Sorting, Searching and Array's Built-in Methods

Lecture 16 – COMP110 – Spring 2018

Warm-up 1: What are the elements of **a**?

```
let a: number[] = [ 2 ]; // Notice initial element 2
let i: number = 0;

while (i < 3) {
    a[a.length] = (i + 1) * 2;
    i++;
}

print(a);
```

How do we **append** an element to an array?

- Given an array **a**, what is the **next** index needed to append?
  - When it is **empty**, or has **0 elements**, the next index is **0**
  - When it has **1 element**, the next index is **1**
  - When it has **2 elements**, the next index is **2**

- **Because of 0-based indexing, we can use the # of elements in an array as the index to use to append a value to the array.**

- Append to an array:

```
a[a.length] = <value>;
```

Warm-up 2: What are the elements of array **numbers** in the **main** function that are printed?

```
let a = [3, 9, 4, 2];

let i = 2;
let temp = a[i];
a[i] = a[i - 1];
a[i - 1] = temp;

print(a);
```

World's 2nd Worst Magic Trick

Follow-along: Sort Numbers Ascending

- Open 00-comparator-sort-app.ts

- Notice we are importing:
  - Interface: Comparator
  - Constants: A_BEFORE_B, A_SAME_AS_B, A_AFTER_B

- Take a look at the **ascending** comparator function implementation

- Let's sort the data using the **ascending** comparator

```
a.sort(ascending);
print("Ascending: " + a);
```

The **Comparator<T>** Functional Interface

- A functional interface for **comparing two objects of type T** for *sorting* and *searching*
- Its signature is:

```
(a: T, b: T): number;
```

- i.e. an implementation of Comparator<WeatherRow>:

```
let ascending: Comparator<number> = (a: number, b: number): number {
    // TODO: Compare a and b
}
```

- A Comparator must Return:
  - A **negative** number when **a** comes **before** **b** **(-1)**
  - A **positive** number when **a** comes **after** **b** **(+1)**
  - **Zero** when a is the same as b and their order doesn't matter

Use Constants to avoid "Magic Numbers"

- A **magic number** is a nameless, literal value in code like comparator's -1 or 1
  - Hard to remember what they mean! "What does this -1 mean here again?"
  - Causes larger projects to become more difficult to maintain.

- Best practice: **Define constants to give meaning to magic numbers.**
  - A constant is just a variable whose value cannot be reassigned

- Here's how you declare a constant in TypeScript:

```
const <NAME>: <type> = <value>;
```

- It is conventional to name constants in ALL_UPPERCASE_LETTERS and separate words with _'s

```
const A_BEFORE_B: number = -1;
```

`Array`'s **`sort`** method

- Every array of type **`T[]`** has a method named **sort**
- Here's its signature:

```
T[] sort(Comparator<T> comparator)
```

- Usage: **`<array>.sort(<comparator>)`**
- If we call **`sort`** on an array object, and tell it how to compare any two elements using a **`Comparator`** function, the array will be sorted for us!
- **Important:** `sort` **modifies** the array's elements in-place. It does not make a new array.

Sorting Order

- Notice that `ascending` is sorting Lowest to Highest
- How could we implement `descending` to sort in Highest to Lowest
- What's the difference?
- A Comparator's logic decides whether sorting is ascending/descending

Hands-on: Sort in **Descending Order**

1. In 00-comparator-sort-app.ts find the `descending` function

2. Implement its logic such that:
   1. When **a** is larger than **b**, the function will return A_BEFORE_B (descending!)
   2. When **a** is smaller than **b**, return A_AFTER_B
   3. Otherwise, return A_SAME_AS_B

3. From 00-comparator-sort-app.ts, at TODO #2, sort the data using your `descending` comparator and print out the data.

4. Check-in when you see the numbers sorting correctly.

```
let descending: Comparator<number> = (a, b) => {
    if (a < b) {
        return A_AFTER_B;
    } else if (a > b) {
        return A_BEFORE_B;
    } else {
        return A_SAME_AS_B;
    }
};
```

```
// TODO #2
a.sort(descending);
print("Descending: " + a);
```

PollEv: What is printed when this code runs?

```
let main = async () => {
    let numbers: number[] = [101, 110, 401, 110, 110, 110];
    print(includes(numbers, 110));
};

let includes = (haystack: number[], needle: number): boolean => {
    let i = 0;
    while (i < haystack.length) {
        if (haystack[i] === needle) {
            return true;
        }
        i++;
    }
    return false;
};

main();
```

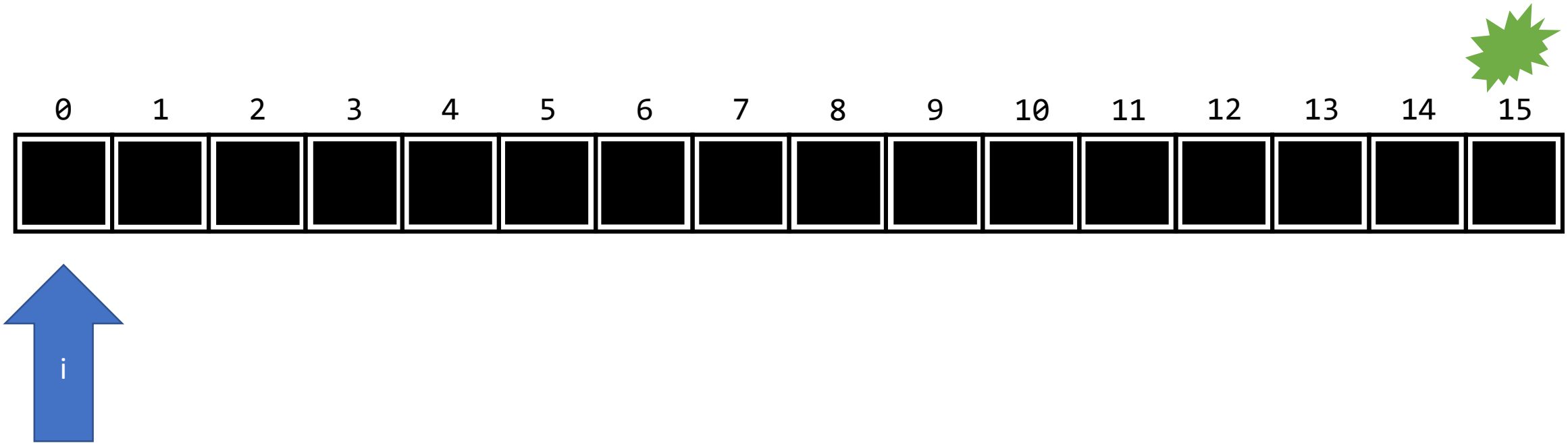
Switching gears: Searching

Linear Search

- Start from one end of an array "haystack".

- Visit each element one-by-one until you find your "needle".
  - Made past the end? No match!

The **Linear Search** Algorithm



Does the word “Yes” exist in this array of Strings?

Follow-Along: Let's Implement Linear Search

- Open 01-linear-search-app.ts
- We'll be working with the CSV file words.csv which has 77,000 words in it
- We'll implement the linearSearch function together

```
let linearSearch = (haystack: string[], needle: string): boolean => {

    let i = 0;
    while (i < haystack.length) {
        // Increment the global variable counting # of comparisons by 1
        comparisons++;

        // If haystack element i is equal to needle, then return true
        if (haystack[i] === needle) {
            return true;
        }
        i++;
    }

    return false;

};
```

How many **steps** does it take to find a word using a **Linear Search** algorithm?

- If we ran this with enough words selected at random, you would expect it takes on average: **`words.length / 2`** or **`N / 2`**
    - Why? Some words will be found in few steps the first half of the list, and others will be found in many steps in the second half of the list.
- When evaluating *runtime characteristics* of an algorithm, as computer scientists we tend to waive our hands and *approximate*.
- We classify linear search as an $O(N)$ algorithm using "big oh" notation.
    - "Given a search space of N items, this algorithm will complete in ***at most*** N steps."

Can we do better?

- Is this how *you* would find a word in a dictionary that *starts with* another word?
  - Start with “aardvark” and scan your finger through each word until matching.
- What is it about a dictionary helps us do better?
  - Dictionaries are ***sorted***!

Aside: Number Guessing Game

Introducing: Binary Search

- **<u>Requirement:</u> Elements must be <u>SORTED</u>!**

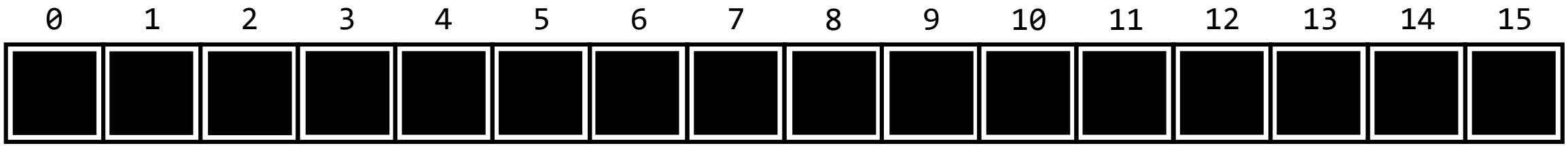
- Why are sorting algos important?
  So we can *search* efficiently!

- Algo: Start in the middle, compare with what we're looking for:
  - Too big? Look only at the smaller half.
  - Too small? Look only at the larger half.

- Intuition: at every step we cut the search space in half...

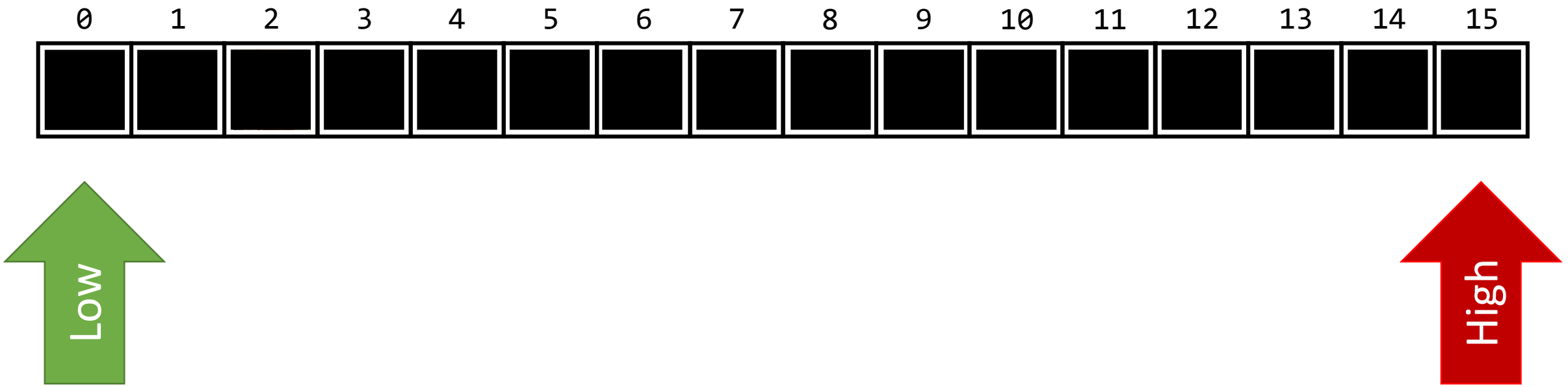
Step	Numbers Left
0	64
1	32
2	16
3	8
4	4
5	2
6	1

The **Binary Search** Algorithm



Does the word “Folt” exist in this array of Strings?

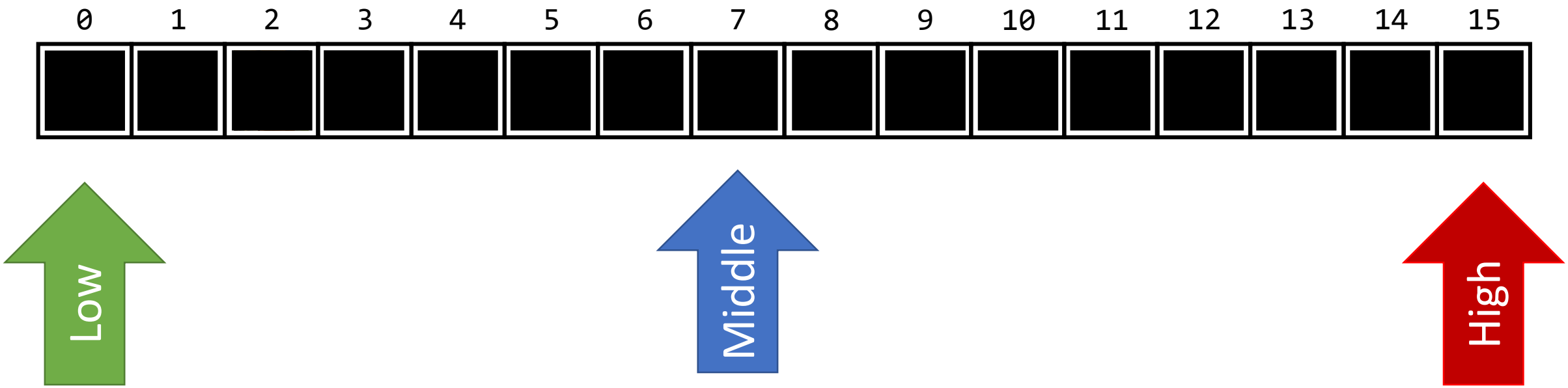
The **Binary Search** Algorithm



Looking for: “Folt”

low	0
high	15
middle	?

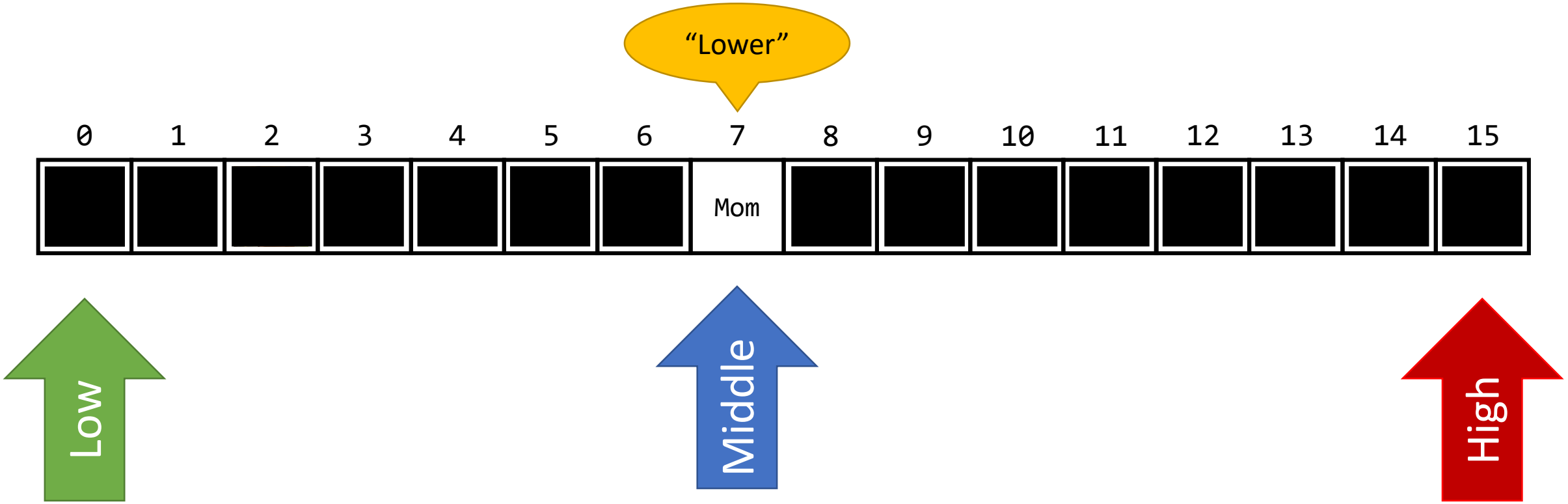
The **Binary Search** Algorithm



Looking for: “Folt”

low	0
high	15
middle	7

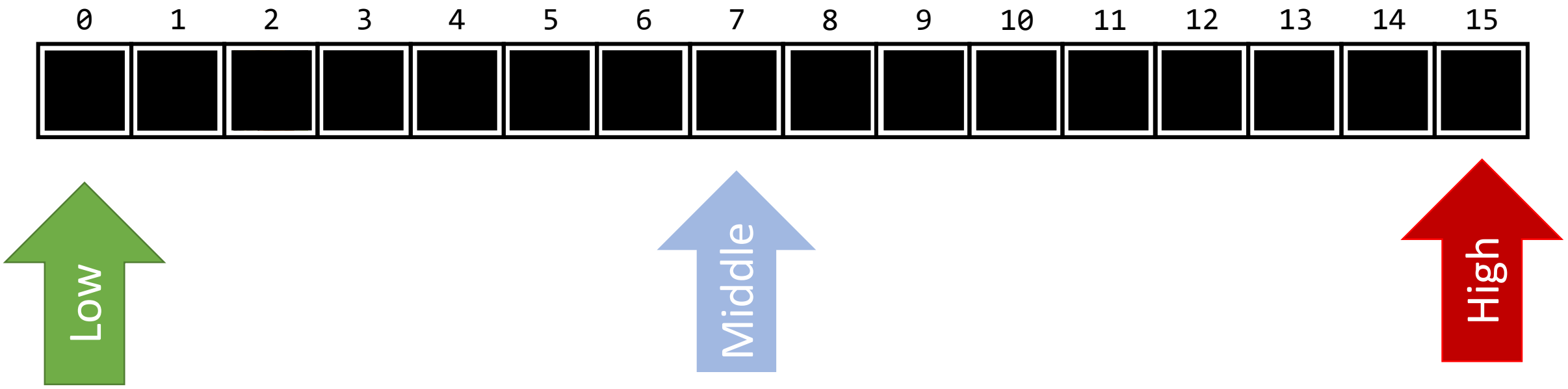
The **Binary Search** Algorithm



Looking for: "Folt"

low	0
high	15
middle	7

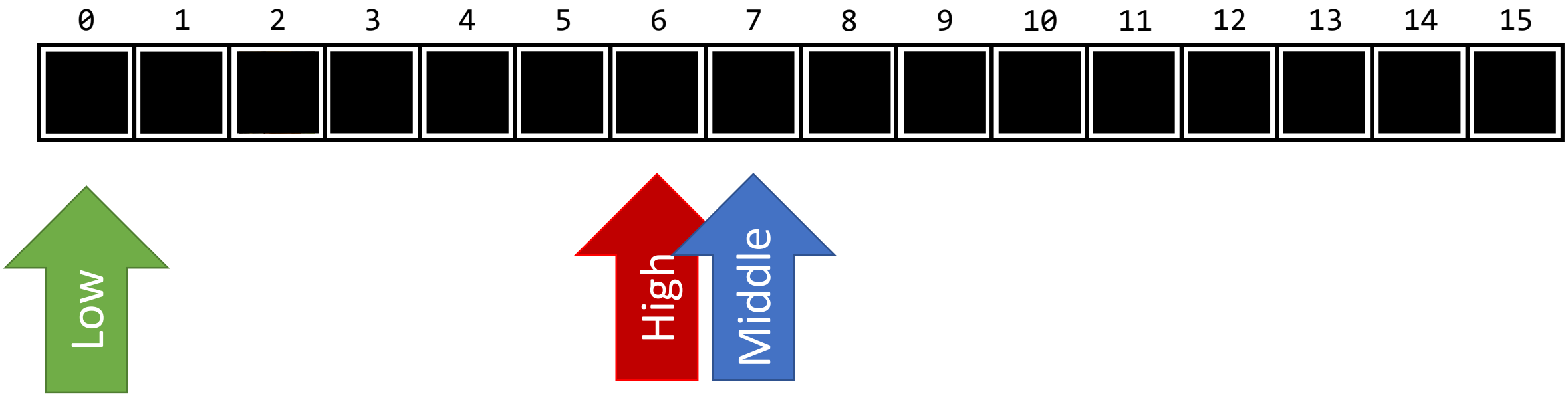
The **Binary Search** Algorithm



Looking for: “Folt”

low	0
high	6
middle	?

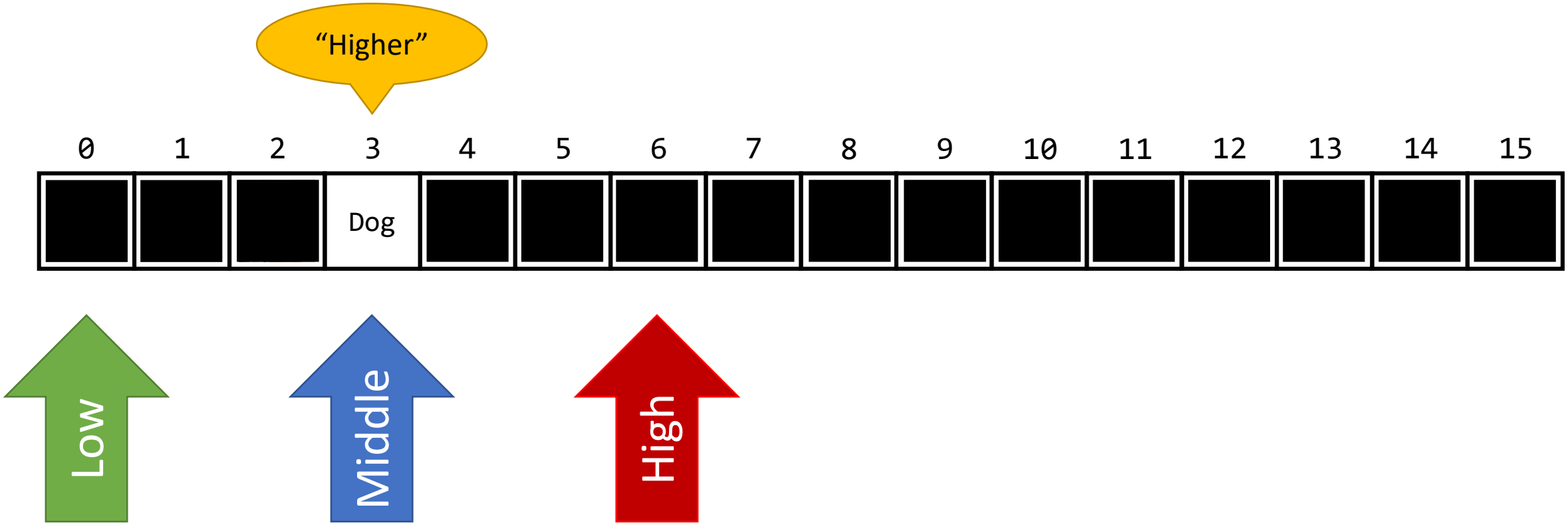
The **Binary Search** Algorithm



Looking for: “Folt”

low	0
high	6
middle	3

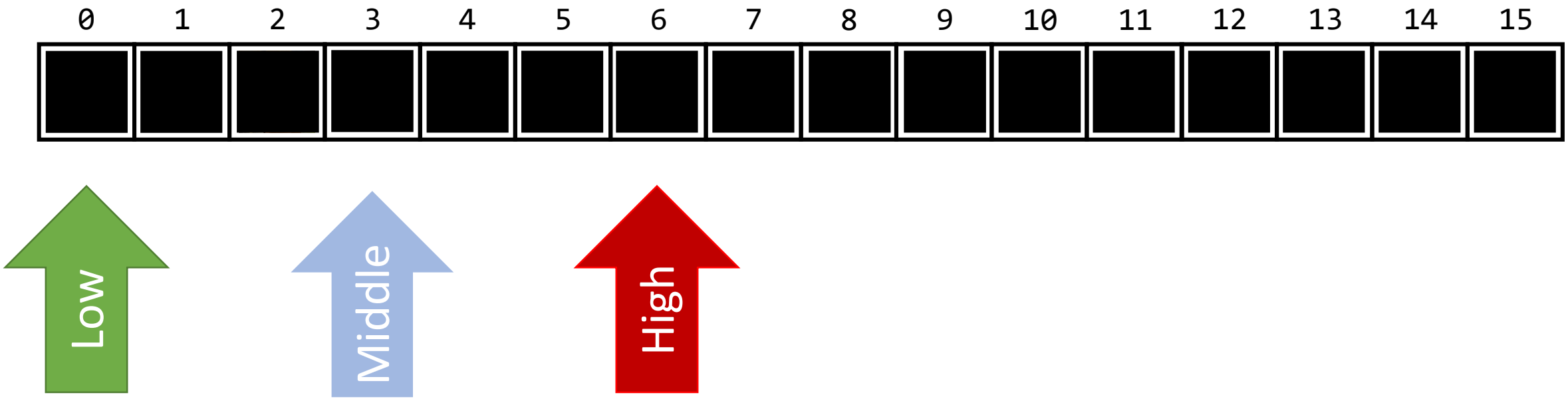
The **Binary Search** Algorithm



Looking for: "Folt"

low	0
high	6
middle	3

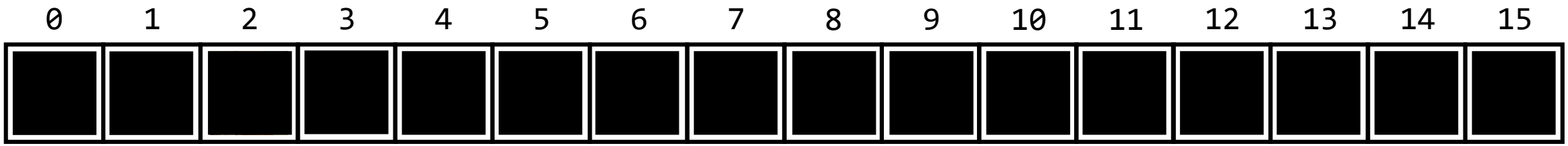
The **Binary Search** Algorithm



Looking for: “Folt”

low	4
high	6
middle	3

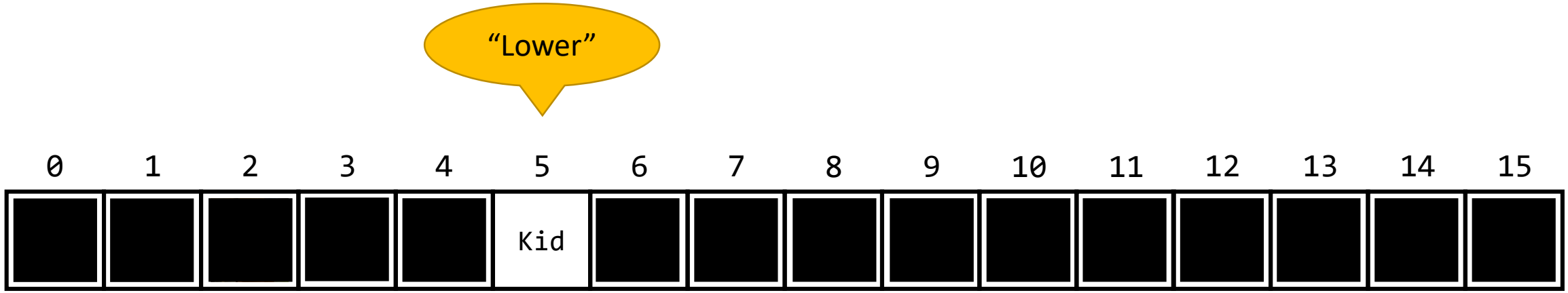
The **Binary Search** Algorithm

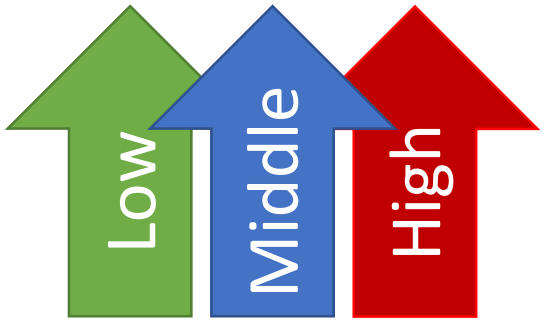


Looking for: “Folt”

low	4
high	6
middle	5

The **Binary Search** Algorithm

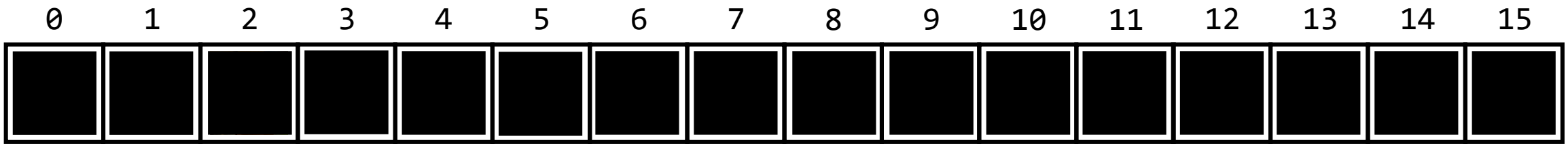


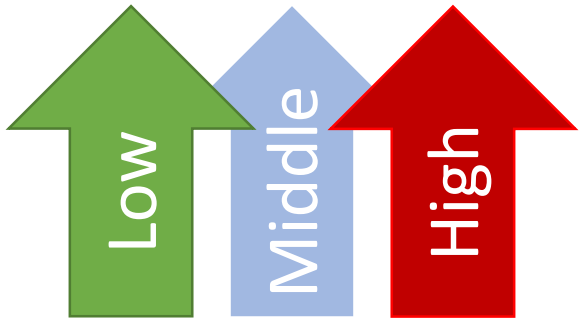


Looking for: "Folt"

low	4
high	6
middle	5

The **Binary Search** Algorithm

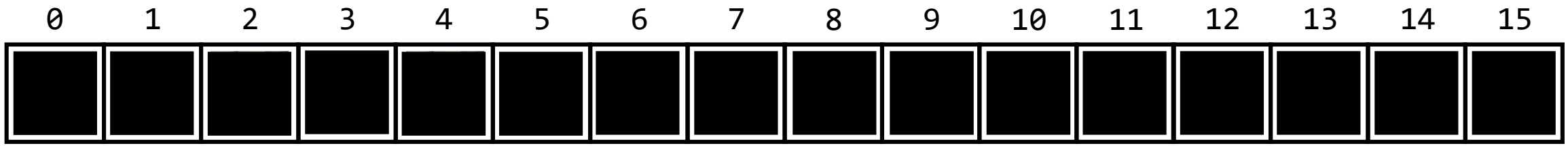


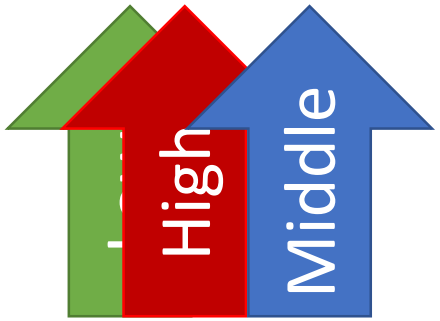


Looking for: “Folt”

low	4
high	4
middle	?

The **Binary Search** Algorithm

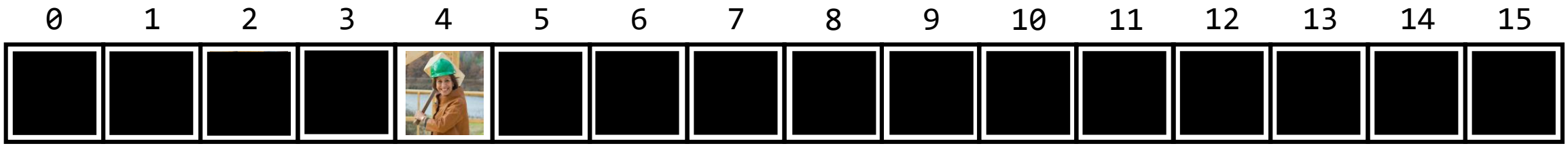


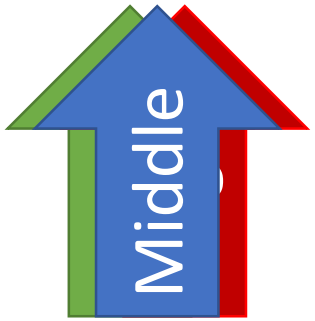


Looking for: “Folt”

low	4
high	4
middle	4

The **Binary Search** Algorithm





Looking for: “Folt”

low	4
high	4
middle	4

How do we calculate the next middle?

Low	High	Middle
0	15	7
0	6	3
4	6	5
4	4	4

```
middle = Math.floor((low + high) / 2)
```

Calling the Math.floor method will cause any decimal value to always be rounded down. This winds up being useful because arrays are 0-indexed.

Imagine the case of the 2-element array. Low would be 0, high would be 1, and so the first middle would be: Math.floor(1 / 2). We choose to try 0 first.

Hands-on: Implement Binary Search

- Open *02-binary-search-app.ts*

1. At the TODO in the `binarySearch` function, declare a variable named order and assign it the result of calling the **`compare`** function with **`needle`** and **`haystack[middle]`**

2. Delete the line that says `low++`

3. Add conditional if-then logic that implements the following logic:

When `order` is equal to…	Then
A_BEFORE_B	Set high to be middle minus 1
A_AFTER_B	Set low to be middle plus 1
A_SAME_AS_B	We've found the word! return true

4. Try searching for words and noting how many comparisons it takes. Check-in.

```
let binarySearch = (haystack: string[], needle: string, compare: Comparator<string>): boolean => {
    let low = 0;
    let high = haystack.length - 1;

    while (low <= high) {
        let middle = Math.floor((low + high) / 2);

        comparisons++; // Count this comparison

        let order = compare(needle, haystack[middle]);

        if (order === A_BEFORE_B) {
            high = middle - 1;
        } else if (order === A_AFTER_B) {
            low = middle + 1;
        } else {
            return true;
        }
    }
    // Needle was not found!
    return false;
};
```

How many **steps** does it take to find a word using the **Binary Search** algorithm?

- If we ran this with enough words selected at random, you would expect it takes ~
  $\log_2$(`words.length`)

- We classify binary search as an **O($\log_2$(N))**

  - "Given a search space of N, this algorithm will complete in **$\log_2$**(N) steps."

  - The best algorithms in computer science tend to have logarithmic solutions.

Step	Words Left
0	77,475
1	38,737
2	19,368
3	9,684
4	4,842
5	2,421
6	1,210
7	605
8	302
9	151
10	75
11	37
12	18
13	9
14	4
15	2
16	1

If we had an array with every one of the **7 billion humans** on Earth's names arranged in order, **how many steps** would it take to find one using binary search?

- 33 steps at most.
- 2^32 is 4.2 billion
- 2^33 is 8.5 billion

Array's `filter`, `map`, and `reduce` Methods

- Like `string` values, arrays have built-in methods
- We've used array's *sort* method today and seen how we can pass a Comparator<T> function to the *sort* method and it acted on the array
- Arrays also have three other built-in methods which should be familiar at this point… filter, map, and reduce.

Array's **filter** Method

- Every array of type **T[]** has a **filter** method.

- The **filter** method has a single parameter: a **Predicate<T>** of the same type **T**

- For example:

```
let a = [-1, 0, 1, 2];
let b = a.filter((x) => x > 0);
print(b); // Prints: 1, 2
```

- Calling the **filter** method on array **a** will return a new array of type **T**.
  The filter method tests all elements in the original array *using the Predicate<T>*.
  Elements that return true will be copied to the returned array.

Array's **map** Method

- Every array of type **T[]** has a **map** method.

- The **map** method has a single parameter: a **Transform<T, U>** of the same type **T**
  - The **map** method will return an array of type **U[]**

- For example:

```
let a = ["one", "two", "three"];
let b = a.map((s) => s.length);
print(b); // Prints: 3, 3, 5
```

- Calling the **map** method on array **a** will return a new array of type **U[]**.
  The map method transforms all elements in the original array *using the Transform<T,U>*.
  All transformed elements are copied to the returned array in the same order.

Array's **reduce** Method

- Every array of type **T[]** has a **reduce** method.

- The **reduce** method has two parameters:
  1. a **Reducer<T, U>** of the same type **T**
  2. An initial **memo** ("memory" accumulator) value of type **U**

- For example:

```
let a = ["one", "two", "three"];
let b = a.reduce((memo, x) => memo + x, 0);
print(b); // Prints: 11
```

- Calling the **reduce** method on array **a** will return a single value of type **U**. Starting with the initial **memo** parameter, it will call the reducer with memo and each element in **a** successively replacing memo's value with the reducer's returned value. The final **memo** value is returned.

PollEv: What is the output?

```
let xs = [1, 2, 3, 4];

let ys = xs.filter((x) => x % 2 === 1)
           .map((x) => x * 2)
           .reduce((memo, x) => memo + x);

print(ys);
```

Chaining Method Calls (1 / 5)

- Arrays methods **filter** and **map** each returns another array… and with method call chaining, we can directly call another method on the array it returns! Let's walk through how this gets evaluated…

```
let numbers = [10, 21, 30];

let result = numbers.filter((x) => x % 2 === 0)
                    .map((x) => x * 2)
                    .reduce((memo, x) => memo + x);
```

Chaining Method Calls (2 / 5)

- At runtime, first, the processor will evaluate numbers to resolve to the array [11, 22, 31]. Then the *filter* method will be called on this array.

```
let numbers = [10, 21, 30];

let result = [10, 21, 30].filter((x) => x % 2 === 0)
                         .map((x) => x * 2)
                         .reduce((memo, x) => memo + x);
```

Chaining Method Calls (3 / 5)

- The *filter* method will return the array [10, 30], because it is applying a Predicate that returns true when a number is even, and this value replaces the method call. Next, the map method will be called on [10,30]

```
let numbers = [10, 21, 30];

let result = [10, 30].map((x) => x * 2)
                     .reduce((memo, x) => memo + x);
```

Chaining Method Calls (4 / 5)

- The return value of the map method was [20, 60] because its Transform doubled each of the elements. Finally, the processor needs to evaluate this reduce method call which is summing the elements.

```
let numbers = [10, 21, 30];

let result = [20, 60].reduce((memo, x) => memo + x);
```

Chaining Method Calls (5 / 5)

- And so a final value of 80 is assigned to result.

```
let numbers = [10, 21, 30];

let result = 80;
```

Follow-along:
Array's filter/map/reduce Methods and Chaining