CSE 143
Lecture 14: binary search and complexity
reading: 13.1-13.2

I heard Java is an exceptional language.
I know, the exception hierarchy is awful.
That's not the object of what I was saying.
Oh, don't be so primitive.

http://www.alexsweet.co.uk/comics.php?comic=2
Searching methods

- Implement the following methods in ArrayIntList:
  - `indexOf` – returns first index of element, or -1 if not found
  - `contains` - returns true if the list contains the given int value
Sequential search

- **sequential search**: Locates a target value in an array / list by examining each element from start to finish. Used in `indexOf`.
  
  - How many elements will it need to examine?
  
  - Example: Searching the array below for the value **42**:

<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>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

  - The array is sorted. Could we take advantage of this?
Sequential search

• What is its complexity class?

```
public int indexOf(int value) {
    for (int i = 0; i < size; i++) {
        if (elementData[i] == value) {
            return i;
        }
    }
    return -1; // not found
}
```

• On average, "only" N/2 elements are visited
  • 1/2 is a constant that can be ignored
Binary search (13.1)

- **binary search**: Locates a target value in a *sorted* array or list by successively eliminating half of the array from consideration.

  - How many elements will it need to examine?
  - Example: Searching the array below for the value 42:

<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>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

min   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
mid   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
max   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
Binary search runtime

- For an array of size $N$, it eliminates $\frac{1}{2}$ until 1 element remains.
  
  $N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, ..., 4, 2, 1$

  - How many divisions does it take?

- Think of it from the other direction:
  
  - How many times do I have to multiply by 2 to reach $N$?
    
    $1, 2, 4, 8, ..., \frac{N}{4}, \frac{N}{2}, N$
  
  - Call this number of multiplications "$x$".

  
  $2^x = N$

  $x = \log_2 N$

- Binary search is in the **logarithmic** complexity class.
### Complexity classes

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size $N$.

<table>
<thead>
<tr>
<th>Class</th>
<th>Big-Oh</th>
<th>If you double $N$, ...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>unchanged</td>
<td>10ms</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log_2 N)$</td>
<td>increases slightly</td>
<td>175ms</td>
</tr>
<tr>
<td>linear</td>
<td>$O(N)$</td>
<td>doubles</td>
<td>3.2 sec</td>
</tr>
<tr>
<td>log-linear</td>
<td>$O(N \log_2 N)$</td>
<td>slightly more than doubles</td>
<td>6 sec</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(N^2)$</td>
<td>quadruples</td>
<td>1 min 42 sec</td>
</tr>
<tr>
<td>cubic</td>
<td>$O(N^3)$</td>
<td>multiplies by 8</td>
<td>55 min</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
<td>multiplies drastically</td>
<td>$5 \times 10^{61}$ years</td>
</tr>
</tbody>
</table>
Complexity classes

The `binarySearch` method in the `Arrays` class searches an array very efficiently if the array is sorted.

- You can search the entire array, or just a range of indexes (useful for "unfilled" arrays such as the one in `ArrayIntList`)
Using `binarySearch`

```java
// index    0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
int[] a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92};
int index  = Arrays.binarySearch(a, 0, 16, 42); // index1 is 10
int index2 = Arrays.binarySearch(a, 0, 16, 21); // index2 is -7
```

- `binarySearch` returns the index where the value is found
- if the value is not found, `binarySearch` returns:
  
  ```java
  -(insertionPoint + 1)
  ```

  - where `insertionPoint` is the index where the element would have been, if it had been in the array in sorted order.
  - To insert the value into the array, negate `insertionPoint + 1`

  ```java
  int indexToInsert21 = -(index2 + 1); // 6
  ```
public static int binarySearch(int[] a, int target) {
    int min = 0;
    int max = a.length - 1;

    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid] < target) {
            min = mid + 1;
        } else if (a[mid] > target) {
            max = mid - 1;
        } else {
            return mid; // target found
        }
    }

    return -(min + 1); // target not found
}
Recursive binary search (13.3)

- Write a recursive `binarySearch` method.
  - If the target value is not found, return its negative insertion point.

<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>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

```java
int index  = binarySearch(data, 42);  // 10
int index2 = binarySearch(data, 66);  // -14
```
The `compareTo` method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
  - Example: in the `String` class, there is a method:
    ```java
    public int compareTo(String other)
    ```
  - A call of `A.compareTo(B)` will return:
    - a value < 0 if `A` comes "before" `B` in the ordering,
    - a value > 0 if `A` comes "after" `B` in the ordering,
    - or 0 if `A` and `B` are considered "equal" in the ordering.
Using `compareTo`

- `compareTo` can be used as a test in an `if` statement.

```java
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {  // true
    ...
}
```

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a &lt; b) { ...</td>
<td>if (a.compareTo(b) &lt; 0) { ...</td>
</tr>
<tr>
<td>if (a &lt;= b) { ...</td>
<td>if (a.compareTo(b) &lt;= 0) { ...</td>
</tr>
<tr>
<td>if (a == b) { ...</td>
<td>if (a.compareTo(b) == 0) { ...</td>
</tr>
<tr>
<td>if (a != b) { ...</td>
<td>if (a.compareTo(b) != 0) { ...</td>
</tr>
<tr>
<td>if (a &gt;= b) { ...</td>
<td>if (a.compareTo(b) &gt;= 0) { ...</td>
</tr>
<tr>
<td>if (a &gt; b) { ...</td>
<td>if (a.compareTo(b) &gt; 0) { ...</td>
</tr>
</tbody>
</table>