CS2400 - Data Structures and Advanced Programming
Module 4: Efficiency of Algorithms

Hao Ji
Computer Science Department
Cal Poly Pomona

© 2018 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

What is “best”?

- An algorithm has both time and space constraints
  - **Time complexity**: the time the algorithm takes to execute
  - **Space complexity**: the memory the algorithm needs to execute

- ***A “best” algorithm*** *might be the fastest one or the one that uses the least memory*

- This process of measuring the complexity of algorithms is called ***analysis of algorithms***

Importance of Efficiency

- ***Problem Size:*** The number of items that an algorithm processes

- Consider the problem of summing

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n$$

Importance of Efficiency

- ***Problem Size:*** The number of items that an algorithm processes

- Consider the problem of summing

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n$$

Algorithm A	Algorithm B	Algorithm C

`sum = 0`
for `i = 1` *to* `n`
    `sum = sum + i`
	`sum = 0`
	for `i = 1` *to* `n`
	`{`
	    for `j = 1` *to* `i`
	        `sum = sum + 1`
	`}`
		`sum = n * (n + 1) / 2`

Counting Basic Operations

Algorithm A	Algorithm B	Algorithm C

sum = 0
for i = 1 *to* n
    sum = sum + i
	sum = 0
	for i = 1 *to* n
	{
	    for j = 1 *to* i
	        sum = sum + 1
	}
		sum = n * (n + 1) / 2

	Algorithm A	Algorithm B	Algorithm C
Additions			
Multiplications			
Divisions			
Total basic operations			

Most significant contributor to its total time requirement.

Counting Basic Operations

Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + i
```

Algorithm B

```
sum = 0
for i = 1 to n
{
    for j = 1 to i
        sum = sum + 1
}
```

Algorithm C

```
sum = n * (n + 1) / 2
```

	Algorithm A	Algorithm B	Algorithm C
Additions	n		
Multiplications			
Divisions			
Total basic operations	n		

Counting Basic Operations

Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + i
```

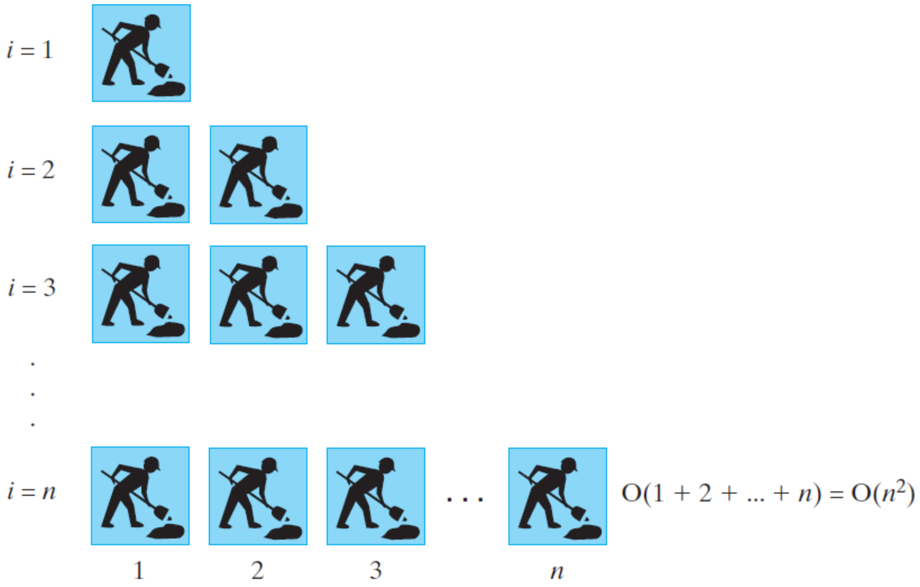
Algorithm B

```
sum = 0
for i = 1 to n
{
    for j = 1 to i
        sum = sum + 1
}
```

Algorithm C

```
sum = n * (n + 1) / 2
```

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n+1)/2$	
Multiplications			
Divisions			
Total basic operations	n	$(n^2+n)/2$	



Counting Basic Operations

Algorithm A	Algorithm B	Algorithm C

<pre>sum = 0
for i = 1 *to* n
    sum = sum + i</pre>
	<pre>sum = 0
	for i = 1 *to* n
	{
	    for j = 1 *to* i
	        sum = sum + 1
	}</pre>
		<pre>sum = n * (n + 1) / 2</pre>

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n+1)/2$	1
Multiplications			1
Divisions			1
Total basic operations	$\boldsymbol{n}$	$\boldsymbol{(n^2+n)/2}$	**3**

Counting Basic Operations

Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + i
```

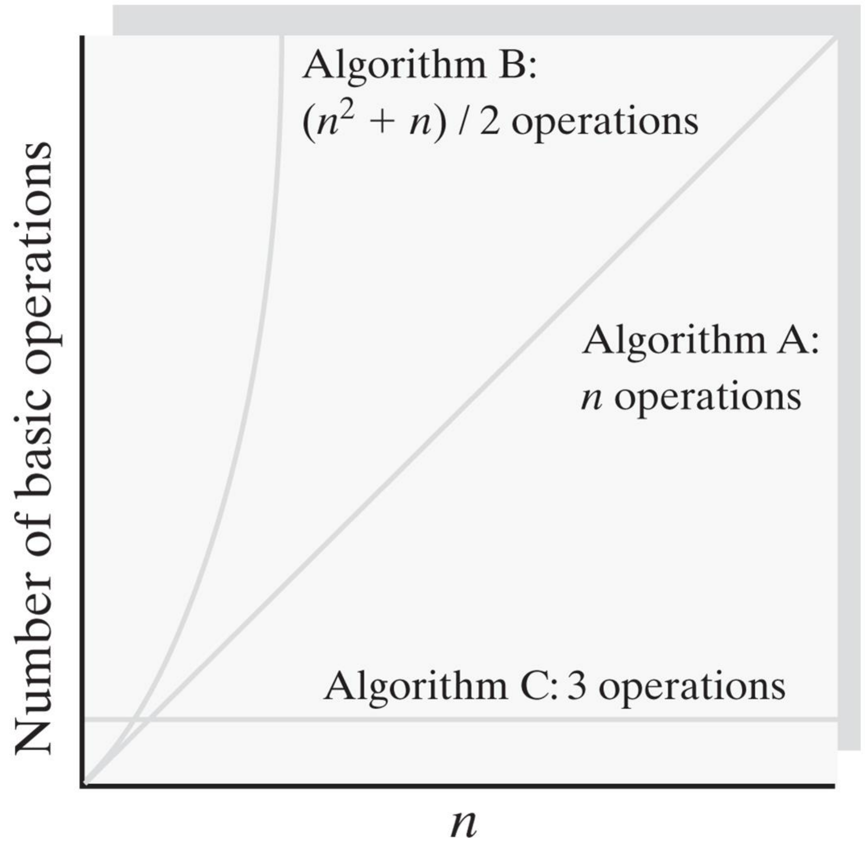
Algorithm B

```
sum = 0
for i = 1 to n
{
    for j = 1 to i
        sum = sum + 1
}
```

Algorithm C

```
sum = n * (n + 1) / 2
```

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n+1)/2$	1
Multiplications			1
Divisions			1
Total basic operations	n	$(n^2+n)/2$	**3**



© 2019 Pearson Education, Inc.

Growth-Rate Function

- Computing the actual time requirement of an algorithm is not easy, especially for large problem size.

- In Computer science, we use a function of the problem size that behaves like the algorithm's actual time requirement.

- This function is called a ***growth-rate function***, as it measures how an algorithm's time requirement grows as the problem size grows.

Growth-Rate Function

- Computing the actual time requirement of an algorithm is not easy, especially for large problem size.

- In Computer science, we use a function of the problem size that behaves like the algorithm's actual time requirement.

- This function is called a ***growth-rate function***, as it measures how an algorithm's time requirement grows as the problem size grows.

Typical growth-rate functions are algebraically simple.

Asymptotic Notation

- Upper bounds
  - $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.
- Lower bounds
  - $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.
- Tight bounds
  - $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Asymptotic Notation

- Upper bounds (**Big Oh Notation**)
  - $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.

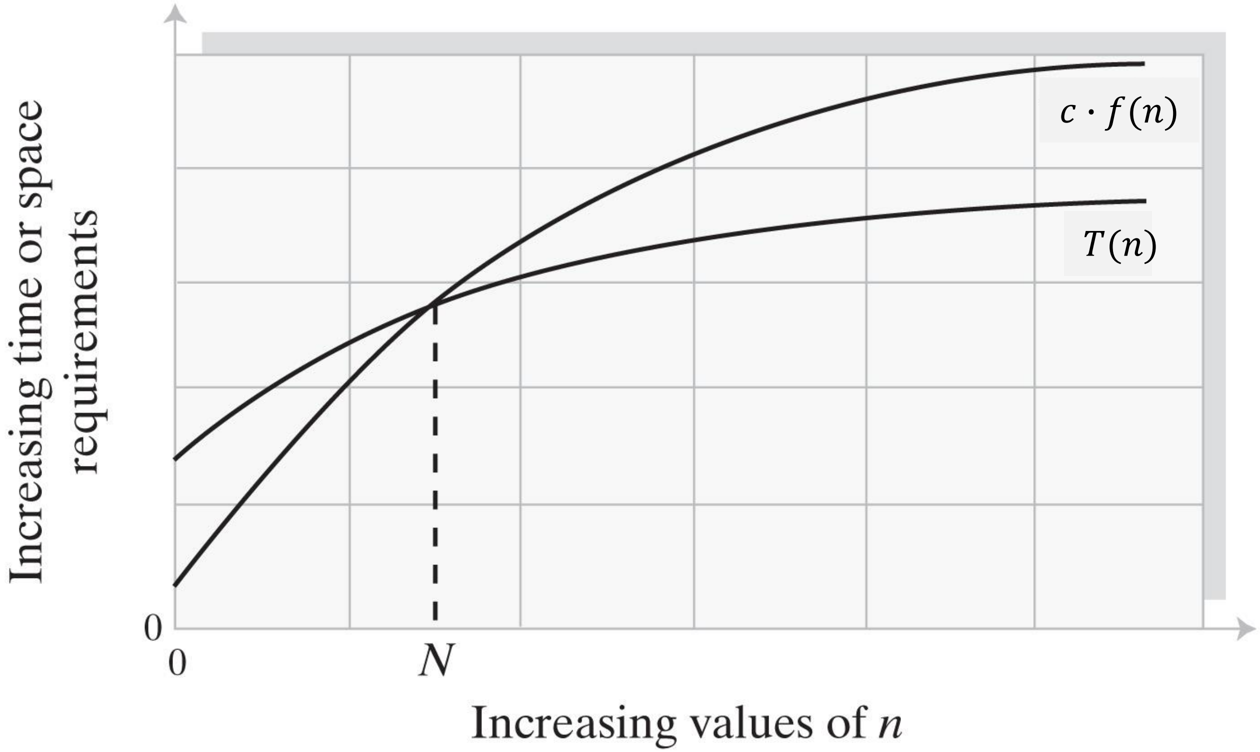
- Lower bounds
  - $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.

- **Tight bounds**
  - $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Asymptotic Notation

$O(f(n))$ means the algorithm requires time proportional to $f(n)$

- Upper bounds (**Big Oh Notation**)
  - $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.



© 2019 Pearson Education, Inc.

Asymptotic Notation (Cont’d)

- Rules of using Big O notation.
  - **If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$.**
    - We can drop the lower order terms and constant factors.
  - **Use the smallest/closest possible class of functions**
    - For example, "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
  - **Use the simplest expression of the class,**
    - For example "$3n + 5$ is $O(n)$" instead of "$3n+5$ is $O(3n)$"

Asymptotic Notation (Cont'd)

- Upper bounds: $T(n) = O(f(n))$
- Lower bounds: $T(n) = \Omega(f(n))$
- Tight bounds: $T(n) = \Theta(f(n))$

- Example: $3n^3 + 6n^2 - 4n + 17$, which of the following statements is true?
  - $T(n)$ is $O(n^2)$
  - $T(n)$ is $O(n^3)$
  - $T(n)$ is $\Omega(n^2)$
  - $T(n)$ is $\Omega(n)$
  - $T(n)$ is $\Theta(n^2)$
  - $T(n)$ is $O(n)$
  - $T(n)$ is $\Omega(n^3)$
  - $T(n)$ is $\Theta(n)$
  - $T(n)$ is $\Theta(n^3)$

Analysis of Algorithms

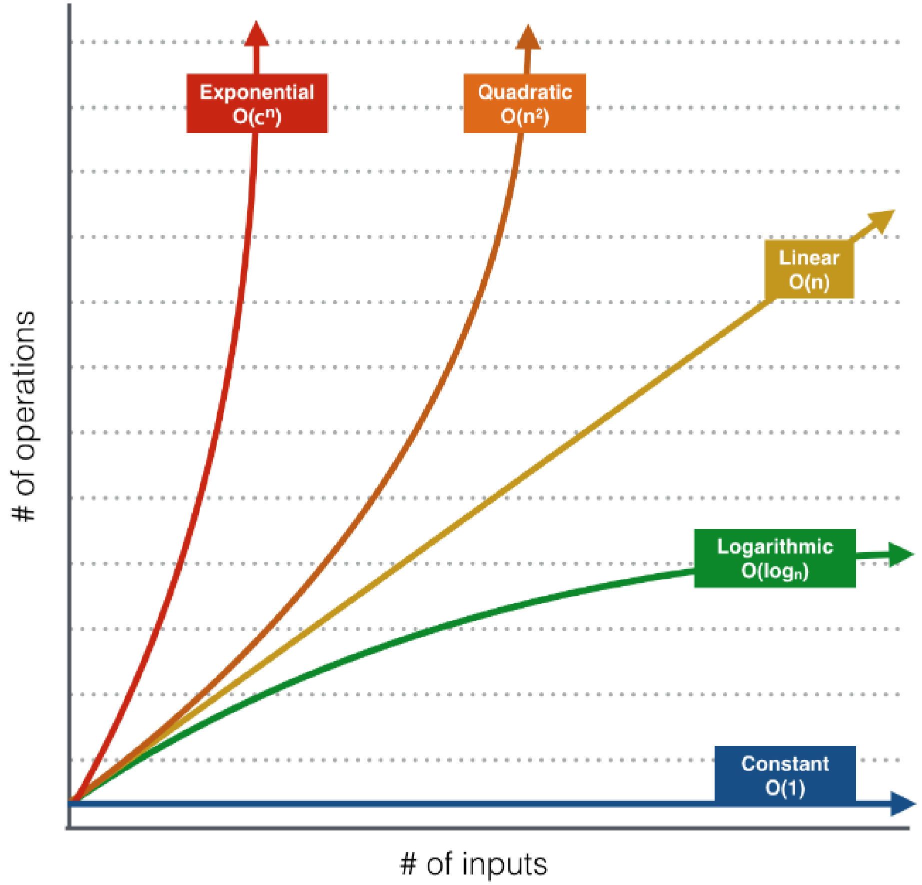
- For example,
  - Traversing an array of n elements
  - Accessing the ith element in an array

Analysis of Algorithms (Cont'd)

- For example,
  - Traversing an array of n elements
    - The time needed is proportional to n, and we say the time is in the order of $O(n)$.
  - Accessing the ith element in an array
    - The time needed is only constant time, which is independent of the size of the array, thus is in the order of $O(1)$.

Analysis of Algorithms (Cont'd)

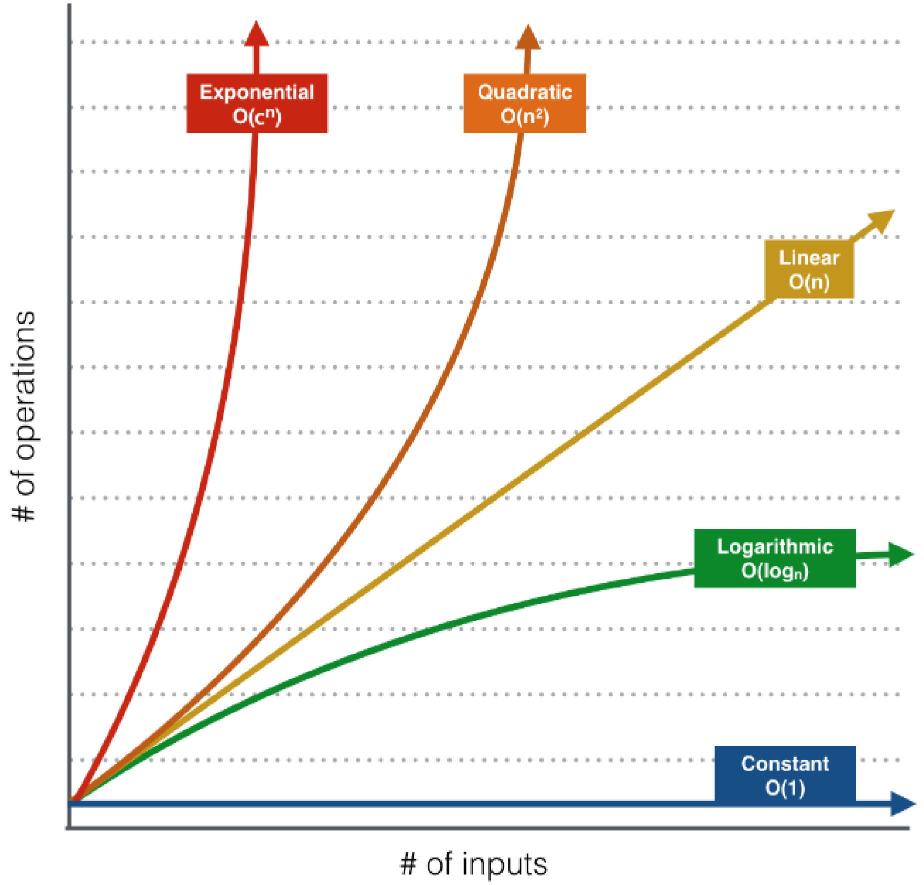
- Common functions used in analysis
  - Constant function $f(n) = C$
    - *Great. Constant algorithm does not depend on the input size.*
  - Logarithm function $f(n) = \log n$
    - Very good. *Logarithm function gets slightly slower as n grows.*
  - Linear function $f(n) = n$
    - *Linea time. Whenever n doubles, so does the running time.*
  - N-Log-N function $f(n) = n \log n$
    - *It grows a little faster than the linear function. Typically, the time needed for sorting a list of elements*
  - Quadratic function $f(n) = n^2$
    - *Whenever n doubles, the running time increases fourfold.*
  - Exponential Function $f(n) = c^n$
  - Factorial Function $f(n) = n!$



http://www.blankmaker.com/basics-time-complex

Analysis of Algorithms (Cont'd)

- **In-Class Exercises:**
  - What is the Big Oh Notation of $3n^2 + 2^n$?
  - Show that $log_b\, n$ is $O(log_2\, n)$. What values of c and N did you use?



http://www.blankmaker.com/basics-time-complex

Worst-Case, Average-Case and Best-Case Analyses

- For some algorithms, execution time depends only on size of data set

- Other algorithms depend on the nature of the data itself
    - Goal is to know best case, worst case, average case

Worst-Case, Average-Case and Best-Case Analyses

- The **best case** is that the algorithm takes the least time.
  - The algorithm can do no better than its best-case time.
  - If the best-case time is still too slow, you need another algorithm.
- The **worst case** is that the algorithm takes the most time.
  - If the algorithm can tolerate this worst-case time, that algorithm is acceptable.
- The **average-case** is a more useful measure of time requirement.
  - Typically, the best and worst cases do not occur.
  - The average-case is harder to estimate, compared to best case and worst case.

Worst-Case, Average-Case and Best-Case Analyses

- The **best case** is that the algorithm takes the least time.
  - The algorithm can do no better than its best-case time.
  - If the best-case time is still too slow, you need another algorithm.
- The **worst case** is that the algorithm takes the most time.
  - If the algorithm can tolerate this worst-case time, that algorithm is acceptable.
- The **average-case** is a more useful measure of time requirement.
  - Typically, the best and worst cases do not occur.
  - The average-case is harder to estimate, compared to best case and worst case.

- What is the time needed to check if an array of integers contains an integer “10”?

Efficiency of Implementations of a Bag

- Adding an entry to a bag
  - Array-based implementation
  - Linked Implementation

- Searching a bag for a given entry
  - Array-based implementation
  - Linked Implementation

Efficiency of Implementations of a Bag

- **Adding an entry to a bag**
  - **Array-based implementation**
  - Linked Implementation

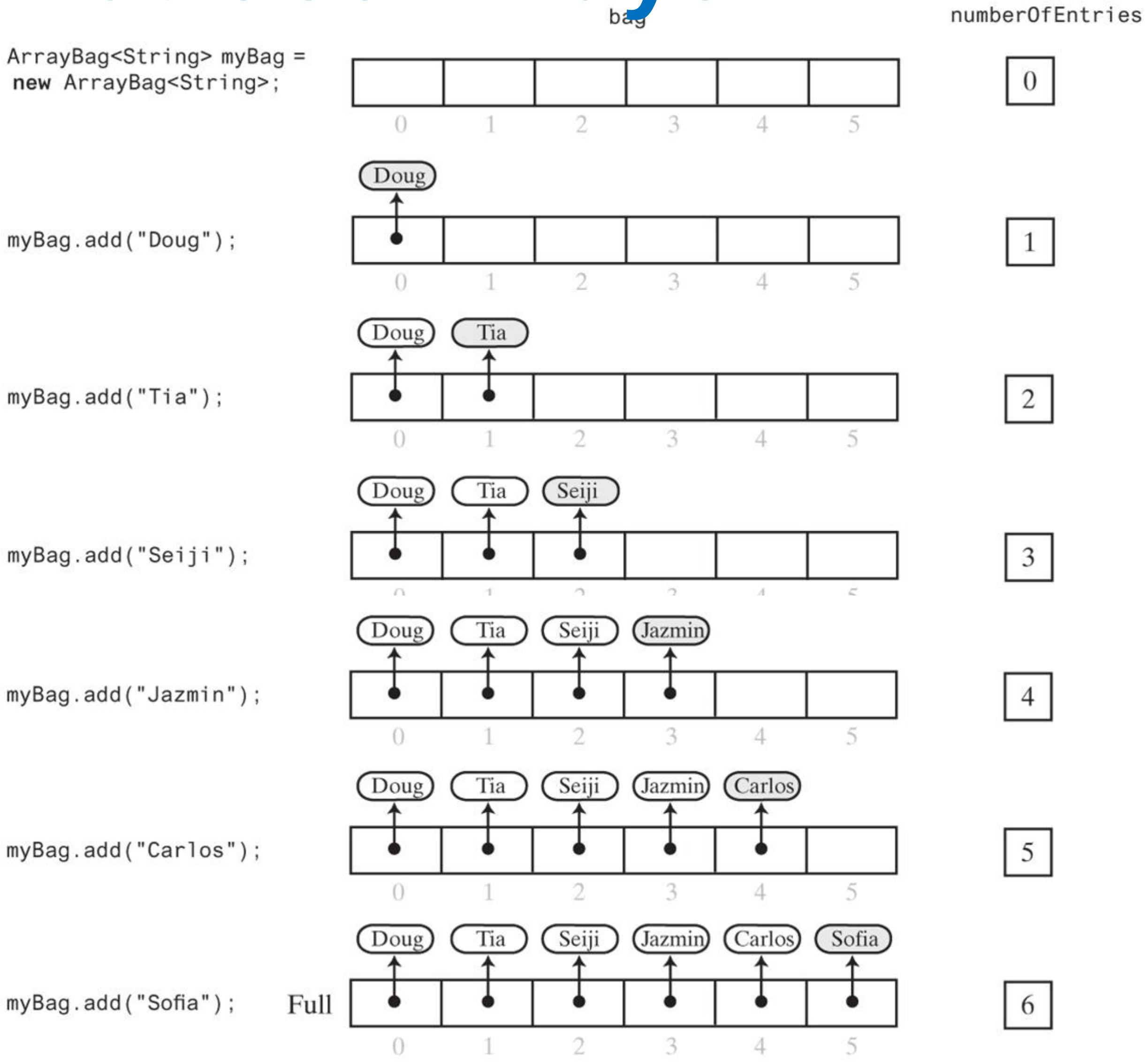
- Searching a bag for a given entry
  - Array-based implementation
  - Linked Implementation

Bag Implementations That Use Arrays

- The method **add**
  - If the bag is full, we cannot add anything to it. In that case, the method add should return false
  - Otherwise, we simply add newEntry immediately after the last entry in the array bag

```
public boolean add(T newEntry)
{
   boolean result = true;
   if (isFull())
   {
      result = false;
   }
   else
   {  // assertion: result is true here
      bag[numberOfEntries] = newEntry;
      numberOfEntries++;
   } // end if

   return result;
} // end add
```



© 2019 Pearson Education, Inc.

Efficiency of Implementations of a Bag

Operation	Fixed-Size Array	Linked
`add(newEntry)`	O(1)	

Efficiency of Implementations of a Bag

- **Adding an entry to a bag**
  - Array-based implementation
  - **Linked Implementation**

- Searching a bag for a given entry
  - Array-based implementation
  - Linked Implementation

A Linked Implementation of a Bag

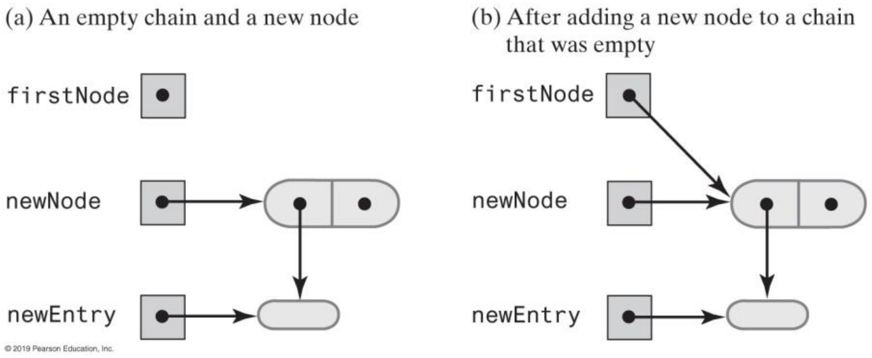
- The method **add**

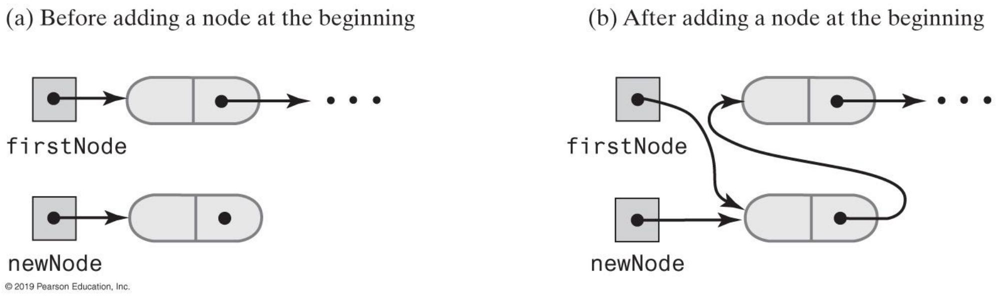
```java
/** Adds a new entry to this bag.
    @param newEntry  The object to be added as a new entry
    @return  True if the addition is successful, or false if not. */

public boolean add(T newEntry)        // OutOfMemoryError possible
{
     // Add to beginning of chain:
     Node newNode = new Node(newEntry);
     newNode.next = firstNode; // Make new node reference rest of chain
                               // (firstNode is null if chain is empty)

     firstNode = newNode;      // New node is at beginning of chain
     numberOfEntries++;

     return true;
} // end add
```





Efficiency of Implementations of a Bag

Operation	Fixed-Size Array	Linked
`add(newEntry)`	O(1)	O(1)

Efficiency of Implementations of a Bag

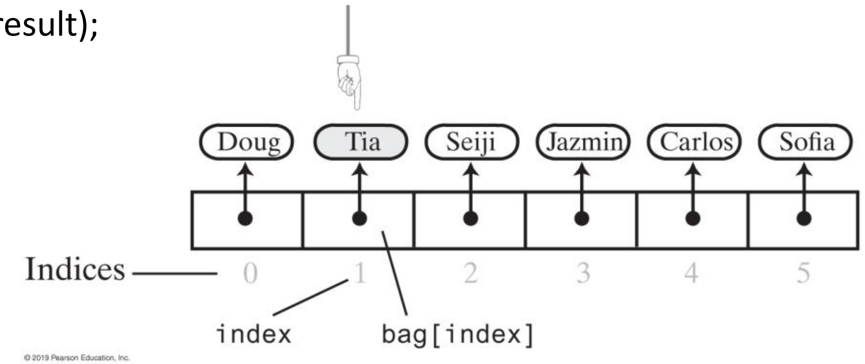
- Adding an entry to a bag
  - Array-based implementation
  - Linked Implementation

- **Searching a bag for a given entry**
  - **Array-based implementation**
  - Linked Implementation

Implementing More Methods

- **Removing a given entry**
  - **Search for the entry**
  - Remove the entry from the bag

```
/** Removes one occurrence of a given entry from this bag.
@param anEntry  The entry to be removed.
@return  True if the removal was successful, or false if not. */
public boolean remove(T anEntry)
{
        checkIntegrity();
        int index = getIndexOf(anEntry);
        T result = removeEntry(index);
        return anEntry.equals(result);
} // end remove
```




```
// Locates a given entry within the array bag.
// Returns the index of the entry, if located, or -1 otherwise.
// Precondition: checkIntegrity has been called.
private int getIndexOf(T anEntry)
{
        int where = -1;
        boolean found = false;
        int index = 0;

        while (!found && (index < numberOfEntries))
        {
                if (anEntry.equals(bag[index]))
                {
                        found = true;
                        where = index;
                } // end if
                index++;
        } // end while

        // Assertion: If where > -1, anEntry is in the array bag, and it
        // equals bag[where]; otherwise, anEntry is not in the array

        return where;
} // end getIndexOf
```

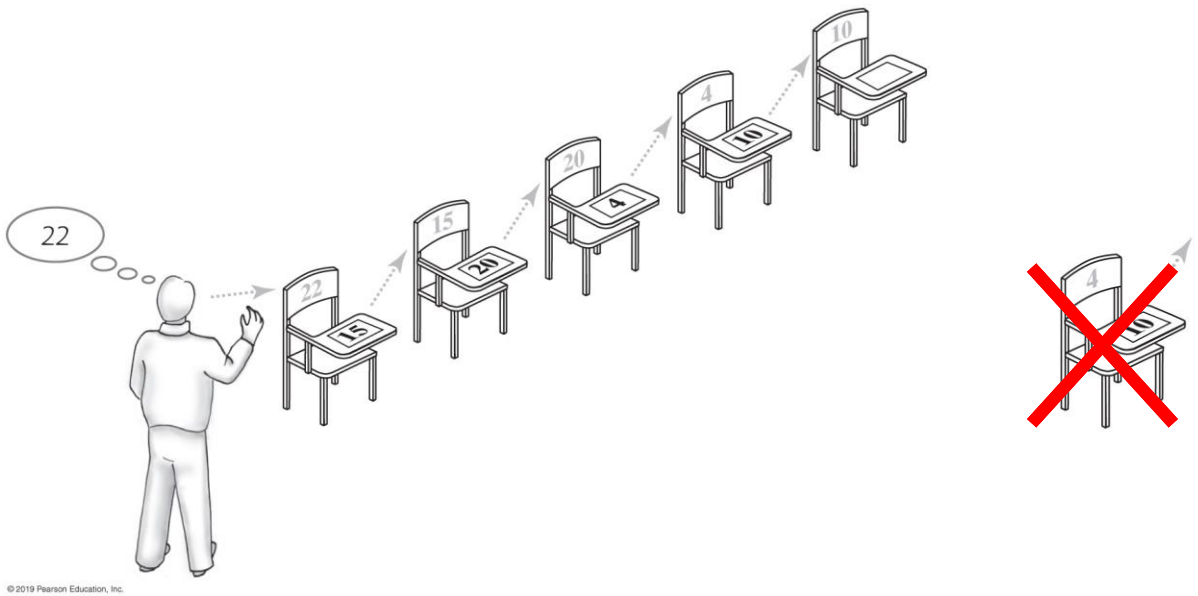
Efficiency of Implementations of a Bag

Operation	Fixed-Size Array	Linked
`add(newEntry)`	O(1)	O(1)
`remove(anEntry)`	**O(1), O(*n*), O(*n*)**	
`contains(anEntry)`	**O(1), O(*n*), O(*n*)**	

Time requirements for the **best**, **worst**, and **average** cases

A Linked Implementation of a Bag

- The method **remove**
  - Removing an unspecified entry from a bag
  - **Removing a given entry from a bag**
    - **Search for the given entry in a bag**
    - Remove the given entry




```java
// Locates a given entry within this bag.
// Returns a reference to the node containing the //
entry, if located, or null otherwise.
private Node getReferenceTo(T anEntry)
{
      boolean found = false;
      Node currentNode = firstNode;

      while (!found && (currentNode != null))
      {
            if (anEntry.equals(currentNode.getData()))
                        found = true;
            else
            currentNode = currentNode.getNextNode();
      } // end while

      return currentNode;
} // end getReferenceTo
```

Efficiency of Implementations of a Bag

Operation	Fixed-Size Array	Linked
`add(newEntry)`	O(1)	O(1)
`remove(anEntry)`	O(1), O(*n*), O(*n*)	**O(1), O(*n*), O(*n*)**
`contains(anEntry)`	O(1), O(*n*), O(*n*)	**O(1), O(*n*), O(*n*)**

Time requirements for the **best**, **worst**, and **average** cases

Efficiency of Implementations of a Bag

Operation	Fixed-Size Array	Linked
`add(newEntry)`	O(1)	O(1)
`remove()`		
`remove(anEntry)`	O(1), O(*n*), O(*n*)	O(1), O(*n*), O(*n*)
`clear()`		
`getFrequencyOf(anEntry)`		
`contains(anEntry)`	O(1), O(*n*), O(*n*)	O(1), O(*n*), O(*n*)
`toArray()`		
`getCurrentSize(), isEmpty()`		

Efficiency of Implementations of a Bag

Operation	Fixed-Size Array	Linked
`add(newEntry)`	O(1)	O(1)
`remove()`	O(1)	O(1)
`remove(anEntry)`	O(1), O(*n*), O(*n*)	O(1), O(*n*), O(*n*)
`clear()`	O(*n*)	O(*n*)
`getFrequencyOf(anEntry)`	O(*n*)	O(*n*)
`contains(anEntry)`	O(1), O(*n*), O(*n*)	O(1), O(*n*), O(*n*)
`toArray()`	O(*n*)	O(*n*)
`getCurrentSize(), isEmpty()`	O(1)	O(1)

Efficiency of Implementations of a Bag

Operation	Fixed-Size Array	Resizable Array	Linked
`add(newEntry)`			
`remove()`			
`remove(anEntry)`			
`clear()`			
`getFrequencyOf(anEntry)`			
`contains(anEntry)`			
`toArray()`			
`getCurrentSize(), isEmpty()`			

Summary

- Efficiency of Algorithms

What I Want You to Do

- Review class slides
- Review Chapter 4

- Next Topic
  - ADT List
  - Implementations of a List