Measuring Efficiency

Measuring Efficiency

- **Even a simple program can be "inefficient".**
- **What is an efficient algorithm?**
  - Algorithms take time to execute
  - Algorithms need storage for data and variables
- **Complexity**
  - Time and storage requirements of an algorithm
- **Analysis of algorithms**
  - Measuring of the complexity of an algorithm

Types of Complexity

- **Time complexity**
  - Speed (number of operations)
- **Space complexity**
  - Storage (memory or disk)
- **Inverse relationship**
  - Faster algorithms can require more space
  - Reducing storage can increase execution time

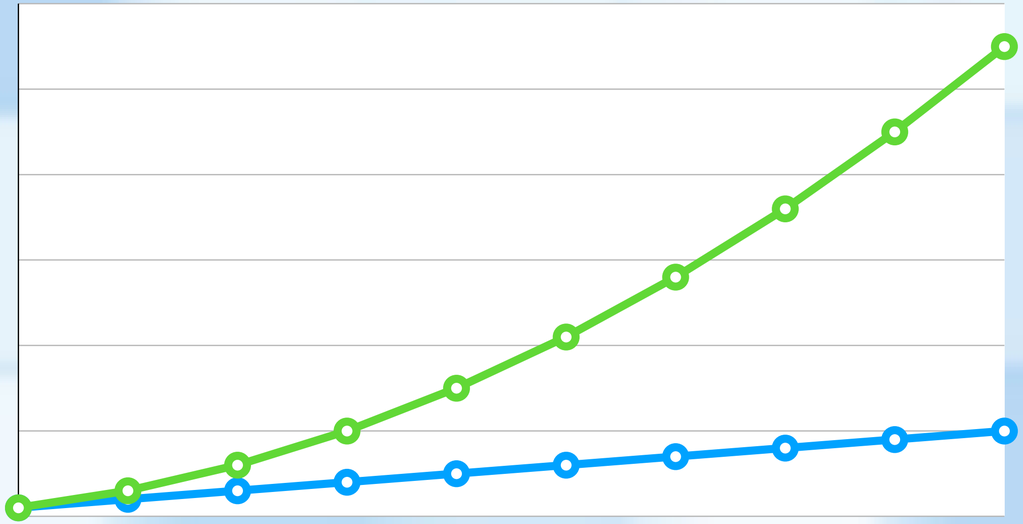
Measuring Complexity

- Express complexity in *problem size*
- Size is number of items processed by algorithm
- Number of items usually represented by `n`
- Our interest is in large problems
  - Small problems can take very little time even if the algorithm is inefficient

`n = 7`

Measuring Complexity

- **Do not compute actual time for an algorithm**
  - Execution time is dependent on
    - Language and compiler efficiency
    - Computer capabilities
- **Compute growth-rate function**
  - Measurement directly proportional to the algorithm's time requirement
  - Gives common basis for comparison



Pearson

Measuring Complexity

- **Comparing algorithms should be independent of**
  - ***Specific Implementations***
    - how are the algorithms coded
  - ***Computer***
    - what computer is used
  - ***Data***
    - the data the program uses

Growth-Rate Functions

GROWTH-RATE FUNCTIONS

- **Find the sum of the first `n` positive integers:**

```
1 + 2 + 3 + … + n-1 + n
for some integer n > 0
```

- **Finding the growth-rate function**
  - Count basic opera

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`

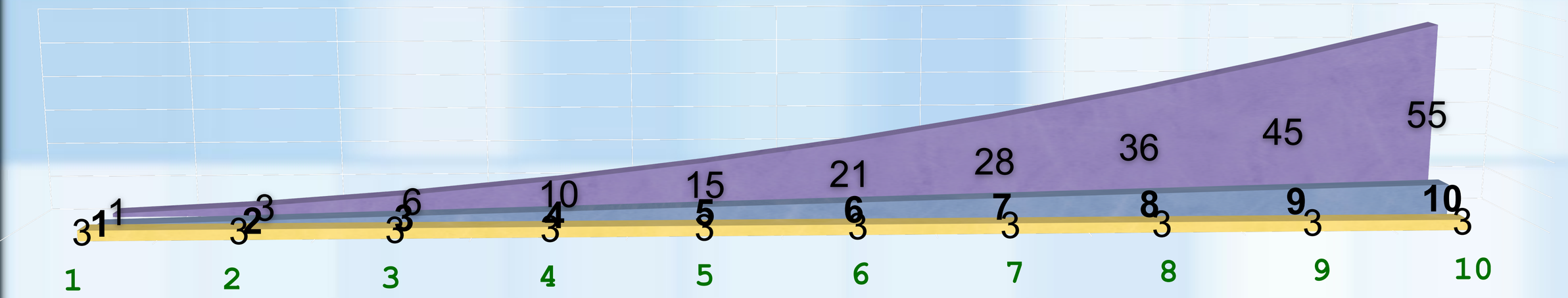
Additions
Multiplications
Divisions
Total Operations

Count "action" statements
-- Statements directly related to accomplishing goal
Additions, Multiplications, Comparisons, Moves
-- Ignore "bookkeeping" statements

Pearson

Growth-Rate Functions

- **Number of operations required as a function of `n`**
  - Algorithm C `3`
  - Algorithm A `n`
  - Algorithm B $(n^2+n)/2$



GROWTH-RATE FUNCTIONS

- **Number of operations required as a function of `n`**
  - Algorithm C `3`
  - Algorithm A `n`
  - Algorithm B `(n²+n)/2`

Consider only the dominant term in each growth-rate function

For very large `n`:

`n²` is much larger than `n`

so `(n² + n)/2` behaves like `n²/2`

and `n²/2` behaves like `n²`

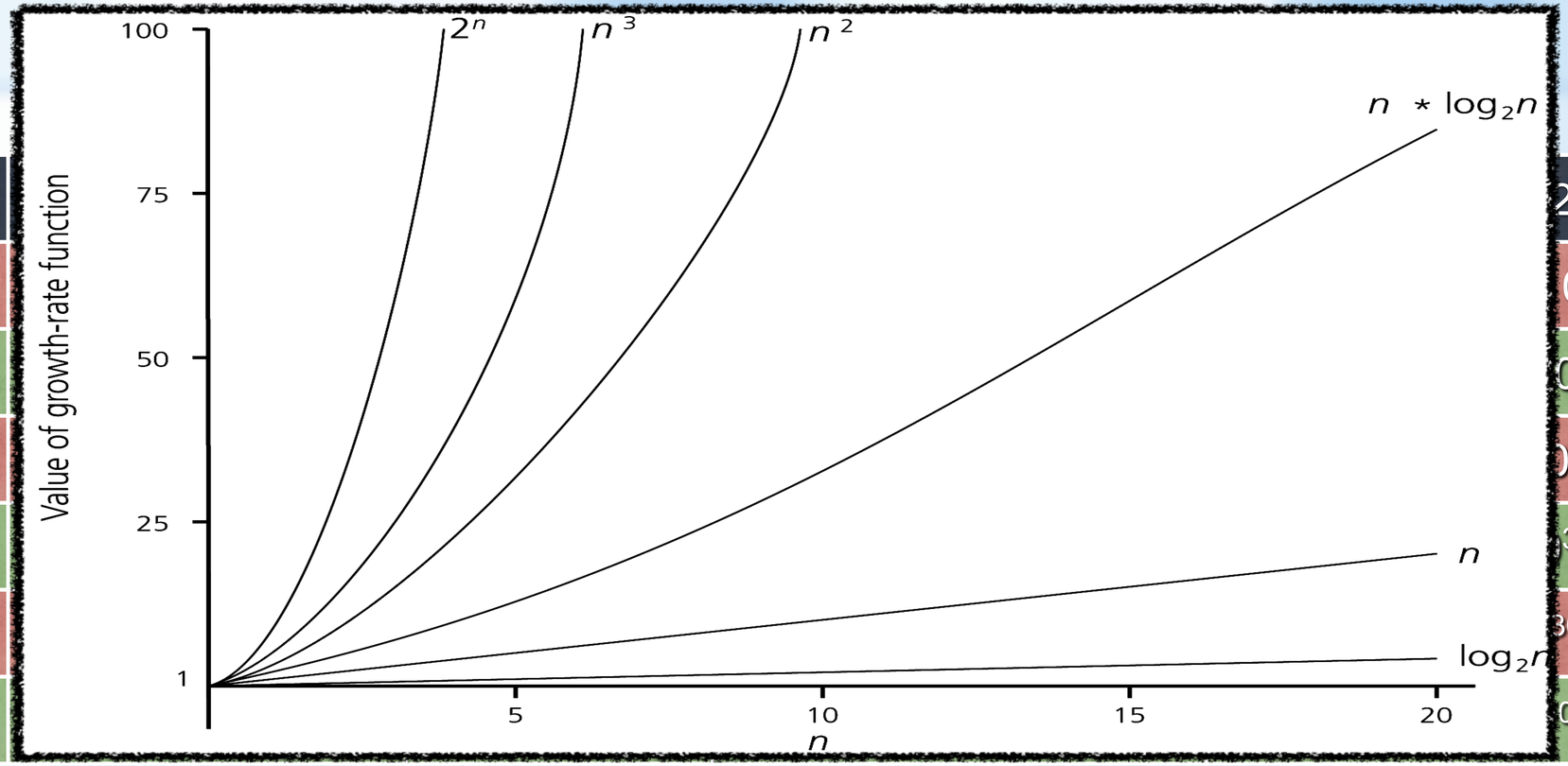
so `n²/2` behaves like `n²`

Pearson

GROWTH-RATE FUNCTIONS

- **Number of operations required as a function of `n`**

n	log(log n)				2^n	n!
10	2				0^3	10^5
100	3				0^{30}	10^{94}
1,000	3				$)^{301}$	10^{1435}
10^4	4				,3010	$10^{19,335}$
10^5	4				30,103	$10^{243,338}$
10^6	4				01,030	$10^{2,9332369}$



$$1 < \log_2 n < n < n * \log_2 n < n^2 < n^3 < 2^n < n!$$

Pearson

Representing Algorithm Complexity

- **Big O notation**
  - Order of at most ***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`

Total Operations	n	$(n^2 + n)/2$	3
Big O (Order of Magnitude)	$O(n)$	$O(n^2)$	$O(1)$

Pearson

Representing Algorithm Complexity

- Effect of doubling the problem size on an algorithm's time requirement

Growth-Rate Function for Size n Problems	Growth-Rate Function for Size 2n Problems	Effect on Time Requirement
1	1	None
log n	1 + log n	Negligible
n	2n	Doubles
n log n	2n log n + 2n	Doubles then add 2n
n^2	$(2n)^2$	Quadruples
n^3	$(2n)^3$	Multiplies by 8
2^n	2^{2n}	Squares

Representing Algorithm Complexity

- Time to process one million items at a rate of one million operations per secon

Growth-Rate Function for Size n Problems	Time to process one million items at a rate of one million operation per second
log n	0.0000199 seconds
n	1 second
n log n	19.9 seconds
n^2	11.6 days
n^3	31,709.8 years
2^n	$10^{301,016}$ years

Understanding Big O Notation

PICTURING EFFICIENCY

- An O(n) algorithm

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

1 + 1+ 1 + 1 + ... + 1 + 1 ≈ O(n)

- An O(1) algorithm

```
for i = 1 to 1000
        sum = sum + n
```

1000 ≈ O(1)

Pearson

PICTURING EFFICIENCY

- An $O(n^2)$ algorithm

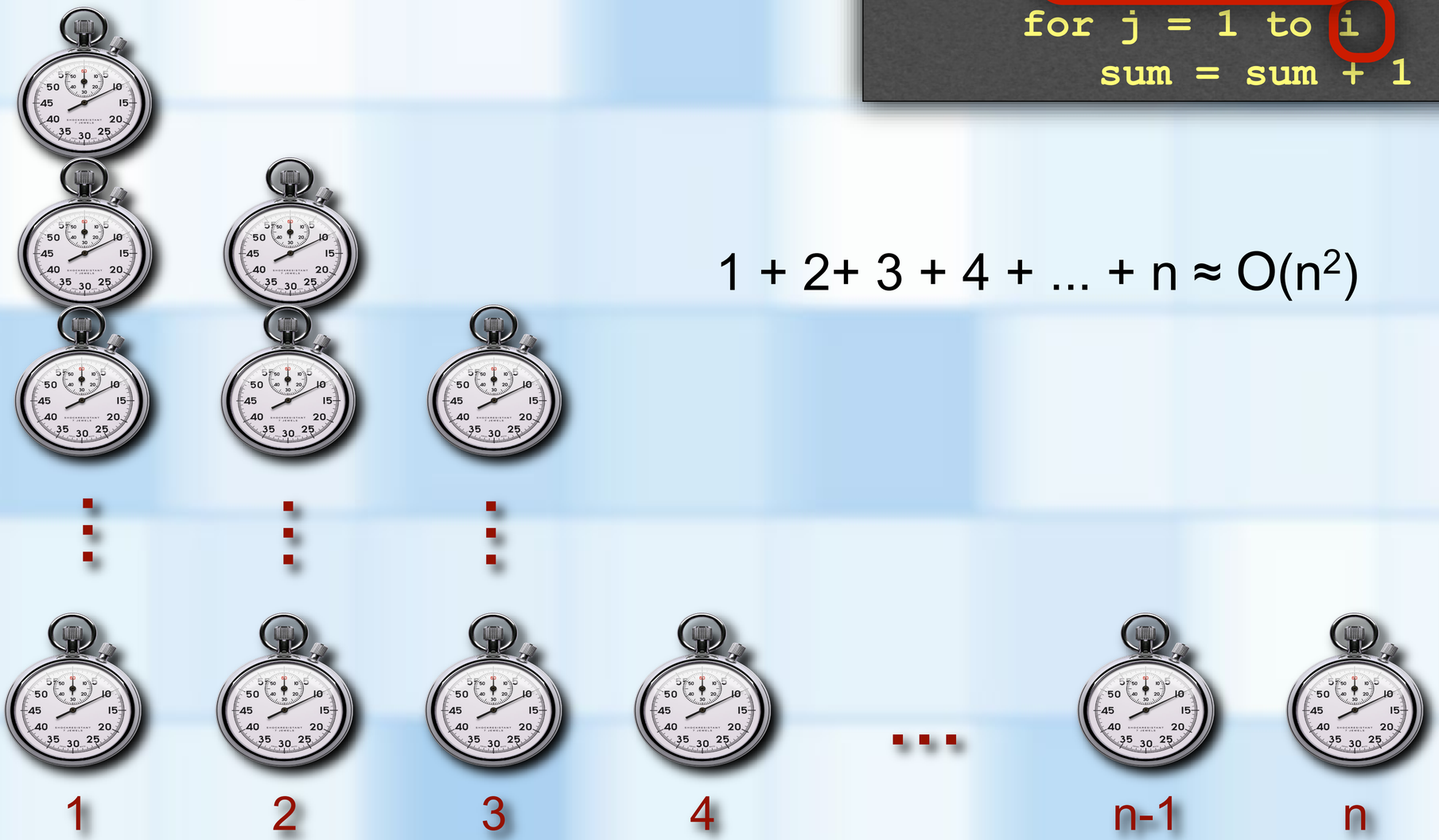
```
for i = 1 to n
  for j = 1 to n
    sum = sum + 1
```

$$n + n + n + n + \ldots + n = n * n \approx O(n^2)$$

PICTURING EFFICIENCY

- Another $O(n^2)$ algorithm

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



$$1 + 2 + 3 + 4 + \ldots + n \approx O(n^2)$$

Comparing Implementations

- **Choosing an implementation**
  - Look for significant differences in efficiency
  - Frequency of operations
    - consider how frequently particular ADT operations occur in a given application
    - sometimes seldom-used but critical operations must be efficient
- **Best-case analysis**
  - Determine minimum amount of time an algorithm requires to solve problems of size `n`
- **Worst-case analysis**
  - Determine maximum amount of time an algorithm requires to solve problems of size `n`
- **Average-case analysis**
  - Determine average amount of time an algorithm requires to solve problems of size `n`

ADT Bag Interface

```
template<class ItemType>
class BagInterface
{
public:
 virtual int getCurrentSize() const = 0;
 virtual bool isEmpty() const = 0;
 virtual bool add(const ItemType& newEntry) = 0;
 virtual bool remove(const ItemType& target) = 0;
 virtual void clear() = 0;
 virtual int getFrequencyOf(const
                            ItemType& target) const = 0;
 virtual bool contains(const
                       ItemType& anEntry) const = 0;
 virtual std::vector<ItemType> toVector() const = 0;
 virtual ~BagInterface() { }
};  // end BagInterface
```

O(1)

```
template<class ItemType>
class ArrayBag : public BagInterface<ItemType>
{
private:
   static const int DEFAULT_CAPACITY = 50;
   ItemType items[DEFAULT_CAPACITY];    // Array of bag items
 int itemCount;              // Current count of bag items
```

```
template<class ItemType>
class LinkedBag : public BagInterface<ItemType>
{
private:
  Node<ItemType>* headPtr; // pointer to first node
 int itemCount;          // current count of bag items
```

Pearson

ADT Bag add Implementations

ArrayBag

```
template<class ItemType>
bool ArrayBag<ItemType>::add(const ItemType& newEntry)
{
   bool hasRoomToAdd = (itemCount < maxItems);
   if (hasRoomToAdd)
   {
      items[itemCount] = newEntry;
      itemCount++;
   }  // end if

   return hasRoomToAdd;
}  // end add
```

O(1)

LinkedBag

```
template<class ItemType>
bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{

   Node<ItemType>* nextNodePtr = new Node<ItemType>();
   nextNodePtr->setItem(newEntry);
   nextNodePtr->setNext(headPtr);

   headPtr = nextNodePtr;

   itemCount++;

   return true;
}  // end add
```

O(1)

ADT Bag contains Implementations

ArrayBag

```
template<class ItemType>
bool ArrayBag<ItemType>::contains(const
                                  ItemType& anEntry) const
{
   bool found = false;
  int searchIndex = 0;

  while (!found && (searchIndex < itemCount))
  {
    found = (items[searchIndex] == anEntry);

    if (!found)
        searchIndex++;

  }  // end while
  return found;
}  // end contains
```

Best Case: O(1)

Worst Case: O(n)

LinkedBag

```
template<class ItemType>
bool LinkedBag<ItemType>::contains(const
                                   ItemType& anEntry) const
{
  bool found = false;
  Node<ItemType>* curPtr = headPtr;

  while (!found && (curPtr != nullptr))
  {
    found = (anEntry == curPtr->getItem());

    if (!found)
      curPtr = curPtr->getNext();
  } // end while
    return found;
}  // end contains
```

Best Case: O(1)

Worst Case: O(n)

Pearson

ADT Bag toVector Implementations

ArrayBag

```
public T[] toArray()
{
  @SuppressWarnings("unchecked")
  T[] result = (T[])new Object[capacity]; // unchecked

  for (int index = 0 ; index < numberOfEntries ; index++)
  {
    result[index] = bag[index];
  } // end for

  return result;
} // end toArray
```

O(n)

LinkedBag

```
public T[] toArray()
{
  @SuppressWarnings("unchecked")
  T[] result = (T[])new Object[numberOfEntries];
  Node<T> currentNode = firstNode;
  int index = 0;

  while ((index < numberOfEntries)
                              && (currentNode != null))
  {
    result[index] = currentNode.getData();
    currentNode = currentNode.getNextNode();
    index++;
  } // end while

  return result;
} // end toArray
```

O(n)

Pearson

COMPARING IMPLEMENTATIONS

ADT Bag method	ArrayBag Implementation	LinkedBag Implementation
getCurrentSize()	O(1)	O(1)
isEmpty()	O(1)	O(1)
add(ItemType anEntry)	O(1)	O(1)
remove(ItemType anEntry)	O(1) to O(n)	O(1) to O(n)
clear()	O(1)	O(1) to O(n)
getFrequencyOf(ItemType anEntry)	O(n)	O(n)
contains(ItemType anEntry)	O(1) to O(n)	O(1) to O(n)
toVector()	O(n)	O(n)

ArrayBag

```
// O(1)
void ArrayBag::clear()
{
   itemCount = 0;
} // end clear
```

LinkedBag

```
// O(n)
void LinkedBag::clear()
{
  Node<ItemType>* deleteMe
                       = nullptr;

  while (headPtr != nullptr)
  {
    deleteMe = headPtr;
    headPtr = headPtr->getNext();

    delete deleteMe;

    itemCount--;
  } // end while
} // end clear
```

Pearson

Comparing Implementations

- **If the problem size is always small,**
  - can probably ignore an algorithm's efficiency
- **Weigh the trade-offs between an algorithm's time requirements and its memory requirements**