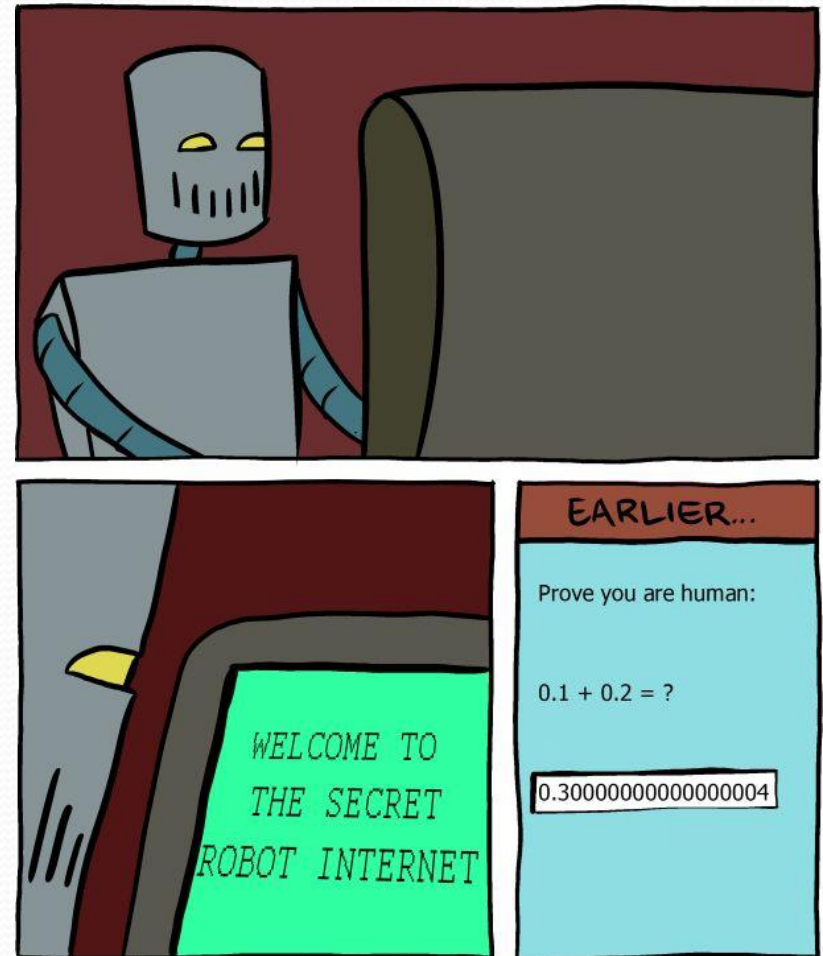


# CSE 143

Lecture 15: binary search  
reading: 13.1-13.2



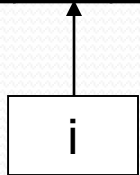
# Searching methods

- The `List` interface contains the following methods:
  - `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**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



- 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  
}
```

} N

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

- 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**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

Diagram illustrating a binary search on a sorted array. The array is shown with indices 0 to 16 and corresponding values. The value 42 is highlighted in yellow at index 10. Below the array, three boxes labeled 'min', 'mid', and 'max' are shown with arrows pointing to the corresponding indices: 'min' points to index 0, 'mid' points to index 8, and 'max' points to index 16.

# Binary search runtime

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

- 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  $N$ ?

$1, 2, 4, 8, \dots, N/4, N/2, N$

- Call this number of multiplications " $x$ ".

$$2^x = N$$

$$x = \log_2 N$$

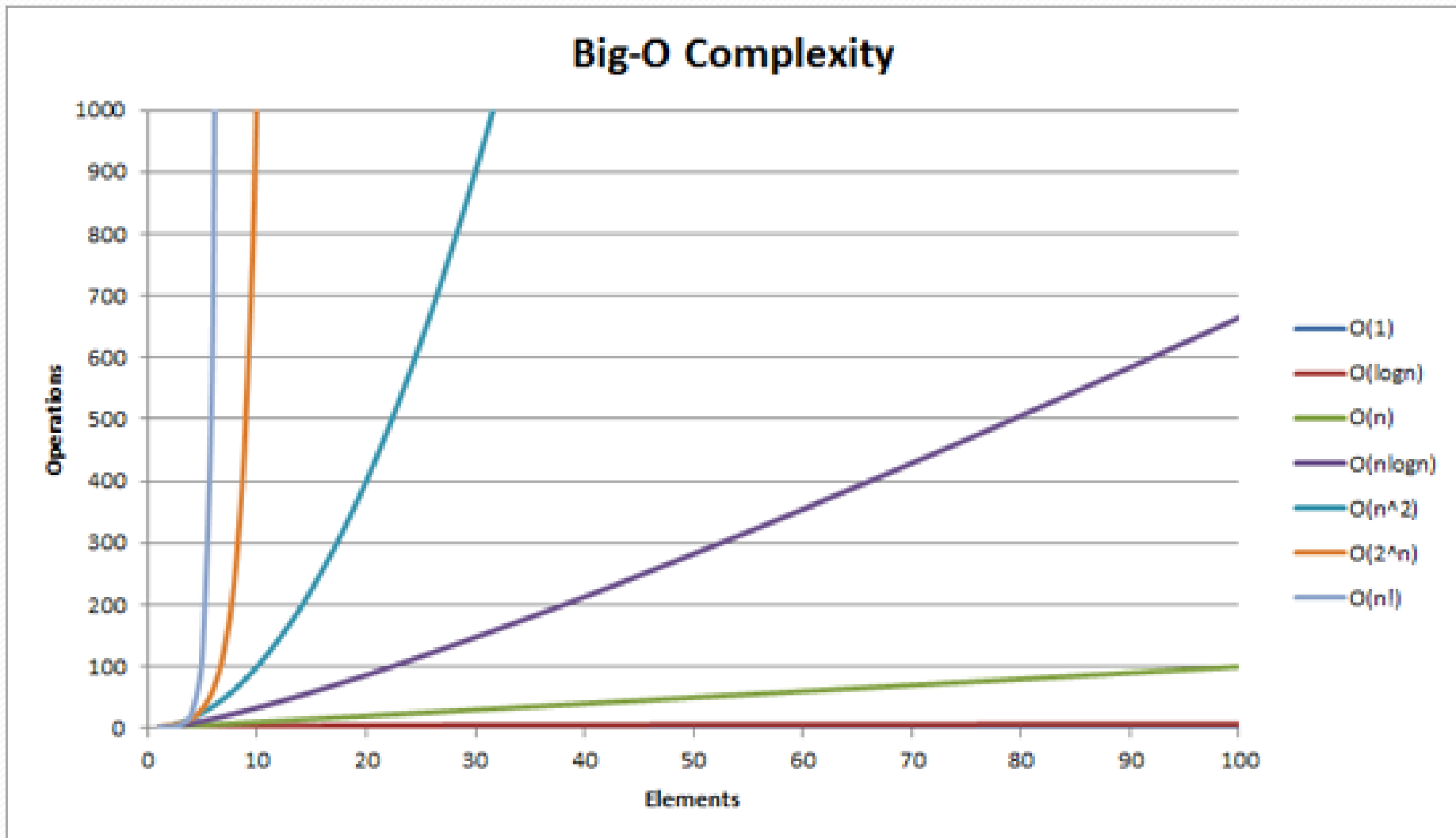
- Binary search is in the **logarithmic** complexity class.

# Complexity classes

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

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

# Complexity classes





# Arrays.binarySearch

```
// searches an entire sorted array for a given value
// returns its index if found; a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, value)
```

```
// searches given portion of a sorted array for a given value
// examines minIndex (inclusive) through maxIndex (exclusive)
// returns its index if found; a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, minIndex, maxIndex, value)
```

- 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`

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

```
int indexToInsert21 = -(index2 + 1); // 6
```

# Binary search code

```
// Returns the index of an occurrence of target in a,  
// or a negative number if the target is not found.  
// Precondition: elements of a are in sorted order  
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.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

```
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:

```
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.

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

Primitives	Objects
if (a < b) { ...	if (a.compareTo(b) < 0) { ...
if (a <= b) { ...	if (a.compareTo(b) <= 0) { ...
if (a == b) { ...	if (a.compareTo(b) == 0) { ...
if (a != b) { ...	if (a.compareTo(b) != 0) { ...
if (a >= b) { ...	if (a.compareTo(b) >= 0) { ...
if (a > b) { ...	if (a.compareTo(b) > 0) { ...