CSE 12 Week 4 Discussion

4-23-21

Focus: PA4, Counting Step & Runtime Analysis

Reminders

- PA4 is an **open** assignment - collaborate!

- PA1 Resubmission due today (Friday, April 23rd 11:59 PM)!
- PA2 Resubmission due Friday, April 30th 11:59 PM, and it's **open for collaboration**!
- PA3 Resubmission due Friday, May 7th 11:59 PM

PA4 Overview

Two Parts

Part 1: Questions

- Big-O Justification
- Analysis of ArrayStringList and LinkedStringList
- 6 Mystery Functions: Determe big-Θ, measure implementations (in part 2) and match

Part 2: Code

- Write a program to measure the mystery methods
- Matches the methods to the source given
- Generate graphs to justify

Only 16/70 points are autograded, don’t rely on resubmission for this assignment!

How to measure runtime in Java

- Remember to do two things to ensure your measurements are as accurate as possible:
  - Turn off Java compiler optimizations
  - Call each method once (dummy call) before calling them to measure their runtimes; the timing of the first call can be noisy and inaccurate
- How to turn off optimization in Eclipse (from the write up; scroll to last page):
  - https://docs.google.com/document/d/1vwckO76TrBT8B5E4xQ2-v2OXncLa6SQWuaQkNZaCPB0/edit
- How to turn off optimization in terminal
  - add in the flag in your **`javac`** and **`java`** commands. Examples:

    ```
    java -Djava.compiler=NONE myClass
    ```
- Example code from discussion on how to calculate runtime of a method using **`System.nanoTime()`** will be posted on the course Github

Counting Steps...

How many times does the following loop run?

```
for (int i = 1; i < 1000; i*=2) {
        System.out.println(i);
}
```

A. 100
B. 50
C. 25
D. 10

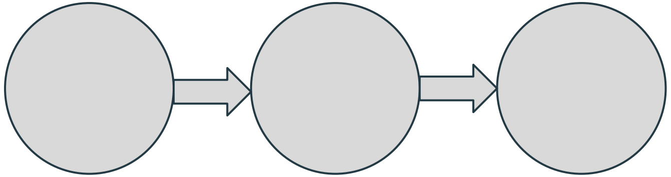
Answer - D

Loop index increments by a factor of 2 each iteration:

i = 1...2...4...8...16...32...64...128...256...512 **(10 times total)**

And fails to run when i = 1024 since i > 1000

Counting Steps...

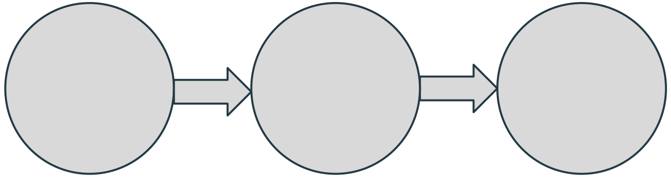


```
//LINKED LIST THAT HOLDS A STRING IN EACH NODE
boolean find(String toFind) {
  Node current = this.front.next;
  while(current != null) {
    if(current.value.equals(toFind)) {return true;}
    current = current.next;
  }
  return false;
}
```

We have a Linked List containing 10 nodes (including the dummy node). If we are searching for “Jerry” using find() but the Linked List does not contain “Jerry”, how many times will the while loop condition execute?

A. 9
B. 1
C. 3
D. 10

Answer - D



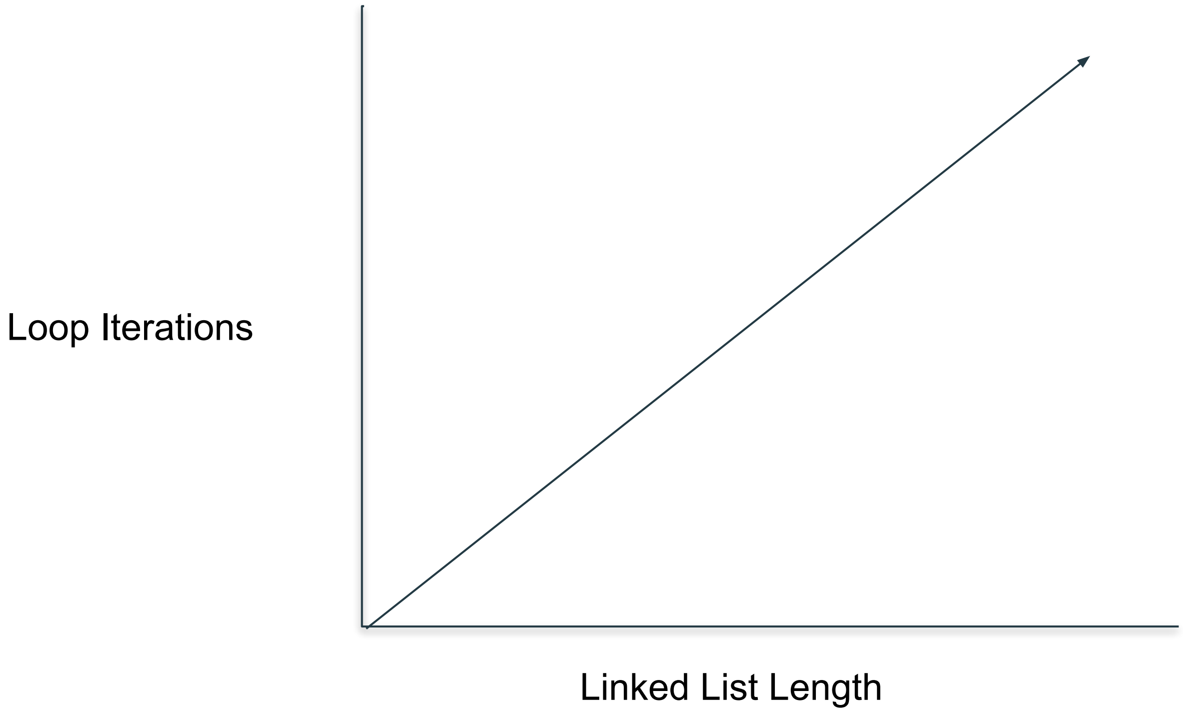
Considering the LinkedList has 10 nodes, we will loop through all Nodes until a match is found.

There will be no match in this case since “Jerry” is not in the LinkedList.

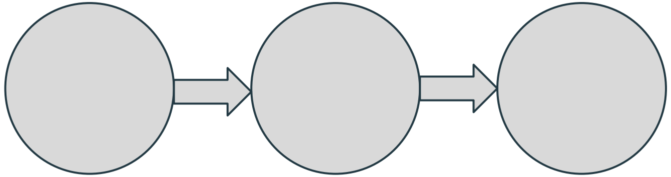
We will therefore go through the loop condition 10 times.

```
//LINKED LIST THAT HOLDS A STRING IN EACH NODE
boolean find(String toFind) {
  Node current = this.front.next;
  while(current != null) {
    if(current.value.equals(toFind)) {return true;}
    current = current.next;
  }
  return false;
}
```

Loop Iterations vs Linked List Length (when String to find is not in list)



Counting Steps...

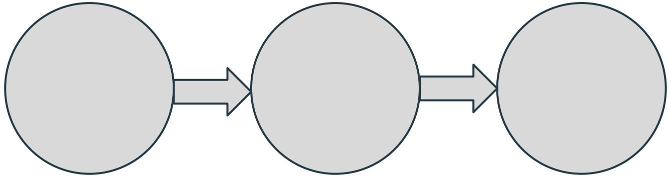


```
//LINKED LIST THAT HOLDS A STRING IN EACH NODE
boolean find(String toFind) {
  Node current = this.front.next;
  while(current != null) {
    if(current.value.equals(toFind)) {return true;}
    current = current.next;
  }
  return false;
}
```

If we are still searching for “Jerry” and it is found in the LinkedList, how many times will the while loop run?

A. 9
B. 1
C. 10
D. Cannot be determined

Answer - D



```
//LINKED LIST THAT HOLDS A STRING IN EACH NODE
boolean find(String toFind) {
  Node current = this.front.next;
  while(current != null) {
    if(current.value.equals(toFind)) {return true;}
    current = current.next;
  }
  return false;
}
```

If “Jerry” is in the LinkedList, we do not know exactly how many Nodes from the start it is located.

It could be located at start of list, end, middle, anywhere...

Counting Steps...

```
//FIND FIRST MATCHING VALUE ACROSS BOTH LISTS

String findMatch(LinkedList a, LinkedList b) {
  Node aNode = a.first.next;
  for (int i = 0; i < a.size; i++) {
    Node bNode = b.first.next;
    for (int j = 0; j < b.size; j++) {
      if (aNode.value.equals(bNode.value) {
        return aNode.value;
      }
      bNode = bNode.next;
    }
    aNode = aNode.next;
  }
  return null;
}
```

Assuming that there are no matches between the two Linked Lists a and b, how many times would the inner if statement be executed?

A. a.size
B. b.size
C. a.size * b.size
D. a.size + b.size
E. None of the above

Answer - C

```
//FIND FIRST MATCHING VALUE ACROSS BOTH LISTS

String findMatch(LinkedList a, LinkedList b) {
  Node aNode = a.first.next;
  for (int i = 0; i < a.size; i++) {
    Node bNode = b.first.next;
    for (int j = 0; j < b.size; j++) {
      if (aNode.value.equals(bNode.value) {
        return aNode.value;
      }
      bNode = bNode.next;
    }
    aNode = aNode.next;
  }
  return null;
}
```

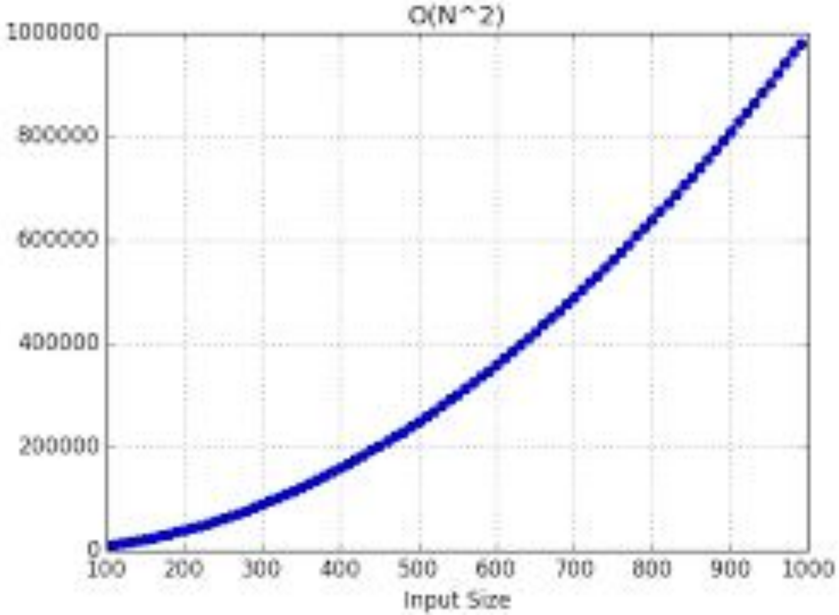
The inner if statement is inside both for loops.

Outer for loop runs a.size times.

Inner for loop runs b.size times.

Anything inside inner for loop is called a.size * b.size times.

If Statement Calls vs List Sizes (when no items match across two lists)

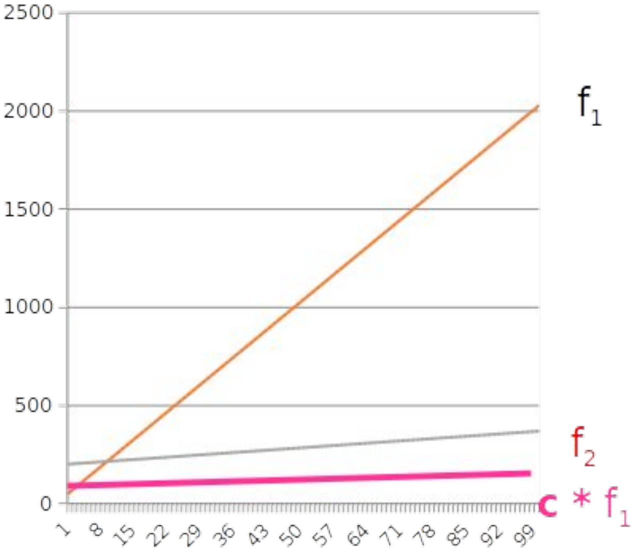


Categorizing Runtimes

f(n) = **O**(g(n)), if there are positive constants c and n_0 such that f(n) ≤ **c** * g(n) for all n ≥ n_0.

f(n) = **Ω**(g(n)), if there are positive constants c and n_0 such that f(n) ≥ **c** * g(n) for all n ≥ n_0.

- f_1 is $\Omega(f_2)$ because $f_1 > f_2$ (after about n=10, so we set n_0 = 10)
  - f_2 is clearly a ***lower bound*** on f_1 and that's what big-Ω is all about
- But f_2 is $\Omega(f_1)$ as well!
  - We just have to use the "**c**" to adjust so f_1 that it moves below f_2



Summary

Big-O

- **Upper bound** on a function
- f(n) = O(g(n)) means that we can expect f(n) will always be **under** the bound g(n)
  - But we don't count n up to some starting point n_0
  - And we can "cheat" a little bit by moving g(n) up by multiplying by some constant c

Big-Ω

- **Lower bound** on a function
- f(n) = Ω(g(n)) means that we can expect f(n) will always be **over** the bound g(n)
  - But we don't count n up to some starting point n_0
  - And we can "cheat" a little bit by moving g(n) down by multiplying by some constant c

Big-θ

- **Tight bound** on a function.
- If f(n) = O(g(n)) *and* f(n) = Ω(g(n)), then f(n) = θ(g(n)).
- Basically it means that f(n) and g(n) are interchangeable
- Examples:
  - 3n+20 = θ(10n+7)
  - $5n^2 + 50n + 3 = \theta(5n^2 + 100)$

Usually in a coding interview, when the interviewer asks you “the big-O bound” of an algorithm, they are referring to the tight bound

Let $f(n) = 100$

Which of the following is true?

A. $f(n)$ is $O(2^n)$
B. $f(n)$ is $O(n^2)$
C. $f(n)$ is $O(n)$
D. All of these
E. None of these

Let f(n) = 100

Which of the following is true?

A. f(n) is $O(2^n)$

B. f(n) is $O(n^2)$

C. f(n) is $O(n)$

[✓ D. All of these]

E. None of these

What about the **tight bound**?

```java
public static void printSomething(int[] items) {
    System.out.println(items[0]);
    int middleIndex = items.length / 2;
    int index = 0;
    while (index < middleIndex) {
        System.out.println(items[index]);
        index++;
    }
    for (int i = 0; i < 100; i++) {
        System.out.println("hi");
    }
  }
```

What is the tight bound of printSomething(), if the length of **`items`** is n?

A. Θ (logn)
B. Θ (n)
C. Θ (1)
D. None of the above

```java
public static void printSomething(int[] items) {

    System.out.println(items[0]);

    int middleIndex = items.length / 2;

    int index = 0;

    while (index < middleIndex) {

        System.out.println(items[index]);

        index++;

    }
```

Θ(n)

```java
    for (int i = 0; i < 100; i++) {

        System.out.println("hi");

    }
```

Θ (1)

```java
}
```

What is the tight bound of printSomething(), if the length of **`items`** is n?

A. Θ (logn)

[✓ B. Θ (n)]

C. Θ (1)

D. None of the above

```
public int binarySearchItem(int[] sortedArray, int value) {
        int index = -1;
        int low = 0;
        int high = sortedArray.length;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (sortedArray[mid] < value) {
                low = mid + 1;
            } else if (sortedArray[mid] > value) {
                high = mid - 1;
            } else if (sortedArray[mid] == value) {
                index = mid;
                break;
            }
        }
        return index;
    }
```

We are using this algorithm to search in a **sorted array** of integers for the index of an integer value. What is the tight bound for this algorithm?

A. Θ (logn)
B. Θ (n)
C. Θ (1)
D. None of the above

```java
public int binarySearchItem(int[] sortedArray, int value) {
        int index = -1;
        int low = 0;
        int high = sortedArray.length;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (sortedArray[mid] < value) {
                low = mid + 1;
            } else if (sortedArray[mid] > value) {
                high = mid - 1;
            } else if (sortedArray[mid] == value) {
                index = mid;
                break;
            }
        }
        return index;
}
```

We are using this algorithm to search in a **sorted array** of integers for the index of an integer value. What is the tight bound for this algorithm?

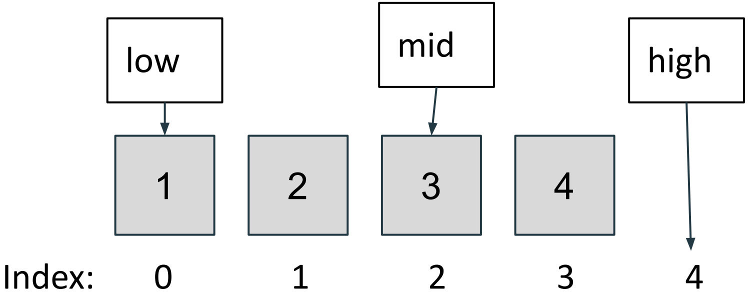
[✓ A. Θ (logn)]

B. Θ (n)

C. Θ (1)

D. None of the above

Assume that we want to look for the integer 2 in the following array:

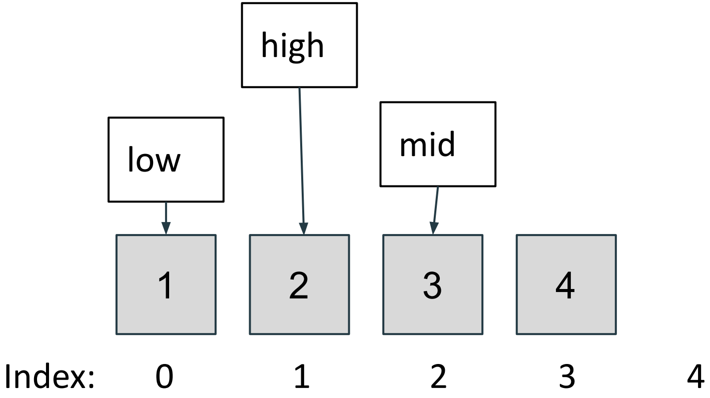


	1	2	3	4	
Index:	0	1	2	3	4

```
public int binarySearchItem(int[] sortedArray,
                                   int value) {
        int index = -1;
        int low = 0;
        int high = sortedArray.length;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (sortedArray[mid] < value) {
                low = mid + 1;
            } else if (sortedArray[mid] > value) {
                high = mid - 1;
            } else if (sortedArray[mid] == value) {
                index = mid;
                break;
            }
        }
        return index;
}
```

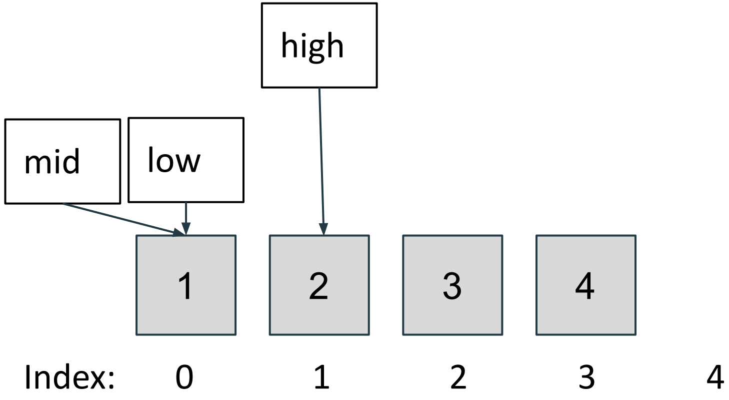
Assume that we want to look for the integer 2 in the following array:



```
public int binarySearchItem(int[] sortedArray,
                                   int value) {
        int index = -1;
        int low = 0;
        int high = sortedArray.length;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (sortedArray[mid] < value) {
                low = mid + 1;
            } else if (sortedArray[mid] > value) {
                high = mid - 1;
            } else if (sortedArray[mid] == value) {
                index = mid;
                break;
            }
        }
        return index;
}
```

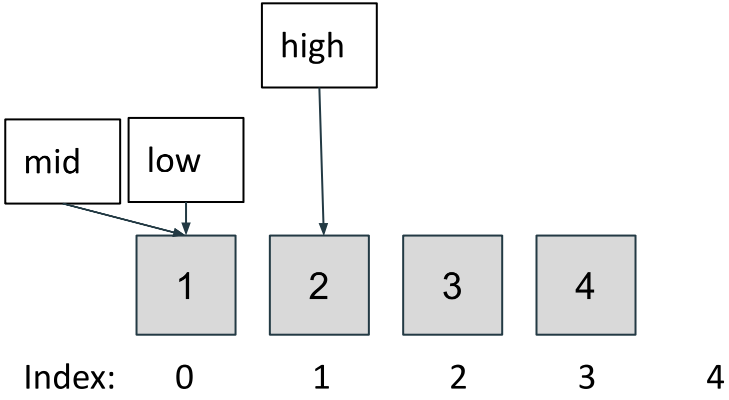
Assume that we want to look for the integer 2 in the following array:



```
public int binarySearchItem(int[] sortedArray,
                                  int value) {
        int index = -1;
        int low = 0;
        int high = sortedArray.length;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (sortedArray[mid] < value) {
                low = mid + 1;
            } else if (sortedArray[mid] > value) {
                high = mid - 1;
            } else if (sortedArray[mid] == value) {
                index = mid;
                break;
            }
        }
        return index;
}
```

Assume that we want to look for the integer 2 in the following array:



high

mid

low

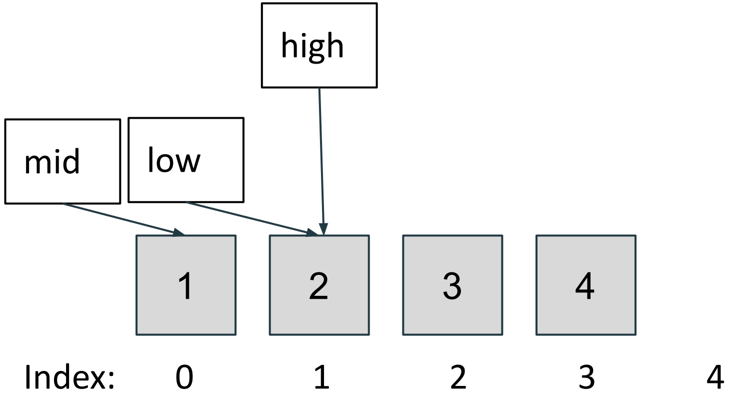
1 2 3 4

Index: 0 1 2 3 4

```
public int binarySearchItem(int[] sortedArray,
                                  int value) {
        int index = -1;
        int low = 0;
        int high = sortedArray.length;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (sortedArray[mid] < value) {
                low = mid + 1;
            } else if (sortedArray[mid] > value) {
                high = mid - 1;
            } else if (sortedArray[mid] == value) {
                index = mid;
                break;
            }
        }
        return index;
}
```

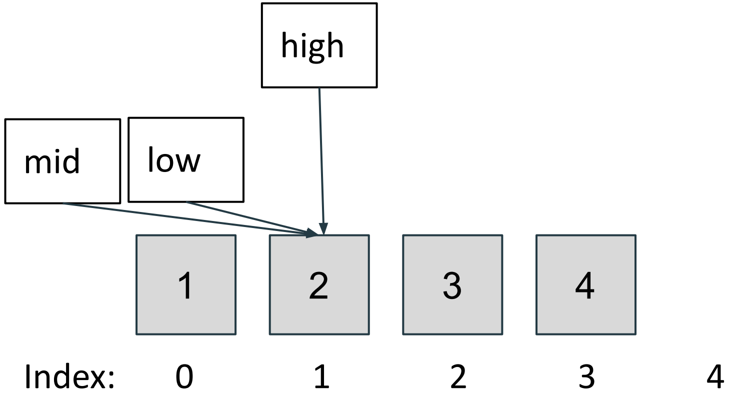
Assume that we want to look for the integer 2 in the following array:



```
public int binarySearchItem(int[] sortedArray,
                                  int value) {
        int index = -1;
        int low = 0;
        int high = sortedArray.length;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (sortedArray[mid] < value) {
                low = mid + 1;
            } else if (sortedArray[mid] > value) {
                high = mid - 1;
            } else if (sortedArray[mid] == value) {
                index = mid;
                break;
            }
        }
        return index;
}
```

Assume that we want to look for the integer 2 in the following array:



high

mid

low

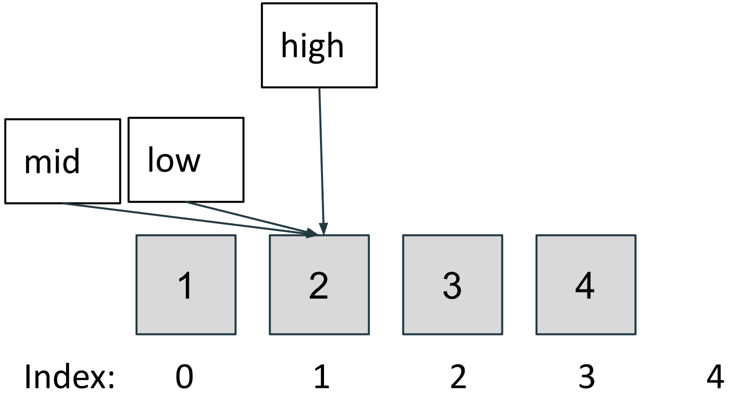
1 2 3 4

Index: 0 1 2 3 4

```
public int binarySearchItem(int[] sortedArray,
                                   int value) {
        int index = -1;
        int low = 0;
        int high = sortedArray.length;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (sortedArray[mid] < value) {
                low = mid + 1;
            } else if (sortedArray[mid] > value) {
                high = mid - 1;
            } else if (sortedArray[mid] == value) {
                index = mid;
                break;
            }
        }
        return index;
}
```

Assume that we want to look for the integer 2 in the following array:



Integer found!

```
public int binarySearchItem(int[] sortedArray,
                                   int value) {
    int index = -1;
    int low = 0;
    int high = sortedArray.length;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (sortedArray[mid] < value) {
            low = mid + 1;
        } else if (sortedArray[mid] > value) {
            high = mid - 1;
        } else if (sortedArray[mid] == value) {
            index = mid;
            break;
        }
    }
    return index;
}
```

Questions?

Feel free to bring up other runtime problems you are confused about!

Poll: Next Topic

A. Midterm 1 Review Q & A

B. Past Quizzes Q & A