CS302 - Data Structures

using C++

Topic: Understanding Big O Notation

Kostas Alexis

AUTONOMOUS ROBOTS LAB

Picturing Efficiency

- An O(n) algorithm

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

Picturing Efficiency

- An O(n) algorithm

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

- An O(1) algorithm

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

Picturing Efficiency

- An O(n) algorithm

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

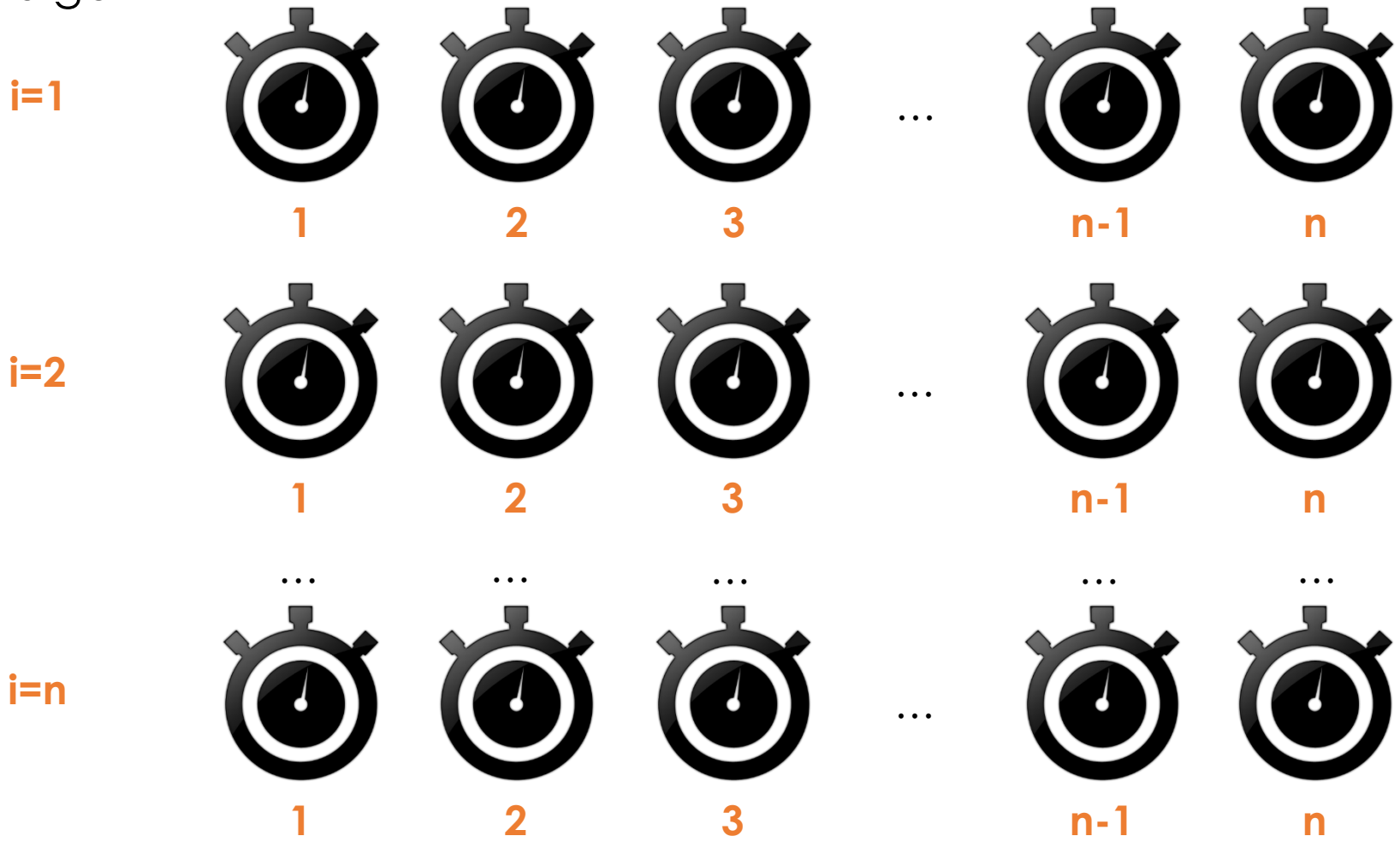
- An O(1) algorithm

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

Picturing Efficiency

- An $O(n^2)$ algorithm

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



Picturing Efficiency

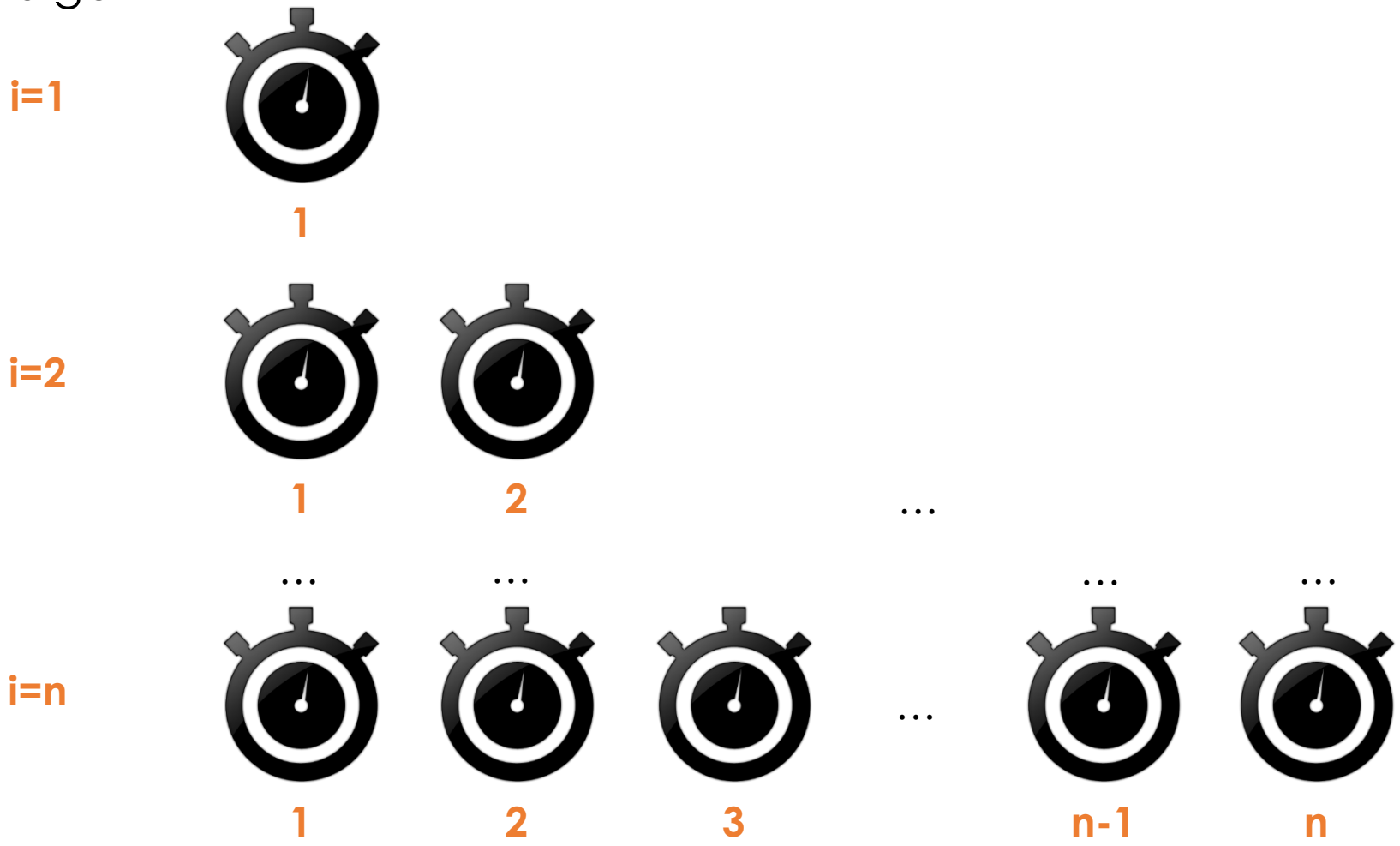
- An $O(n^2)$ algorithm

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

Picturing Efficiency

- An $O(n^2)$ algorithm

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



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

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`

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`

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 of time an algorithm requires to solve problems of size `n`
  - Very hard to estimate

ADT Bag Interface

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 getFreuqnecyOf(const ItemType& target) const = 0;
     virtual bool contains(const ItemType& anEntry) const = 0;
     virtual std::vector<ItemType> toVector() const = 0;
     virtual ~BagInterface() { }
}; // end BagInterface
```

```
template<class ItemType>
class ArrayBag : public BagInterface<ItemType>
{
private:
     static const int DEFAULT_CAPACITY = 6;
     ItemType items[DEFAULT_CAPACITY];  // bag items
     int itemCount;                     // count of bag items
```

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

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 getFreuqnecyOf(const ItemType& target) const = 0;
     virtual bool contains(const ItemType& anEntry) const = 0;
     virtual std::vector<ItemType> toVector() const = 0;
     virtual ~BagInterface() { }
}; // end BagInterface
```

```
template<class ItemType>
class ArrayBag : public BagInterface<ItemType>
{
private:
     static const int DEFAULT_CAPACITY = 6;
     ItemType items[DEFAULT_CAPACITY];  // bag items
     int itemCount;                     // count of bag items
```

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

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 getFreuqnecyOf(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 = 6;
     ItemType items[DEFAULT_CAPACITY];  // bag items
     int itemCount;                     // count of bag items
```

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

ADT Bag – ADD implementations

```
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
```

ADT Bag – ADD implementations

```
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)

ADT Bag – ADD implementations

```
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)

```
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
```

ADT Bag – ADD implementations

```
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)

```
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

```
template<class ItemType>
bool ArrayBag<ItemType>::contains 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
```

ADT Bag – CONTAINS implementations

```
template<class ItemType>
bool ArrayBag<ItemType>::contains 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
```

ADT Bag – CONTAINS implementations

```
template<class ItemType>
bool ArrayBag<ItemType>::contains 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
```

O(1)

Best-case

O(n)

Worst-case

ADT Bag – CONTAINS implementations

```
template<class ItemType>
bool ArrayBag<ItemType>::contains 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
```

O(1)

Best-case

```
template<class ItemType>
bool LinkedBag<ItemType>::contains 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
```

O(n)

Worst-case

ADT Bag – CONTAINS implementations

```
template<class ItemType>
bool ArrayBag<ItemType>::contains 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
```

O(1)

Best-case

```
template<class ItemType>
bool LinkedBag<ItemType>::contains 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
```

O(n)

Worst-case

ADT Bag – CONTAINS implementations

```
template<class ItemType>
bool ArrayBag<ItemType>::contains 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
```

O(1) Best-case

O(n) Worst-case

```
template<class ItemType>
bool LinkedBag<ItemType>::contains 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
```

O(1) Best-case

O(n) Worst-case

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)

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)

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

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

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

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

LinkedBag

```
void LinkedBag::clear()
{
      Node<ItemType>* deleteMe =
      nullptr;
      while (headPtr != nullptr)
      {
            deleteMe = headPtr;
            headPtr = headPtr-
      >getNext();
            delete deleteMe;
            itemCount--;
      } // end while
} // end clear
```

Comparing Implementations

- If problem size is always small
  - Possible to ignore algorithm's efficiency
- Weight trade-offs between
  - Algorithm's time and memory requirements
- Compare algorithms for style and efficiency

Thank you