Algorithms and Complexity, Part 2

Kylie A. Bemis

Northeastern University
Khoury College of Computer Sciences

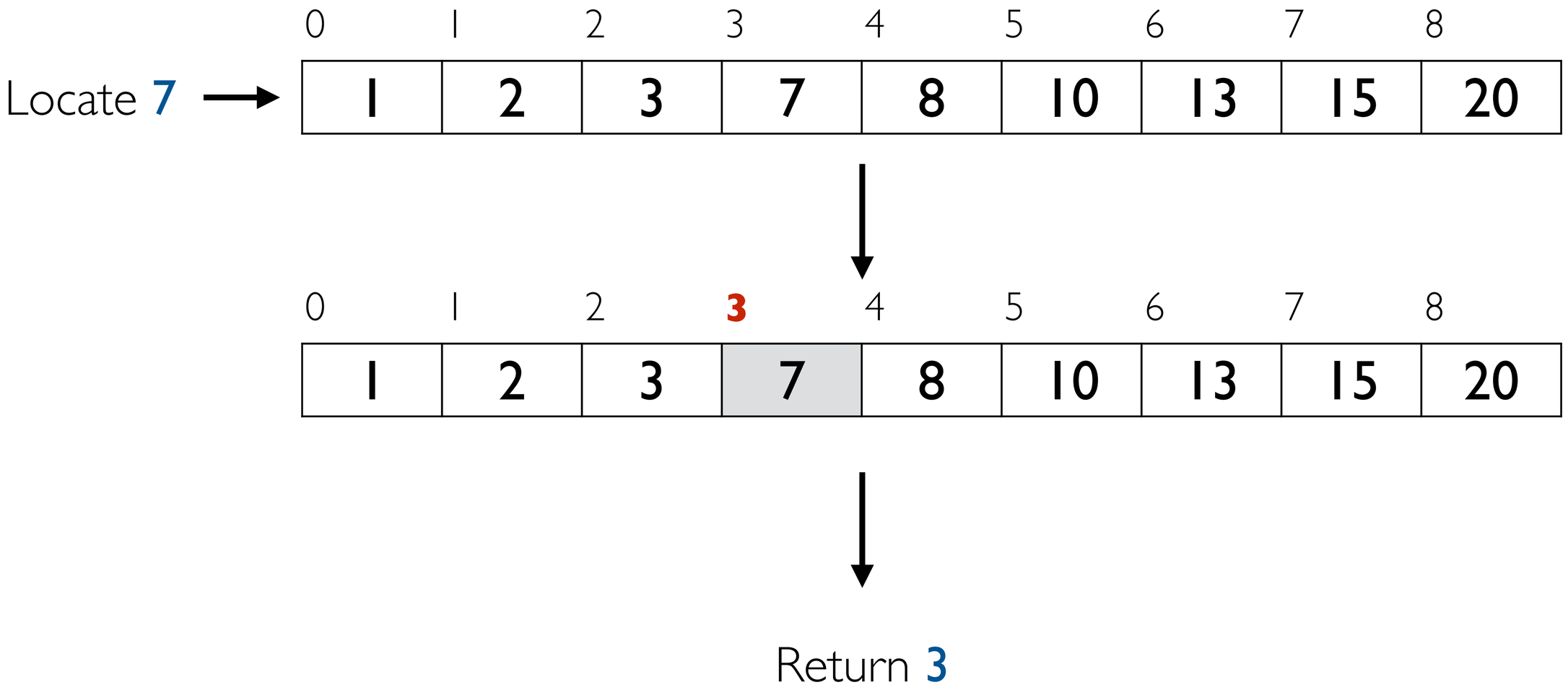
Northeastern University

Goals for today

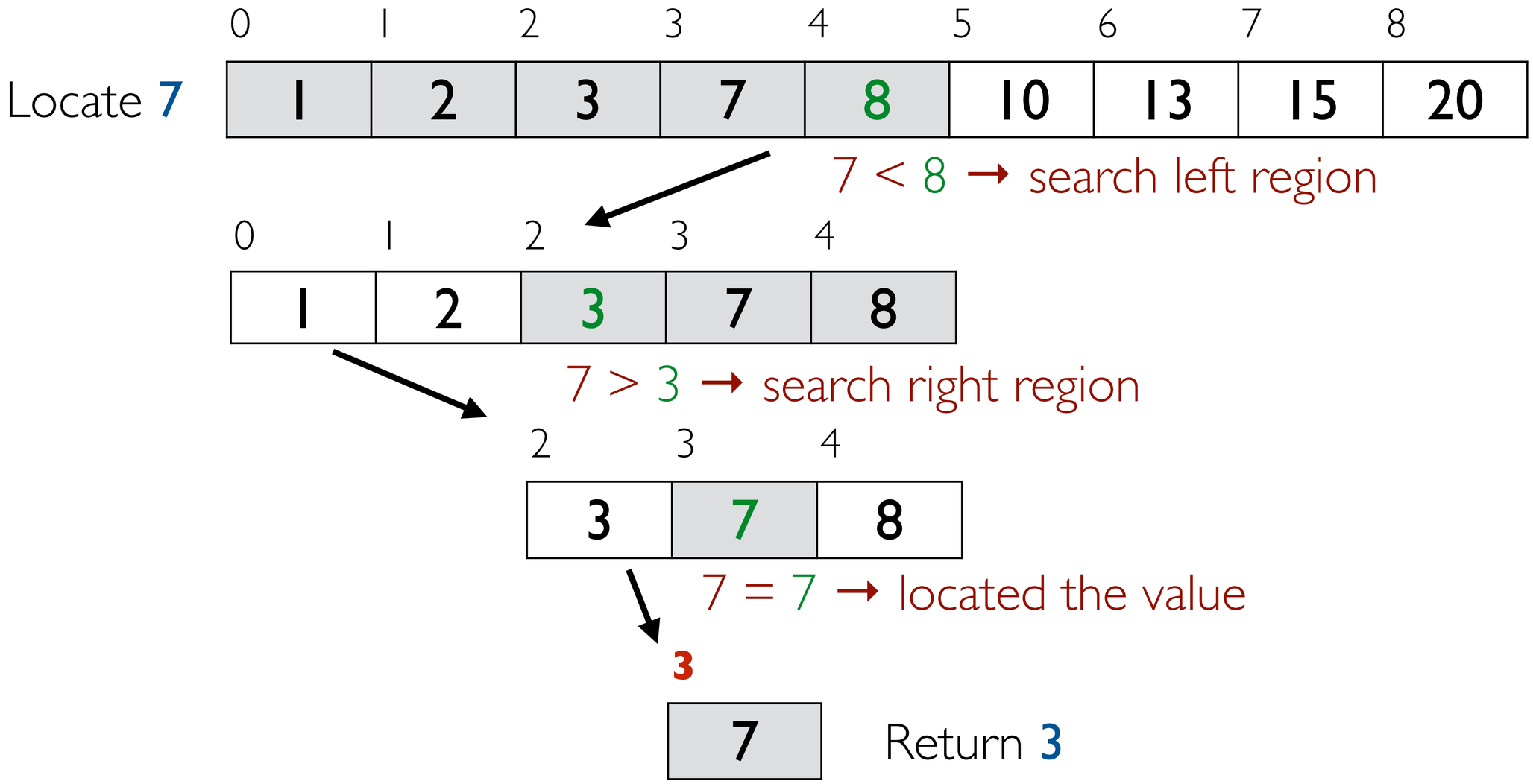
- Algorithmic complexity
- Amortized time
- Efficiency of common operations

ALGORITHMIC COMPLEXITY

Search for a value in a sorted array



Divide the problem into sub-problems



Binary search

- Split sorted array into two subarrays
  - Examine middle element of the array
  - Reduce search to left or right subarray

Also called
bisection search &
half-interval search

```
BinarySearch(x, key):

L, R = 0, length of x
while L ≤ R do:
    i = midpoint of x[L:R]
    if key < x[i]:
        R = i
    else if key > x[i]:
        L = i
    else
        return i
```

Binary search vs. linear search

"Which is faster?"

- *Measure* the running time of the algorithms
- *Analyze* the complexity of the algorithms

Timing an algorithm

- Measure running time of an algorithm:
  - Start clock
  - Run algorithm
  - End clock
- Separate algorithm from *setup time*
  - Loading necessary modules
  - Creating or copying data
- *Repeat* many times (due to variation)

Timing in Python

```
In : import timeit

In : timeit.timeit('[None] * 100', number=1000000)
Out: 0.3574

In : timeit.timeit('[None for i in range(100)]', number=1000000)
Out: 2.8074
```

Timing in Python

Import benchmarking module

```
In : import timeit

In : timeit.timeit('[None] * 100', number=1000000)
Out: 0.3574

In : timeit.timeit('[None for i in range(100)]', number=1000000)
Out: 2.8074
```

Number of times to run the code snippet

Statement to run (as a string or callable)

Limitations of timing

- Depends on implementation of algorithm
- Depends on computer hardware
- Depends on input sizes
- Depends on input configurations

How to control for these variables?

Analyzing an algorithm

- *Mathematically* compare performance of algorithms
- Understand *why* an algorithm is fast/slow
- Understand *how* an algorithm behaves
  - Evaluate an algorithm as a function of input size
  - Evaluate how an algorithm *scales* to larger datasets
- Complexity is an *abstraction* of algorithmic behavior

Complexity cases

- **Best case**: *fastest time with optimal input*
  - Rarely considered
  - *E.g.*, value to find is first one considered
- **Average case**: *typical time to complete*
  - Considered for some algorithms
  - Difficult to compute, requires assumptions about possible inputs
- **Worst case**: *slowest time with pessimistic input*
  - Most frequently considered
  - *E.g.*, list to sort is sorted in reverse order

Counting computer operations

- Measure time complexity of an algorithm
- Assume primitive operations take constant time:
  - Arithmetic operations
  - Comparisons and boolean operations
  - Variable assignments
  - Accessing and returning objects in memory
- Count total number of operations

Complexity of linear search

```
def lsearch(x, value):
    """
    Linear search of list x for value
    param x: The list to search
    param value: The value to find
    returns: The index of value in x, or None if not found
    """
    for i, xi in enumerate(x):
        if value == xi:
            return i
    return None
```

Cost	*Times*
c_1	k
c_2	k
c_3	1
c_3	1

Running time of linear search = $(c_1 + c_2)k + c_3$

Complexity of linear search

- Depends on location of value (k)
- What is k?
  - **Best** case: **k = 0** → $T = c_1 + c_2 + c_3$
  - **Average** case: **k = n / 2** → $T = (c_1 + c_2)\ n / 2 + c_3$
  - **Worst** case: **k = n** → $T = (c_1 + c_2)\ n + c_3$
- Where **n** is the number of items in **x**

Limitations of exact counting

- Depends on implementation
- *Difficult to measure* cost of some operations
- *Difficult to compare* with so many factors
- Exact number of operations *doesn't matter*

Asymptotic efficiency

- How does an algorithm behave *asymptotically*?
  - As a function of input size ***n***
  - As the size of the input *grows larger*
- Does not depend on implementation or computer
- Ignores constant factors and small datasets
- Allows easier comparison of algorithms

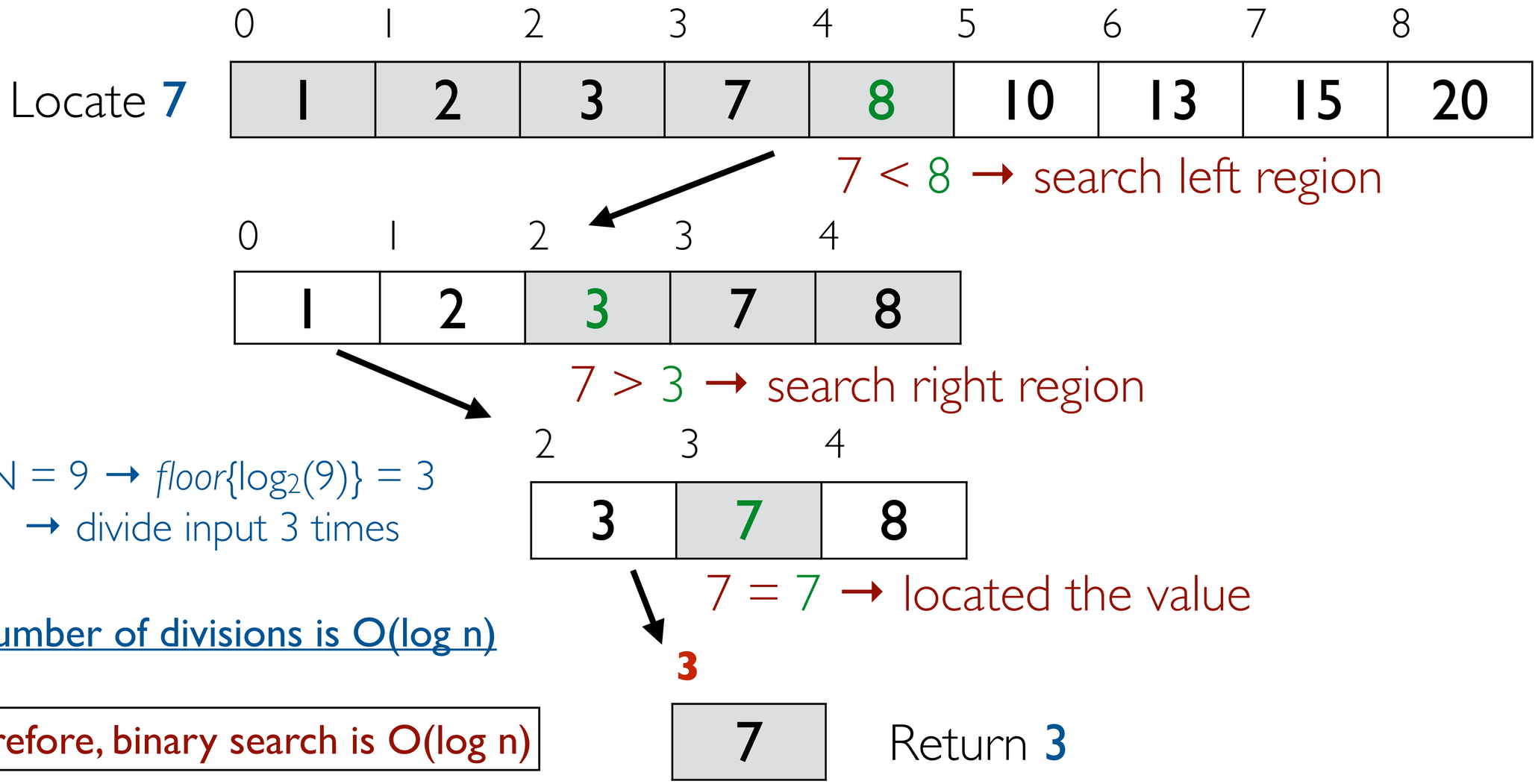
Complexity of linear search

- Find a specific item in a list **x**
  - **Best** case: *O(1)*
  - **Average** case: *O(n)*
  - **Worst** case: *O(n)*
- Where **n** is the number of items in **x**

Complexity of binary search

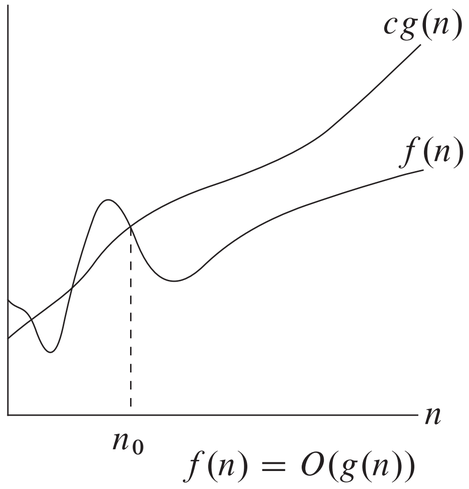
- Find a specific item in a list **x**
  - **Best** case: *O(1)*
  - **Average** case: *O(log n)*
  - **Worst** case: *O(log n)*
- Where **n** is the number of items in **x**

Logarithmic efficiency of binary search



O-notation

- *Upper bound* on amount of time for an algorithm to complete for input size ***n***
- Asymptotic *upper bound* described as *O(g(n))*
- A function *f(n)* is *O(g(n))* if
  - For some constant *c* and all values of $n \geq$ some value n_0
  - $f(n) \leq c \times g(n)$

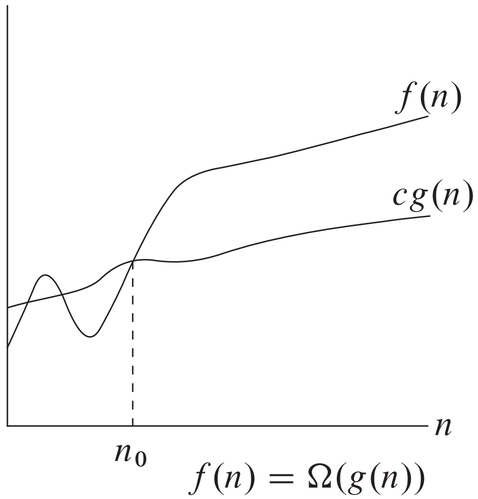


E.g., 4n + 3 → O(n)

E.g., 4n + 3 → O(n²)

Ω-notation

- *Lower bound* on amount of time for an algorithm to complete for input size ***n***
- Asymptotic *lower bound* described as **Ω**(*g*(*n*))
- A function *f*(*n*) is **Ω**(*g*(*n*)) if
  - For some constant *c* and all values of *n* ≥ some value n_0
  - *c* × *g*(*n*) ≤ *f*(*n*)

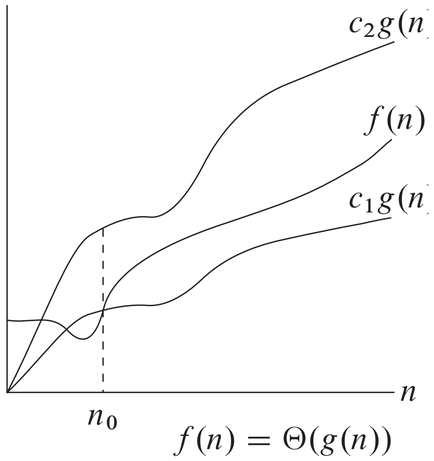


E.g., 4*n* + 3 → **Ω**(*n*)

E.g., 4*n* + 3 → **Ω**(*1*)

Θ-notation

- *Tight bound* on amount of time for an algorithm to complete for input size ***n***
- Asymptotic *tight bound* described as $\Theta(g(n))$
- A function $f(n)$ is $\Theta(g(n))$ if



- For some constants c_1, c_2 and all values of $n \geq$ some value n_0
- $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$

E.g., $4n + 3 \rightarrow \Theta(n)$

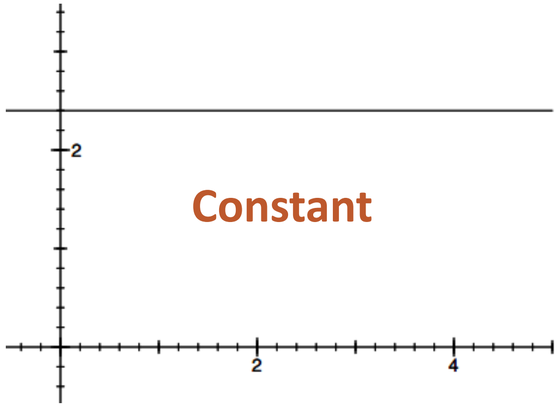
Asymptotic growth

- Drop constants and multiplicative factors
- Keep only *dominant terms*:
  - $n^2 + n + 100 \rightarrow n^2$
  - $100n^2 + 100n + 5 \rightarrow n^2$
  - $\log n + n + 100 \rightarrow n$
  - $\log n + 100 \rightarrow \log n$
  - $n \log n + 100n \rightarrow n \log n$

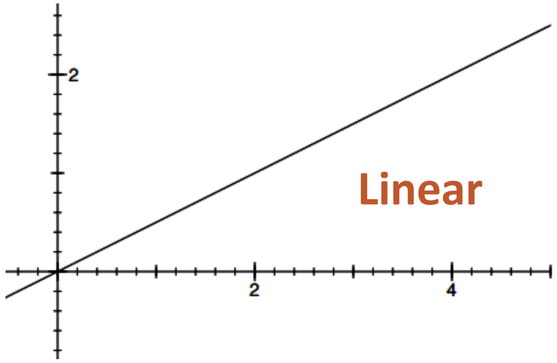
$O()$ vs. $\Omega()$ vs. $\Theta()$

- Consider $f(n) = 7n^3 + 5n + 100$
  - $O(n^{100})$ → $f(n)$ grows asymptotically no faster than n^{100}
  - $O(n^3)$ → $f(n)$ grows asymptotically no faster than n^3
  - $\Theta(n^3)$ → $f(n)$ grows asymptotically as fast as n^3
  - $\Omega(n^2)$ → $f(n)$ grows asymptotically at least as fast or faster than n^2
- Typically concerned with $O()$ behavior
  - May be used interchangeably with $\Theta()$ in some contexts

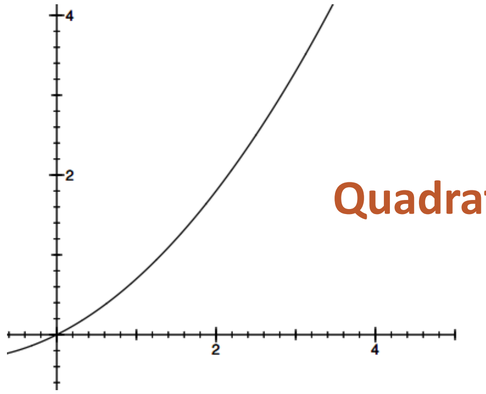
Growth rates of functions



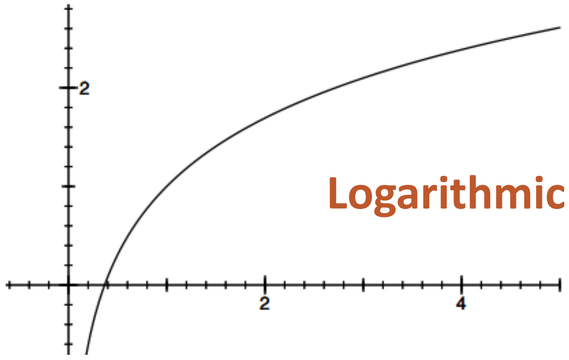
Constant



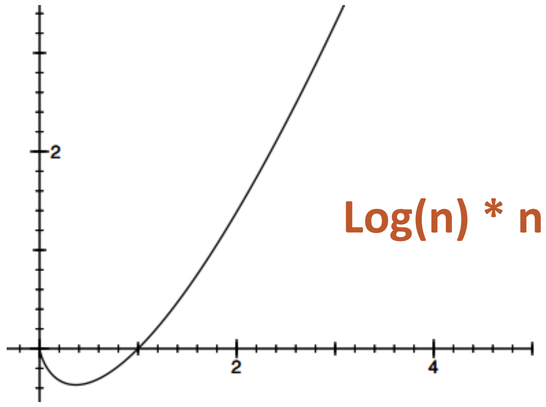
Linear



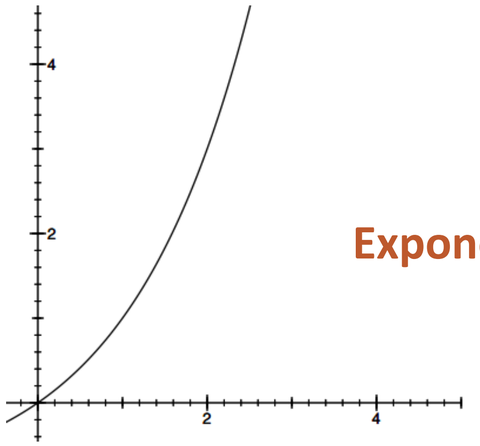
Quadratic



Logarithmic



Log(n) * n

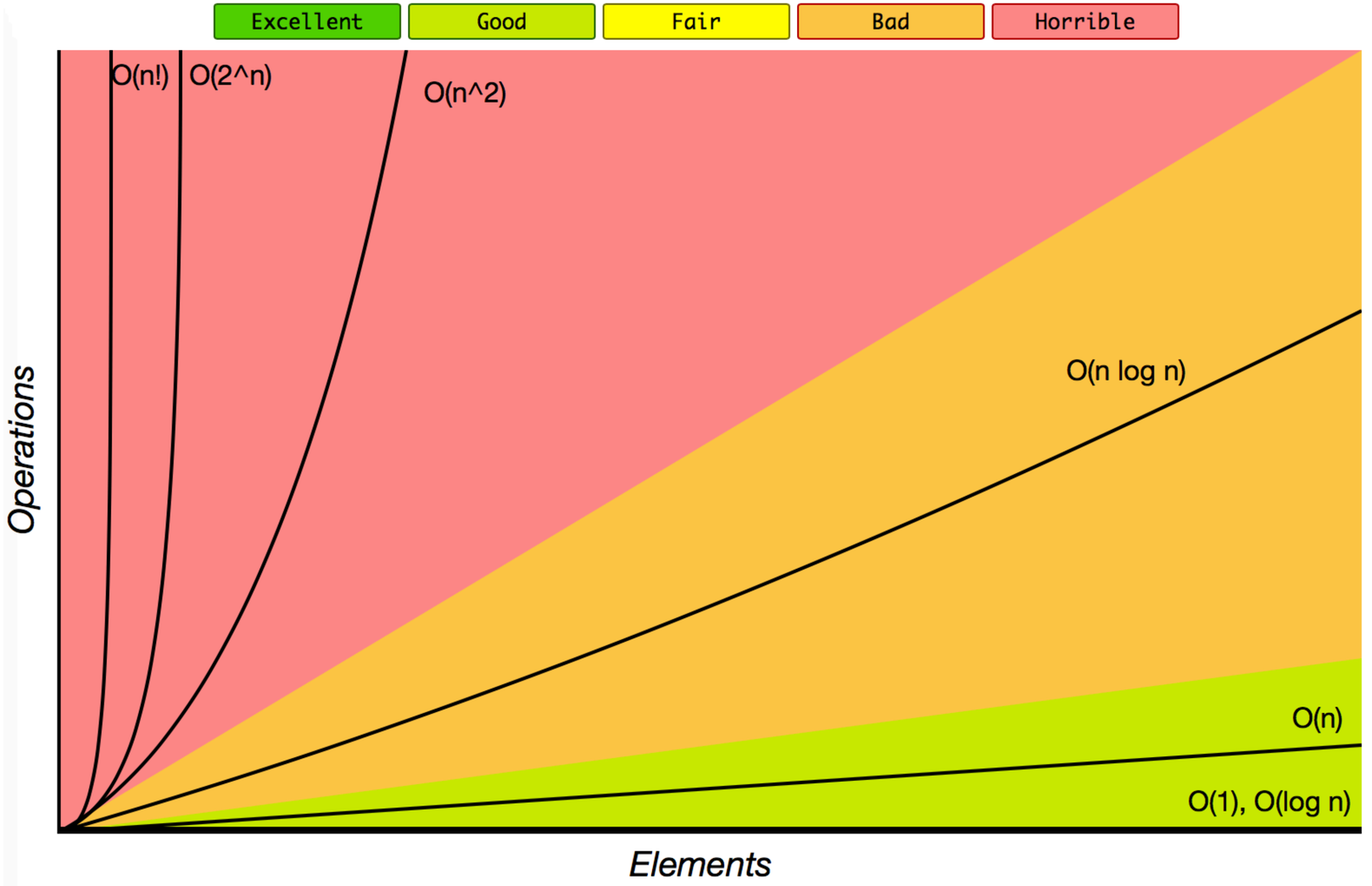


Exponential

Common complexities

$O(1)$	Constant time (not affected by input size)
$O(n)$	Time increases linearly with input
$O(n^2)$	Time increases quadratically with input
$O(\log n)$	Time increases logarithmically (divide & conquer)
$O(n \log n)$	Time increases log-linearly (divide & conquer)
$O(x^n)$	Time increases by factor of x for each new input
$O(n!)$	Time increases by a larger factor for each new input

Visualizing complexities



https://learntocodetogether.com/big-o-cheat-sheet-for-common-data-structures-and-algorithms/

Common patterns of complexity

- Constant time
  - Operations that do not depend on input size
- Linear and polynomial time
  - Simple loops *dependent on input size* are linear: $O(n)$
  - *Nested* loops are polynomial: $O(n) \rightarrow O(n^2) \rightarrow O(n^3)$
- Logarithmic time
  - Algorithms that *repeatedly divide input*
  - Don't forget to consider sub-operations!

Complexity of selection sort

```
def ssort(x):
    """
    Sorts a list of numbers in-place using selection sort
    param x: The list to sort (in place)
    returns: None
    """
    for i in range(len(x)):
        imin = i
        # find minimum in sublist x[i:]
        for j in range(i, len(x)):
            if x[j] < x[imin]:
                imin = j
        swap(x, i, imin)
```

Outer loop grows as O(n)

Inner loop grows as O(n)

Therefore, selection sort is $O(n \times n) = O(n^2)$

Complexity of insertion sort

```
def isort(x):
    """
    Sorts a list of numbers in-place using insertion sort
    param x: The list to sort (in place)
    returns: None
    """
    for i in range(len(x)):
      j = i - 1
      # swap x[i] toward front of list
      while j >= 0 and x[i] < x[j]:
        if x[i] < x[j]:
          swap(x, i, j)
          i = j # no effect next iteration!
        j -= 1
```

Outer loop grows as O(n)

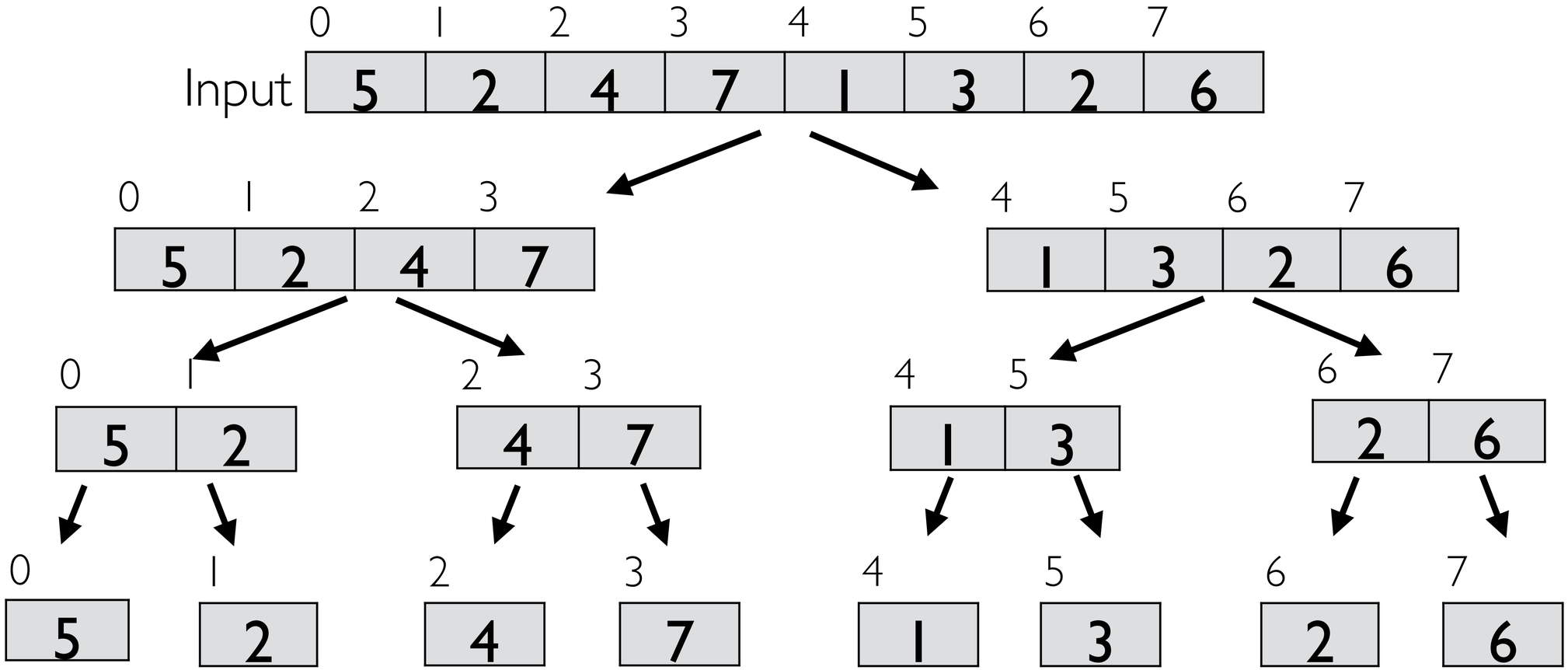
Inner loop grows as O(n)

Therefore, insertion sort is also $O(n \times n) = O(n^2)$

Complexity of merge sort

N = 8 → $\log_2(8) = 3$ → divide input 3 times

Depth complexity is O(log n)



N elements

N/2 elements

N/4 elements

N/8 elements

At each level, process O(n) elements

Therefore, merge sort is O(n × log n)

AMORTIZED TIME

Amortized costs

amortize *(verb)* gradually write off the initial cost (of an asset) over a period

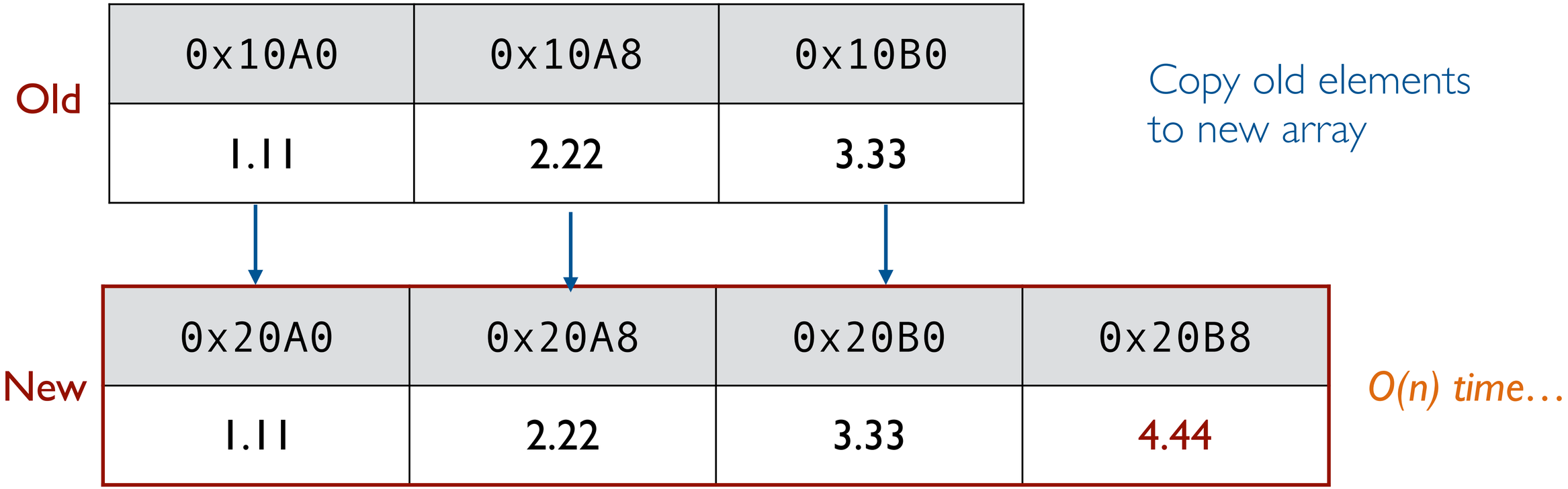
- Computing operations each have a cost
- Can we pay the cost at a different time?
- Consider algorithm's whole lifetime
  - Improve average time for each individual operation
  - Reduce frequency of expensive operations

Appending to an array in Python

```
x = array.array("d", [1.11, 2.22, 3.33])

x.append(4.44)
```

Need to allocate a new block of memory



Copy old elements to new array

O(n) time…

A dynamic array

```
x = DynamicArray("d", [1.11, 2.22, 3.33])

x.append(4.44)
```

Over-allocate more memory than needed

0x10A0	0x10A8	0x10B0	0x10B8
1.11	2.22	3.33	-

0x10A0	0x10A8	0x10B0	0x10B8
1.11	2.22	3.33	4.44

O(1) time!

Extending a dynamic array

```
x.append(5)
```

Over-allocate a new block of memory

Old

0x10A0	0x10A8	0x10B0	0x10B8
1.11	2.22	3.33	4.44

Copy old elements to new array

Leave room to next append operations!

New

0x20A0	0x20A8	0x20B0	0x20B8	0x20C0	0x20C8	0x20D0	0x20D8
1.11	2.22	3.33	4.44	5	-	-	-

O(n) time…

Appending to a dynamic array

```
x.append(99)
```

Amortize cost of future append operations

0x20A0	0x20A8	0x20B0	0x20B8	0x20C0	0x20C8	0x20D0	0x20D8
1.11	2.22	3.33	4.44	5	-	-	-

0x20A0	0x20A8	0x20B0	0x20B8	0x20C0	0x20C8	0x20D0	0x20D8
1.11	2.22	3.33	4.44	5	99	-	-

O(1) time!

Amortized analysis

- Consider whole *lifetime* of algorithm
- Consider most expensive operations
  - Alter program *state* to avoid need for expensive operation again soon
  - *E.g.*, allocate extra space for future appends to a dynamic array
- Requires algorithm to have an alterable *state*
  - Mutable data structure
  - Memoized function

Asymptotic vs. amortized analysis

- Asymptotic analysis
  - Considers each run of an algorithm independently
  - Considers *best/average/worst* case of input size ***n***
  - "Average case" means over all possible inputs
- Amortized analysis
  - Considers multiple runs of an algorithm with *state*
  - Considers a worst case average time
  - "Average time" means over many runs of the algorithm

EFFICIENCY OF COMMON OPERATIONS

Performance of arrays

- *O(1)* random *read/write*
- *O(n) searching*
- *O(n) insertion/deletion*
- *O(1) insertion at end* (if amortized)

Performance of linked lists

- *O(n)* random *read/write*
- *O(n) searching*
- *O(1) insertion/deletion* (at current node)
- *O(n) insertion/deletion* (at random node)

Performance of hash tables

- *O(1) search/access* by key (average case)
- *O(n) search/access* by key(worst case)
- *O(1) insertion/deletion* (average case)
- *O(n) insertion/deletion* (worst case)

Performance of binary search trees

- *O(log n) search/access* by key (average case)
- *O(n) search/access* by key (worst case)
- *O(log n) insertion/deletion* (average case)
- *O(n) insertion/deletion* (worst case)

Comparison of data structures

	Average Case			*Worst Case*		
	Index	Search	Insert/ Delete	Index	Search	Insert/ Delete
Array	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)
Linked list	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)
Hash table	-	O(1)	O(1)	-	O(n)	O(n)
BST	-	O(log n)	O(log n)	-	O(n)	O(n)