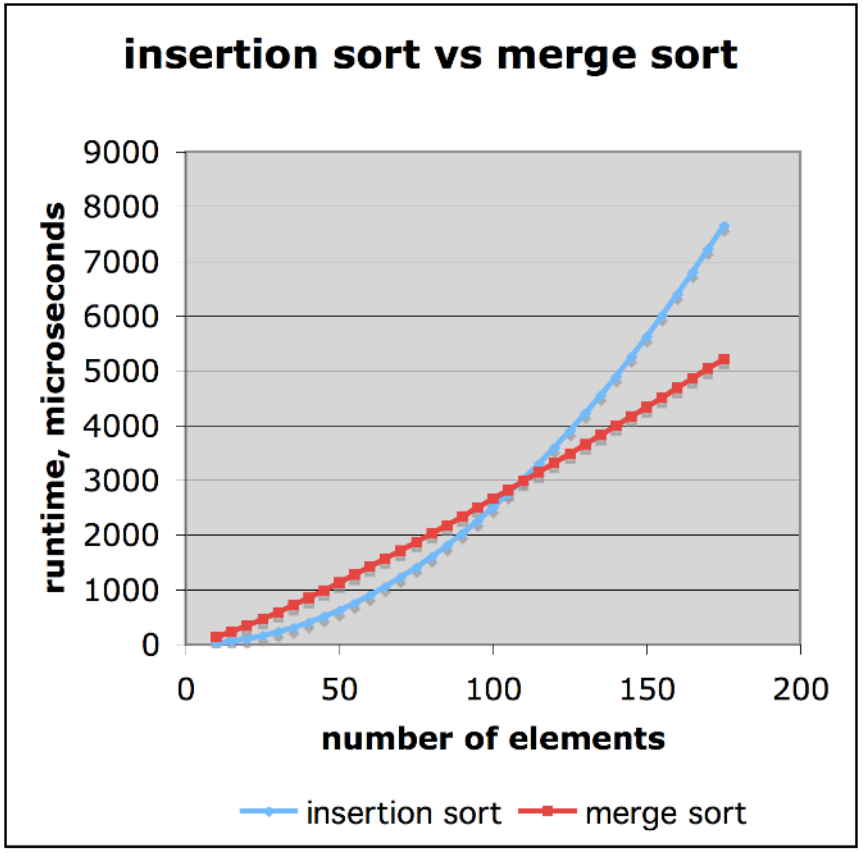
CS151

Complexity Analysis

Lists

ArrayList

Comparison of Two Algorithms



- insertion sort: $n^2/4$
- merge sort: $2nlgn$
- suppose n=10^8
  - insertion sort: $10^8*10^8/4 = 2.5*10^{15}$
  - merge sort: $10^8*26*2 = 5.2*10^9$
  - or merge sort can be expected to be about 10^6 times faster
  - so if merge sort takes 10 seconds then insertion sort takes about 100 days

Asymptotic Notation

- Provides a way to simplify analysis
- Allows us to ignore less important elements
  - constant factors
- Focus on the largest growth of n
  - Focus on the dominant term

How do these functions grow?

Big O

- Constant factors are ignored
- Upper bound on time
- Goal is to have an easily understood summary of algorithm speed
  - not implementation speed

Sublinear Algorithms

- O(1)
  - runtime does not depend on input

```
public int order1(int[] data) {
    int count=0;
    return count;
}
```

- O($lg_2 n$)
  - algorithm constantly halves (or whatever) input
    - loop index changes using * or /

```
public int orderLgN(int[] data) {
    int count=0;
    for (int i = data.length; i >= 1; i = i / 2) {
        count++;
    }
    return count;
}
```

Linear Time Algorithms: $O(n)$

- The algorithm's running time is at most a constant factor times the input size
- Process the input in a single pass spending constant time on each item
  - max, min, sum, average, linear search
- Any single loop that updates index using + - a constant amount (usually the constant is 1)

```
public int orderN(int[] data) {
    int count = 0;
    for (int i = 0; i < data.length; i++) {
        count++;
    }
    return count;
}
```

$O(nlogn)$ time

Frequent running time in cases when algorithms involve sorting

An O(n) style loop inside an O(lg n) style loop (or the converse)

- For example:
  - the "good" sorting algorithms

```
public int orderNlgN(int[] data) {
    int count = 0;
    for (int i = 1; i < data.length; i *= 2) {
        for (int j = 0; j < data.length; j++) {
            count++;
        }
    }
    return count;
}
```

Quadratic Time: $O(n^2)$

- Nested loops
  - when each loop would be O(n)
- For example
  - The `doSomething` algorithm (from timer in last class)
  - The less-good sorting algorithms
  - Processing all pairs of elements

```
public int orderNsquared(int[] data) {
    int count = 0;
    for (int i = 0; i < data.length; i++) {
        int j = i;
        while (j < data.length) {
            count++;
            j++;
        }
    }
    return count;
}
```

Slow!!!! Times

- polynomial time: $O(n^k)$
  - All subsets of n elements of size k
- exponential time: $O\left(2^n\right)$
  - All subsets of n elements (power set)
- factorial time: $O(n!)$
  - All permutations of n elements

Algorithm Run Times

N	log(n)	n	n log(n)	n*n	n*n*n	n!
10	3	10	33	100	1000	10^5
100	7	100	664	10000	10^6	10^94
1000	10	1000	9966	10^6	10^9	10^1435
10000	13	10000	132877	10^8	10^12	10^19355
100000	17	100000	1660964	10^10	10^15	10^(10^6)

Analyzing StuffBag

- add
- remove one
- count
- remove all of X

- Can these times be improved?
  - at what cost?

Lists

Lists

- A list is a bag in which the items are ordered (not sorted).
    - No empty list items allowed!
    - Position in list is not fixed, but relative order is
        - how does this statement make sense?
- Lists can GROW and shrink
    - A major difference from Array
- Actions with lists
    - Add item at location N
        - add item to beginning or end of list
    - Get Nth item
    - Change Nth item
    - Remove Nth item
    - Others from BagOfStuff
        - Number of items in list (another big difference from Array)

clear, count, empty?, contains?, display

Java Interface for List

interface extends an interface!

Generics

```
public interface List151<W> extends BagOfStuff<W> {
   boolean add(int index, W t) throws IndexOutOfBoundsException;
   void remove(int index) throws IndexOutOfBoundsException;
   W get(int index) throws IndexOutOfBoundsException;
   boolean set(int index, W t) throws IndexOutOfBoundsException;
   int indexOf(W t);
}
```

interfaces mention exceptions

From BagOfStuff

```
    public int numberOfItems();   public boolean isEmpty();
    public boolean add(S p);      public S remove();
    public boolean remove(S p);   public void clear();
    public int countOf(S p);      public boolean contains(S p);
    public void display();
```

Why throws exceptions???

- Signal to user that something went wrong and the operation failed.
  - Alternative: have a special return value that indicates failure.
    - both approaches work
    - some classes have methods for both
- By throwing an exception you force the user (of your class) to do something or have the program die. (This is fairly aggressive)

Implementing List151

- List151 looks a lot like BagOfStuff; BUT
  - Order is important
- Internally again use an array but this time need to be sure there are no empty spaces (unlike Bags)
- Also, speed matters ... implementation should be efficient

List151 indexOf(T t)

- Problem … how can you compare equality of two generics
  - The only functions you can use for a generic are those with Object.
    - We will discuss ways around this limitation later in semester

- Solution: Use equals. Document this! Then it is the users responsibility to either accept the default equals or override it appropriately

List151Impl

```
public class List151Impl<Y> implements List151<Y> {
    /** The actual number of items stored.
      * Not required, but it does speed up several operations
     */
    protected int count;
    /** The array in which all the data is actually stored */
    protected Y[] arra;
    public List151Impl() {
        this(100);
    }
    @SuppressWarnings("unchecked")
    public List151Impl(int initialCapacity) {
        arra = (Y[]) new Object[initialCapacity];
        count = 0;
    }
```

get(index)

- Tasks
  - check to see if index is valid
  - return item at index in array

```
public Y get(int index) throws IndexOutOfBoundsException {
    if (index > count) {
        throw new IndexOutOfBoundsException("Can only get where there are  already items");
    }
    if (index < 0) {
        throw new IndexOutOfBoundsException("Cannot get from a negative location");
    }
    return arra[index];
}
```

Time Complexity?

indexOf(item)

- loop through all items
  - if the provided item is equal current item stop and return index of current item
- if provided item not found return -1

```
public int indexOf(Y t) {
    for (int i = 0; i < count; i++) {
        if (arra[i].equals(t))
            return i;
    }
    return -1;
}
```

Add

- StuffBag had to look through the bag to find first space

```
@Override
public boolean add(R p) {
    int loc=0;
    while (loc < stuffArray.length &&
           stuffArray[loc] != null) {
        loc++;
    }
    if (loc == stuffArray.length){
        return false;
    } else {
        stuffArray[loc] = p;
        return true;
    }
}
```

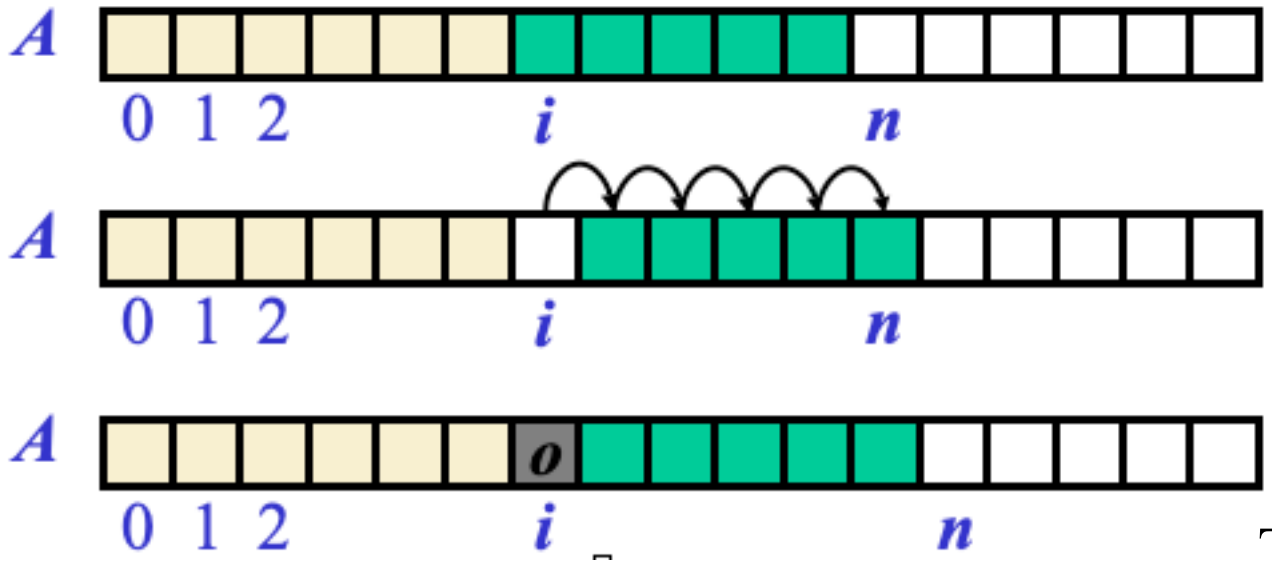
- For list151, there are no spaces, and end is known, so just add to end

```
@Override
public boolean add(Y t) {
    if (count >= arra.length)
        return false;
    arra[count] = t;
    count++;
    return true;
}
```

Time complexity of these add methods

add(int index, W t)

- Tasks
  - Check location to ensure it is valid
    - What is "valid"?
  - Make space for new item



To make a space start at nth item move it to n+1

Time Complexity?

add(int index, W t)

- live write

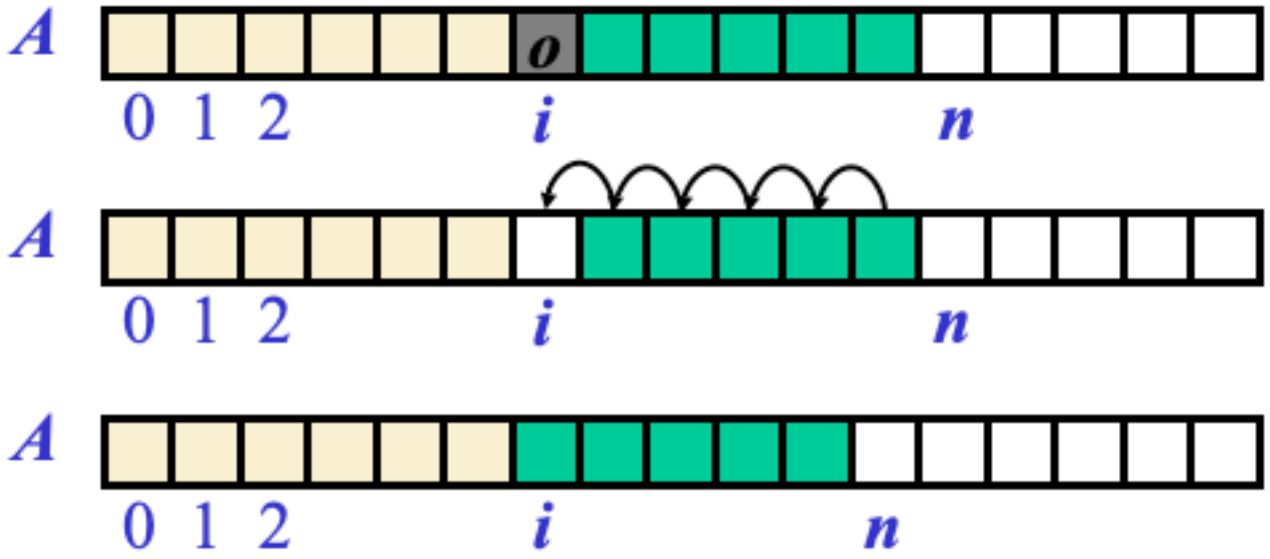
Time complexity of this add

List Growth

- When
  - What functions should be able to call
- Visible to users?
- How?

remove(index)

- Tasks
  - check to see if index is valid
  - move remaining items over to fill hole



Groups

- For the List151Impl class write
  - `remove(index)`
  - `remove(item)`

```
/**
     * Removes the element at the specified position in this list. Shifts any
     * subsequent elements to the left (subtracts one from their indices).
     *
     * @param index the index of the element to be removed
     */
    void remove(int index) throws IndexOutOfBoundsException;
```

Add -- Again

- When introduced, I said lists can grow
- If growth is possible, it should happen in add method
  - Where?
  - How?

```
@Override
public boolean add(Y t) {
    if (count >= arra.length)
        return false;
    arra[count] = t;
    count++;
    return true;
}
```

2d List151Impl

```
public class AL2d {
    public static void main(String[] args) {
        List151Impl<List151Impl<String>> al2d = new List151Impl<>();
        al2d.add(new List151Impl<String>());
        // etc
        al2d.get(0).add("Hello");
        al2d.get(0).add(1);
    }
}
```

a real mouthful!

Add an AL to the "outer" AL

add a string to the inner AL

Not legal!

Testing List151Impl

- Perfect testing would exercise and validate every line of code
  - A perfect test suite can be as hard to write as the code it is testing
  - Alternative: test-driven development
    - write the tests first, then write code that always satisfies all tests
  - Tests should be written pretending you do not have the code, but rather only a pseudocode
- Tests:
  - Construct: Make different capacities
  - Construct: Hold different object types
  - Add(item): Add 1 item? Two items, Three items (once you get to three you can assume more — kind of proof by induction.)
    - how do you know they are added?
    - Is order preserved?
  - Add(item): what happens when you run out of space?
  - Add(item): wrong type addition should be caught by compiler.
  - Add(index, item): what happens in each index of out range condition?
  - Add(index, item): what happens when there is no room to add?
  - ETC.

Test Code for List151Impl

```
    public static void main(String[] args) {
        System.out.println("\nTest A: adding consecutive integers to List151 with capacity of
10\nResult should be 0; 0,1; 0,1,2; etc");
        for (int i = 0; i < 4; i++) {
            List151Impl<Integer> test = new List151Impl<>(10);
            for (int j = 0; j <= i; j++) {
                test.add(j);
            }
            System.out.println("\n"+i+":");
            test.display();
        }

        System.out.println("\nTest B: Fill a list to capacity, then overfill");
        List151Impl<Integer> test = new List151Impl<>(10);
        for (int i = 10; i < 20; i++) {
            test.add(i);
        }
        System.out.println("Should be numbers 10..19 in positions 0..9");
        test.display();
        System.out.println("\nOverfill!!");
        for (int i = 100; i < 105; i++) {
            if (test.add(i)) {
                System.out.println("Should have returned false!!");
            }
        }
        System.out.println("Should Still be numbers 10..19 in positions 0..9");
        test.display();
    }
```