NP-complete is a complexity class
• No known polynomial time (O(n), O(n^5)…) solutions!
• Solutions are, for example, O(2^n) – ouch!
Collection efficiency

- Complexity class of various operations on collections:

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</th>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (or push)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add(index, value)</td>
<td>O(N)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>indexOf</td>
<td>O(N)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>get</td>
<td>O(1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>remove</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>set</td>
<td>O(1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>size</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

- Could we build lists differently to optimize other operations?
Non-contiguous memory

- Array

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>-3</td>
<td>17</td>
<td>9</td>
</tr>
</tbody>
</table>

- Spread in memory

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
<td>9</td>
<td>-3</td>
<td>17</td>
</tr>
</tbody>
</table>
Value semantics

- **value semantics**: Behavior where values are copied when assigned to each other or passed as parameters.
  - When one primitive is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```java
int x = 5;
int y = x;  // x = 5, y = 5
y = 17;    // x = 5, y = 17
x = 8;     // x = 8, y = 17
```
Reference semantics

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - When one reference variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.

```java
define
int[] a1 = {4, 5, 2, 12, 14, 14, 9};
int[] a2 = a1;  // refers to same array as a1
a2[0] = 7;
System.out.println(a1[0]);  // 7
```

```
<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>9</td>
</tr>
</tbody>
</table>
```
References and objects

- In Java, objects and arrays use reference semantics. Why?
  - efficiency. Copying large objects slows down a program.
  - sharing. It's useful to share an object's data among methods.

```java
drawingPanel panel1 = new DrawingPanel(80, 50);
drawingPanel panel2 = panel1; // same window
panel2.setBackground(Color.CYAN);
```

```
panel1
```

```
panel2
```
A list node class

```java
public class ListNode {
    int data;
    ListNode next;
}
```

- Each list node object stores:
  - one piece of integer data
  - a reference to another list node

- `ListNode` can be "linked" into chains to store a list of values:
Dereferencing

- **dereference**: To access data or methods of an object.
  - Done with the dot notation, such as `s.length()`
  - When you use a . after an object variable, Java goes to the memory for that object and looks up the field/method requested.

```java
Student timmy = new Student();
timmy.name = "Timmmm";
String s = timmy.name.toUpperCase();
```

```
<table>
<thead>
<tr>
<th>Student</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>'T'</td>
</tr>
<tr>
<td>id</td>
<td>'i'</td>
</tr>
<tr>
<td></td>
<td>'m'</td>
</tr>
<tr>
<td></td>
<td>'m'</td>
</tr>
<tr>
<td></td>
<td>'m'</td>
</tr>
<tr>
<td></td>
<td>'m'</td>
</tr>
</tbody>
</table>

public int indexOf(String s) {...}
public int length() {...}
public String **toUpperCase**() {...}
```
Null pointer exception

- It is illegal to dereference `null` (it causes an exception).
- `null` does not refer to any object; it has no methods or data.

```java
Student timmy = new Student();
String s = timmy.name.toUpperCase();  // ERROR
```

Output:
```
Exception in thread "main"
java.lang.NullPointerException
at Example.main(Example.java:8)
```
public class ConstructList1 {
    public static void main(String[] args) {
        ListNode list = new ListNode();
        list.data = 42;
        list.next = new ListNode();
        list.next.data = -3;
        list.next.next = new ListNode();
        list.next.next.data = 17;
        list.next.next.next = null;
        System.out.println(list.data + " " + list.next.data
                           + " " + list.next.next.data);
        // 42 -3 17
    }
}
List node w/ constructor

```java
public class ListNode {
    int data;
    ListNode next;

    public ListNode(int data) {
        this.data = data;
        this.next = null;
    }

    public ListNode(int data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}
```

- Exercise: Modify the previous client to use these constructors.
Linked node problem 1

- What set of statements turns this picture:

  list

  ![Diagram 1](data 10 next)
  ![Diagram 2](data 20)

- Into this?

  list

  ![Diagram 3](data 10 next)
  ![Diagram 4](data 20)
  ![Diagram 5](data 30)
References vs. objects

`variable = value;`

- A `variable` (left side of `=`) is an arrow (the base of an arrow)
- A `value` (right side of `=`) is an object (a box; what an arrow points at)

- For the list at right:
  - `a.next = value;`
    means to adjust where points
  - `variable = a.next;`
    means to make `variable` point at
Reassigning references

- When you say:
  - `a.next = b.next;`

- You are saying:
  - "Make variable `a.next` refer to the same value as `b.next`."  
  - Or, "Make `a.next` point to the same place that `b.next` points."
Linked node problem 2

- What set of statements turns this picture:

  ![Diagram of a linked list with nodes data=10 and data=20]

- Into this?

  ![Diagram of a linked list with nodes data=30, data=10, and data=20]
Linked node problem 3

- What set of statements turns this picture:

```
list1 ─── data ─── next ─── data ─── next
  10                                    20

list2 ─── data ─── next ─── data ─── next
  30                                    40
```

- Into this?

```
list1 ─── data ─── next ─── data ─── next ─── data ─── next
  10                                    20                                    30

list2 ─── data ─── next
  40
```
Linked node problem 3

- How many ListNode variables?

```
list1  →
①
② data  10
      next
③ data  20
      next

list2  →
④
⑤ data  30
      next
⑥ data  40
      next
```

- Which variables change?

```
list1  →
① data  10
      next
② data  20
      next
⑤ data  30
      next

list2  →
④ data  40
      next
```

1  2  3  4  5  6
Linked node problem 4

• What set of statements turns this picture:

```
<table>
<thead>
<tr>
<th>list</th>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
```

```
|      | 990  |      |
```

• Into this?

```
<table>
<thead>
<tr>
<th>list</th>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
```

```
|      | 990  |      |
```

```
|      | 1000 |      |
```