWDCSession *existingSession = results[existingSessionIndex];
        // Merge into existingSession
    } else {
        // Insert into context
    }
}
```

WWDC App — Refreshing Sessions

Approach 2

```
NSArray *sessionsToImport = ...; // Fetched from server
NSManagedObjectContext *context = ...;

NSArray *sessionIDs = [sessionsToImport valueForKey:@"sessionID"];
NSFetchRequest *r = [NSFetchRequest fetchRequestWithEntityName:@"WWDCSession"];
r.predicate = [NSPredicate predicateWithFormat:@"sessionID in %@", sessionIDs];
NSArray *results = [context executeFetchRequest:r error:nil];

for (WWDCSession *session in sessionsToImport) {
    NSInteger existingSessionIndex = [results indexOfObject:session];
    if (existingSessionIndex != NSNotFound) {
        WWDCSession *existingSession = results[existingSessionIndex];
        // Merge into existingSession
    } else {
        // Insert into context
    }
}
```

WWDC App — Refreshing Sessions

Approach 2

```
NSArray *sessionsToImport = ...; // Fetched from server
NSManagedObjectContext *context = ...;

NSArray *sessionIDs = [sessionsToImport valueForKey:@"sessionID"];
NSFetchRequest *r = [NSFetchRequest fetchRequestWithEntityName:@"WWDCSession"];
r.predicate = [NSPredicate predicateWithFormat:@"sessionID in %@", sessionIDs];
NSArray *results = [context executeFetchRequest:r error:nil];

for (WWDCSession *session in sessionsToImport) {
    NSInteger existingSessionIndex = [results indexOfObject:session];
    if (existingSessionIndex != NSNotFound) {
        WWDCSession *existingSession = results[existingSessionIndex];
        // Merge into existingSession
    } else {
        // Insert into context
    }
}
```

WWDC App — Refreshing Sessions

Approach 3

```
NSArray *sessionsToImport = ...; // Fetched from server
NSManagedObjectContext *context = ...;

NSArray *sessionIDs = [sessionsToImport valueForKey:@"sessionID"];
NSFetchRequest *r = [NSFetchRequest fetchRequestWithEntityName:@"WWDCSession"];
r.predicate = [NSPredicate predicateWithFormat:@"sessionID in %@", sessionIDs];
NSArray *results = [context executeFetchRequest:r error:nil];
NSDictionary *sessionsKeyedByID = [NSDictionary dictionaryWithObjects:results forKeys:
[results valueForKey:@"sessionID"]];

for (WWDCSession *session in sessionsToImport) {
    WWDCSession *existingSession = sessionsKeyedByID[session.sessionID];
    if (existingSession != nil) {
        // Merge into existingSession
    } else {
        // Insert into context
    }
}
```

WWDC App — Refreshing Sessions

Approach 3

```
NSArray *sessionsToImport = ...; // Fetched from server
NSManagedObjectContext *context = ...;

NSArray *sessionIDs = [sessionsToImport valueForKey:@"sessionID"];
NSFetchRequest *r = [NSFetchRequest fetchRequestWithEntityName:@"WWDCSession"];
r.predicate = [NSPredicate predicateWithFormat:@"sessionID in %@", sessionIDs];
NSArray *results = [context executeFetchRequest:r error:nil];
NSDictionary *sessionsKeyedByID = [NSDictionary dictionaryWithObjects:results forKeys:
[results valueForKey:@"sessionID"]];

for (WWDCSession *session in sessionsToImport) {
    WWDCSession *existingSession = sessionsKeyedByID[session.sessionID];
    if (existingSession != nil) {
        // Merge into existingSession
    } else {
        // Insert into context
    }
}
```

WWDC App — Refreshing Sessions

Approach 3

```
NSArray *sessionsToImport = ...; // Fetched from server
NSManagedObjectContext *context = ...;

NSArray *sessionIDs = [sessionsToImport valueForKey:@"sessionID"];
NSFetchRequest *r = [NSFetchRequest fetchRequestWithEntityName:@"WWDCSession"];
r.predicate = [NSPredicate predicateWithFormat:@"sessionID in %@", sessionIDs];
NSArray *results = [context executeFetchRequest:r error:nil];
NSDictionary *sessionsKeyedByID = [NSDictionary dictionaryWithObjects:results forKeys:
[results valueForKey:@"sessionID"]];

for (WWDCSession *session in sessionsToImport) {
    WWDCSession *existingSession = sessionsKeyedByID[session.sessionID];
    if (existingSession != nil) {
        // Merge into existingSession
    } else {
        // Insert into context
    }
}
```

Eliminating Extra Work

Eliminating Extra Work

- Minimize redundancy, especially "expensive" code
  - Particularly within loops

Eliminating Extra Work

- Minimize redundancy, especially "expensive" code
  - Particularly within loops
- Take advantage of faster lookup when possible
  - Dictionaries and sets

Eliminating Extra Work

- Minimize redundancy, especially "expensive" code
  - Particularly within loops
- Take advantage of faster lookup when possible
  - Dictionaries and sets
- Use mutable collections (and strings) when it makes sense

Eliminating Extra Work

- Minimize redundancy, especially "expensive" code
  - Particularly within loops
- Take advantage of faster lookup when possible
  - Dictionaries and sets
- Use mutable collections (and strings) when it makes sense
- Streamline how you access your data

Eliminating Extra Work

- Minimize redundancy, especially "expensive" code
  - Particularly within loops
- Take advantage of faster lookup when possible
  - Dictionaries and sets
- Use mutable collections (and strings) when it makes sense
- Streamline how you access your data
- Don't reinvent the wheel
  - e.g. `-[NSArray componentsJoinedByString:]`

Eliminating Extra Work

Example

Eliminating Extra Work

Example

```
- (NSArray*) doSomethingWithArray:(NSArray*)array
{
    NSArray *newArray = [NSArray array];
    for (id object in array) {
        id newObject = [self doSomethingWithObject:object];
        if (newObject != nil) {
            newArray = [newArray arrayByAddingObject:newObject];
        }
    }
    return newArray;
}
```

Eliminating Extra Work

Example

```
- (NSArray*) doSomethingWithArray:(NSArray*)array
{
    NSArray *newArray = [NSArray array];
    for (id object in array) {
        id newObject = [self doSomethingWithObject:object];
        if (newObject != nil) {
            newArray = [newArray arrayByAddingObject:newObject];
        }
    }
    return newArray;
}
```

Eliminating Extra Work

Example

```
- (NSArray*) doSomethingWithArray:(NSArray*)array
{
    NSMutableArray *newArray = [NSMutableArray array];
    for (id object in array) {
        id newObject = [self doSomethingWithObject:object];
        if (newObject != nil) {
            [newArray addObject:newObject];
        }
    }
    return [newArray copy]; // Copy is immutable
}
```

Eliminating Extra Work

Example

```
- (NSArray*) doSomethingWithArray:(NSArray*)array
{
    NSMutableArray *newArray = [NSMutableArray array];
    for (id object in array) {
        id newObject = [self doSomethingWithObject:object];
        if (newObject != nil) {
            [newArray addObject:newObject];
        }
    }
    return [newArray copy]; // Copy is immutable
}
```

- Similarly, appending to an NSMutableString

Don’t leave performance on the table

More Information

Dave DeLong
App Frameworks and Developer Tools Evangelist
delong@apple.com

Apple Developer Forums
http://devforums.apple.com

Collections Programming Topics
http://developer.apple.com/documentation/Cocoa/Conceptual/Collections/

Property List Programming Guide
http://developer.apple.com/documentation/Cocoa/Conceptual/PropertyLists/

Archives and Serializations Programming Guide
http://developer.apple.com/documentation/Cocoa/Conceptual/Archiving/

Related Sessions

Building Efficient OS X Apps	Nob Hill Tuesday 4:30PM
Hidden Gems in Cocoa and Cocoa Touch	Nob Hill Friday 10:15AM

Summary

Summary

- Complexity kills large-scale performance

Summary

- Complexity kills large-scale performance
- Know how much work your code does

Summary

- Complexity kills large-scale performance
- Know how much work your code does
- Avoid redundancy, strive for efficiency

Summary

- Complexity kills large-scale performance
- Know how much work your code does
- Avoid redundancy, strive for efficiency
- Focus on biggest performance wins
  - Profile and analyze, don’t assume

Summary

- Complexity kills large-scale performance
- Know how much work your code does
- Avoid redundancy, strive for efficiency
- Focus on biggest performance wins
  - Profile and analyze, don’t assume
- Prefer built-in collections and API

Summary

- Complexity kills large-scale performance
- Know how much work your code does
- Avoid redundancy, strive for efficiency
- Focus on biggest performance wins
  - Profile and analyze, don’t assume
- Prefer built-in collections and API
- Design according to your needs

Summary

- Complexity kills large-scale performance
- Know how much work your code does
- Avoid redundancy, strive for efficiency
- Focus on biggest performance wins
  - Profile and analyze, don’t assume
- Prefer built-in collections and API
- Design according to your needs
- Think about performance early

WWDC2013