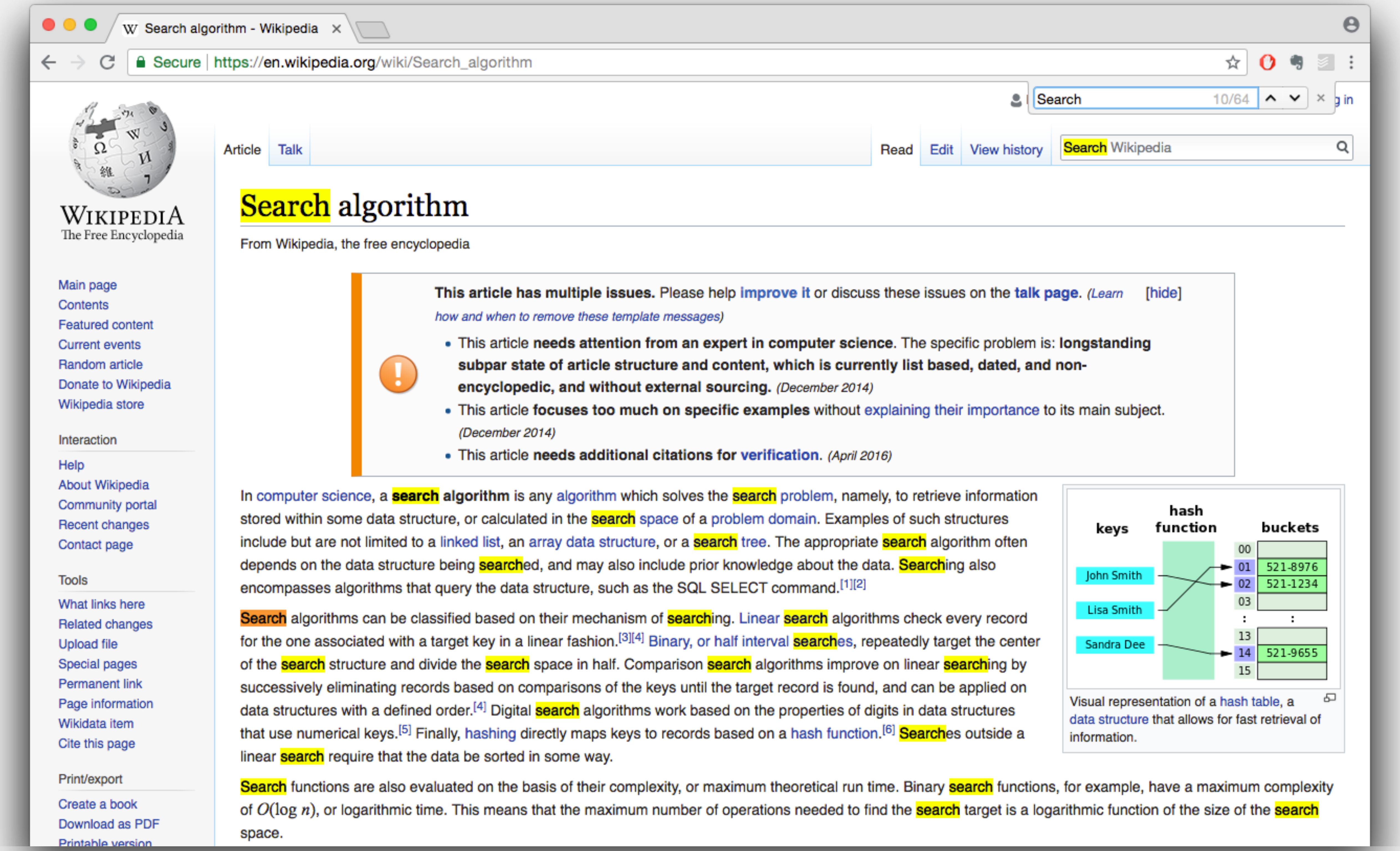
Week 11: Search Algorithms

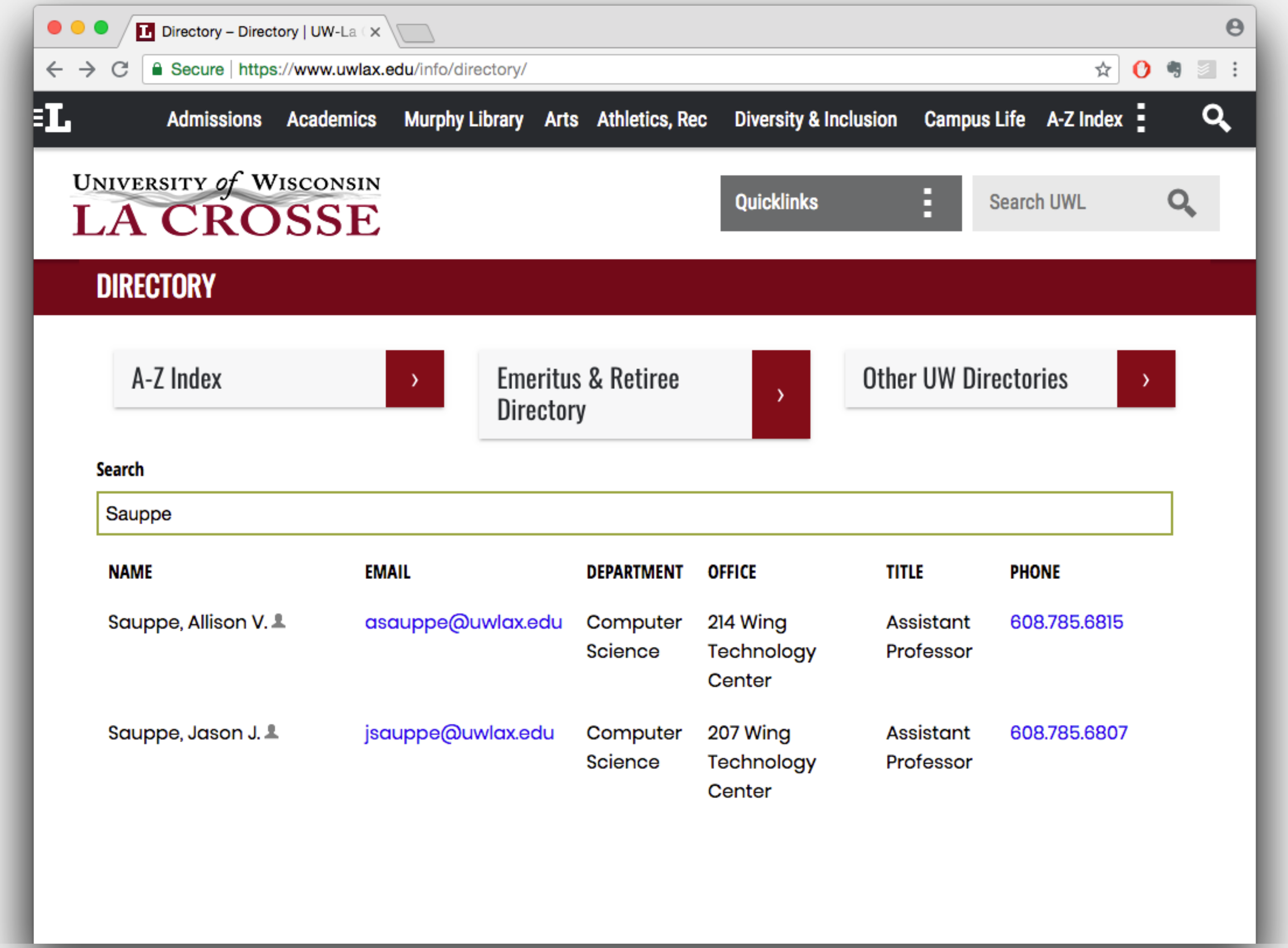
UNIVERSITY of WISCONSIN LA CROSSE COMPUTER SCIENCE

CS 220: Software Design II — D. Mathias

Searching

Common activity when working with data





Linear Search

```java
int toFind = 86;
for (int i = 0; i < grades.length; i++) {
    if (grades[i] == toFind) {
        System.out.print(i + ", ");
    }
}
```

A *linear search* looks at each element in a data structure, in order, until encountering the desired element(s)

grades (int[])

100	68	49	77	95	82	99	83	86	87	70	86	64	40	86
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Linear Search

```java
int toFind = 86;
for (int i = 0; i < grades.length; i++) {
    if (grades[i] == toFind) {
        System.out.print(i);
        break;
    }
}
```

What if we can guarantee that there is at most one element that will match?

- i.e., the element will appear either 0 or 1 times

grades (int[])

100	68	49	77	95	82	99	83	65	87	70	86	64	40	88
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Linear Search

```
int toFind = 86;
for (int i = 0; i < grades.length; i++) {


    // to write


}
```

What if there can be multiple instances of the number, and the list is sorted?

What should the loop code be to terminate as quickly as possible?

grades (int[])

40	49	64	65	68	70	77	82	86	86	86	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Linear Search

```java
int toFind = 86;
for (int i = 0; i < grades.length; i++) {
    if (grades[i] == toFind) {
        System.out.print(i);
    } else if (grades[i] > toFind) {
        break;
    }
}
```

What if there can be multiple instances of the number, and the list is sorted?

What should the loop code be to terminate as quickly as possible?

grades (int[])

40	49	64	65	68	70	77	82	86	86	86	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Exercise: Runtime Analysis

Fill in the following chart with runtimes using linear search

	unsorted list	sorted list (no repeats)
search for the smallest value		
search for the largest value		
search for the median value		
search for a value that doesn't exist		
search for some random value that does exist (worst case)		

e.g., we're looking for the number 100, but don't realize it's the largest number in the array

`grades (int[])`

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Exercise: Runtime Analysis

Fill in the following chart with runtimes using linear search

	unsorted list	sorted list (no repeats)
search for the smallest value	O(n)	O(1)
search for the largest value	O(n)	O(1)
search for the median value	O(?)	O(1)
search for a value that doesn't exist	O(n)	O(n)
search for some random value that does exist (worst case)	O(n)	O(n)

`grades (int[])`

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Exercise: Runtime Analysis

Fill in the following chart with runtimes using linear search

	unsorted list	**sorted list (no repeats)**
search for the smallest value	O(n)	O(1)
search for the largest value	O(n)	O(1)
search for the median value	O(?)	O(1)
search for a value that doesn't exist	O(n)	O(n)
search for some random value that does exist (worst case)	O(n)	O(n)

relies on absolute positions in the array

relies on relative positions in the array

grades (int[])

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
40	49	64	65	68	70	77	82	83	86	87	88	95	99	100

Search Strategies

- Linear Search is not the only way to find a value in an array

- Strategies:
  - Linear search
  - Random guessing
  - Any others?

Let’s play a game

- I’m going to think of a number in [1. 1000].
- You have two goals:
  - Determine the number
  - Use the smallest possible number of guesses

Binary Search

A *binary search* uses a *divide and conquer* approach to subdivide a sorted list to find a number

- *divide and conquer* approaches take a problem and break it down into smaller problems

Basic premise:

- array to search in is the whole array; start at midpoint
- is the number to find higher than the midpoint? search above; lower? search below
- redefine the array to search in as either the above or below half; repeat

Binary Search

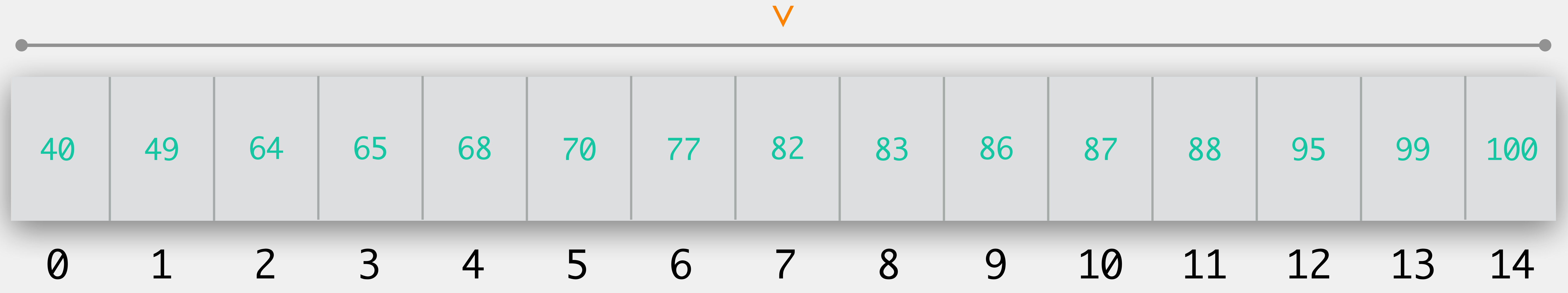
Start at midpoint in range to search in

Decide if number to find is higher or lower than midpoint

Redefine the range to either upper or lower half of array

Repeat

`toFind = 86`



Binary Search

Start at midpoint in range to search in

Decide if number to find is higher or lower than midpoint

Redefine the range to either upper or lower half of array

Repeat

`toFind = 86`

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary Search

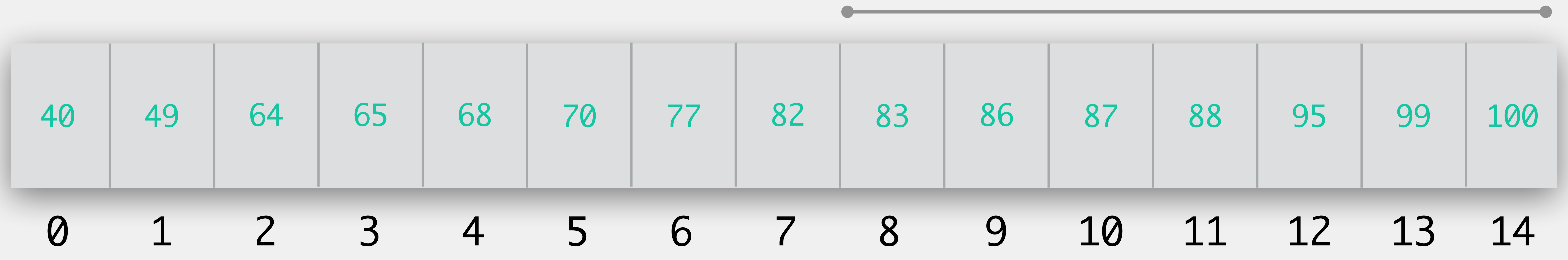
Start at midpoint in range to search in

Decide if number to find is higher or lower than midpoint

Redefine the range to either upper or lower half of array

Repeat

`toFind = 86`



Binary Search

Start at midpoint in range to search in

Decide if number to find is higher or lower than midpoint

Redefine the range to either upper or lower half of array

Repeat

`toFind = 86`

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary Search

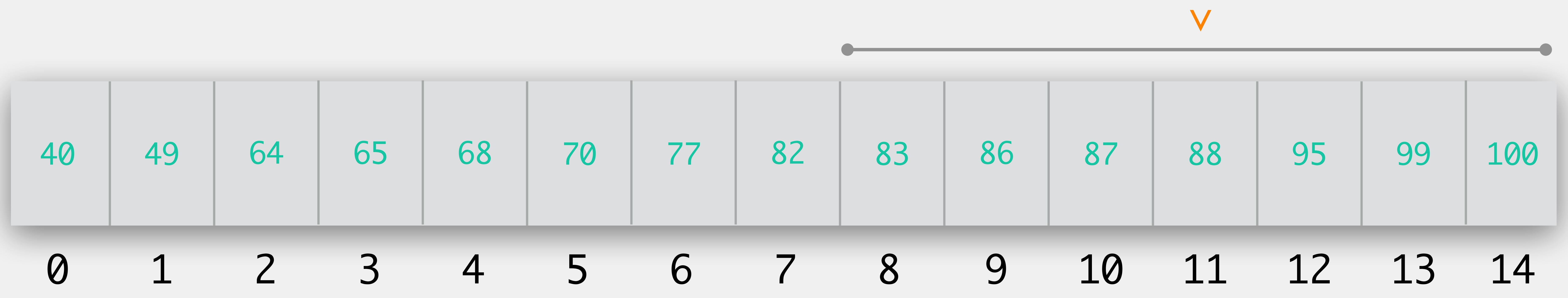
Start at midpoint in range to search in

Decide if number to find is higher or lower than midpoint

Redefine the range to either upper or lower half of array

Repeat

`toFind = 86`



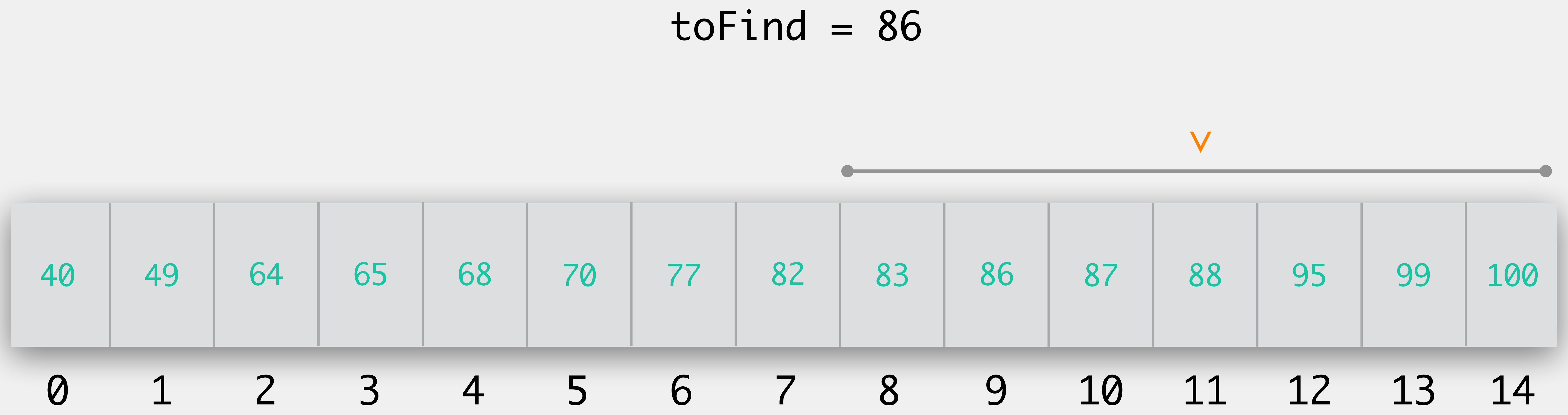
Binary Search

Start at midpoint in range to search in

Decide if number to find is higher or lower than midpoint

Redefine the range to either upper or lower half of array

Repeat



Binary Search

Start at midpoint in range to search in

Decide if number to find is higher or lower than midpoint

Redefine the range to either upper or lower half of array

Repeat

`toFind = 86`

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

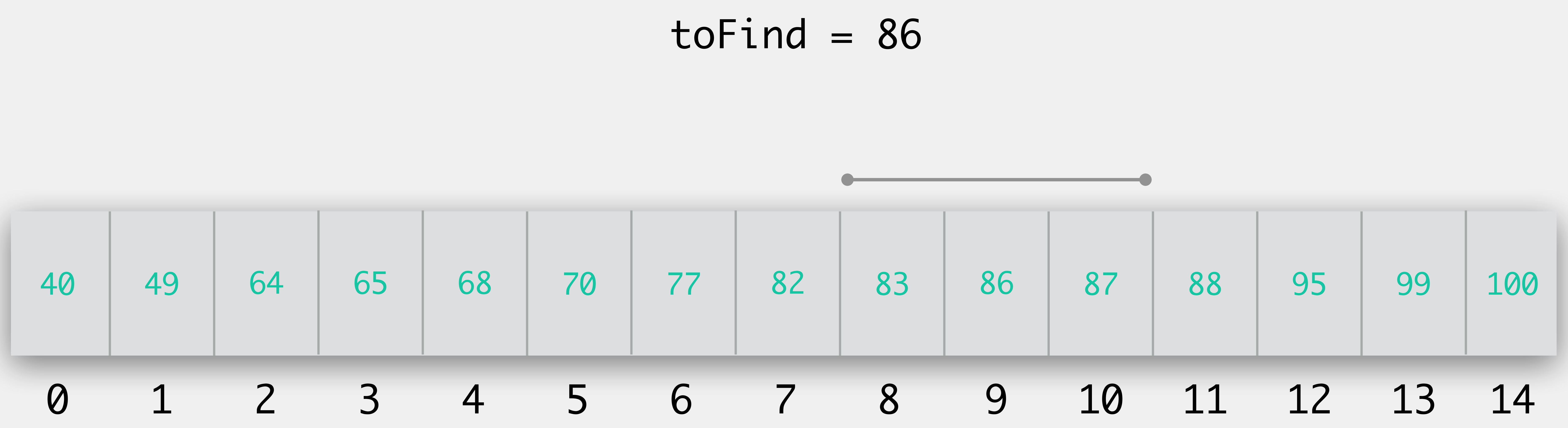
Binary Search

Start at midpoint in range to search in

Decide if number to find is higher or lower than midpoint

Redefine the range to either upper or lower half of array

Repeat



Binary Search

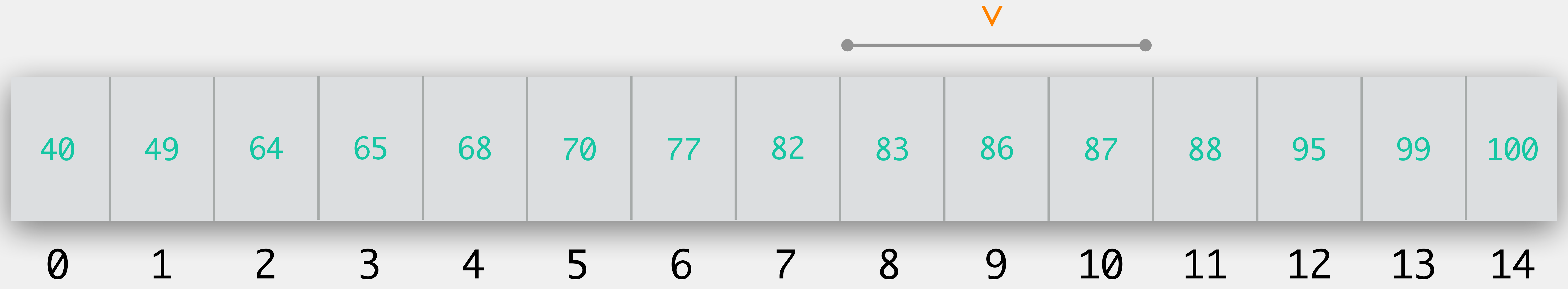
Start at midpoint in range to search in

Decide if number to find is higher or lower than midpoint

Redefine the range to either upper or lower half of array

Repeat

`toFind = 86`



Binary Search

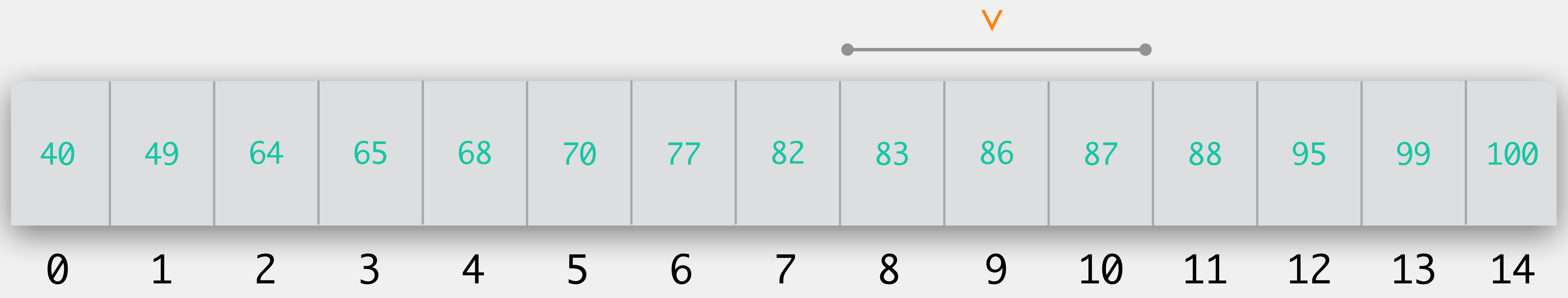
Start at midpoint in range to search in

Decide if number to find is higher or lower than midpoint

Redefine the range to either upper or lower half of array

Repeat

`toFind = 86`



Exercise: Runtime Analysis

Considering binary search…

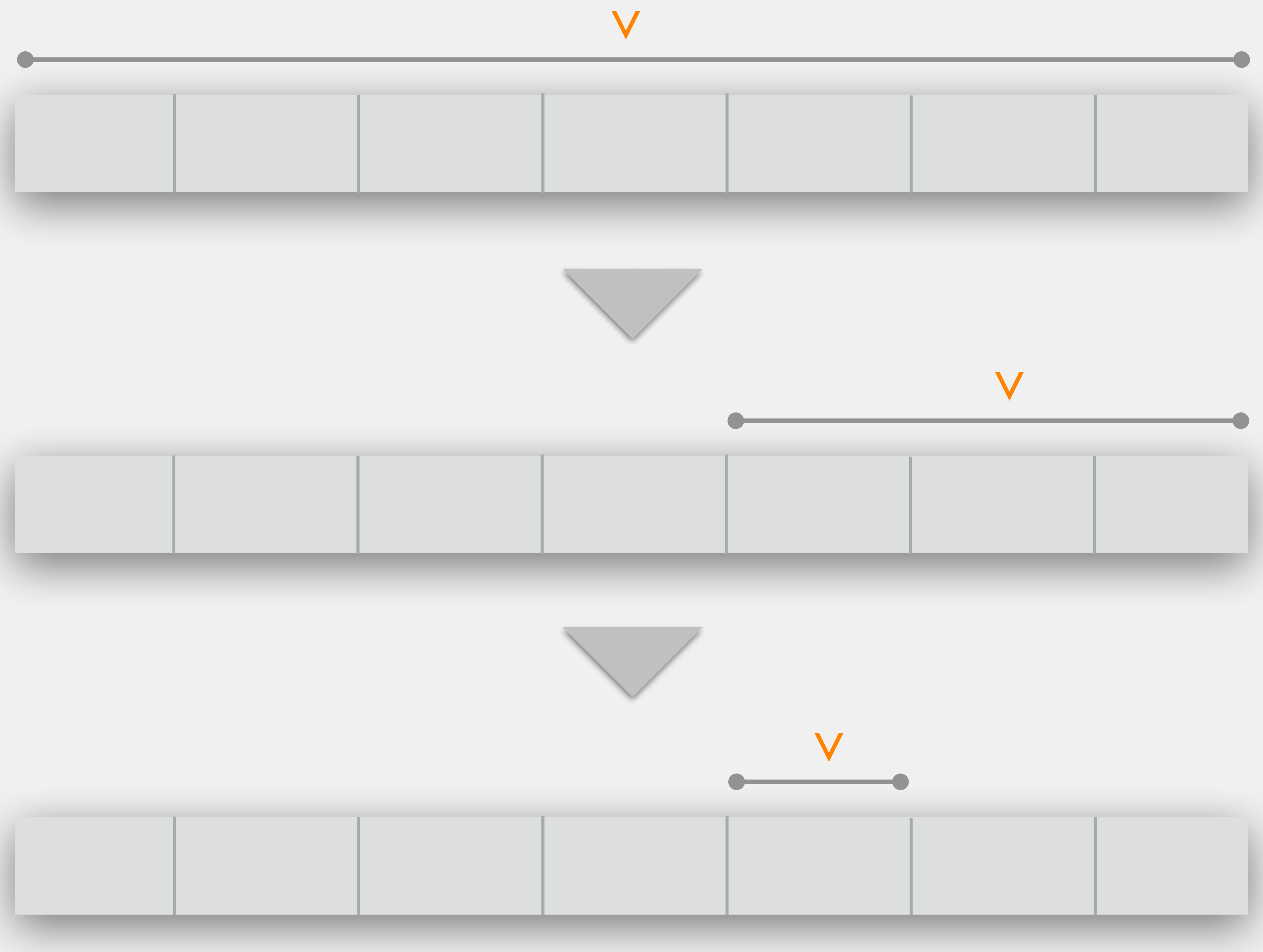
- what is the best case scenario?
- what is the worst case scenario?

Use the array below as an example if helpful

`grades (int[])`

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

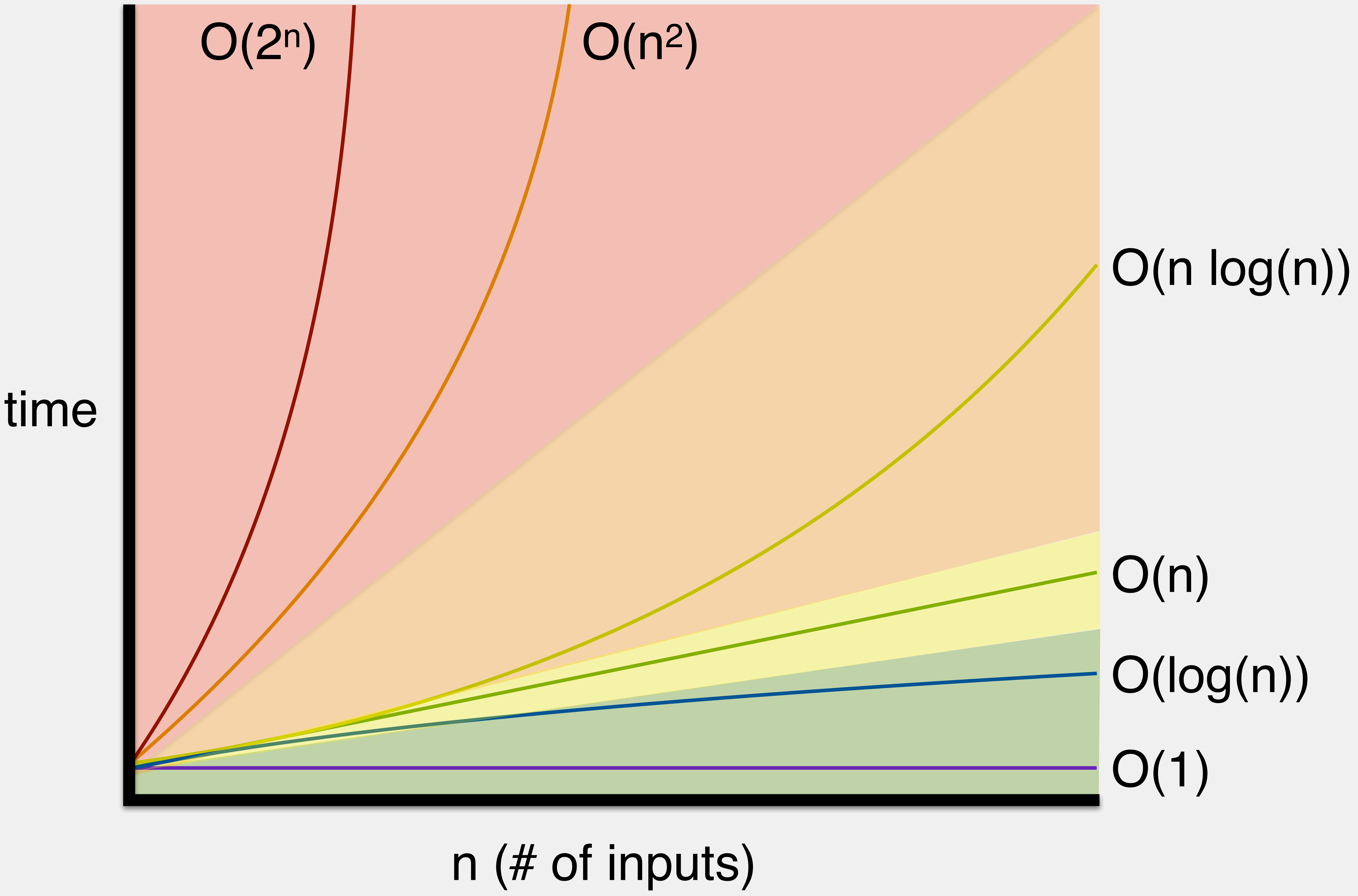
O(log(n)) Algorithm



Most divide and conquer algorithms involve a worst-case runtime with a log(n) term

For Binary Search, log(n) is an approximation of the number of divisions necessary to arrive at an answer in the worst-case scenario

Big O Notation



O(log(n)) is actually pretty good!

Aside: logarithms

What is $\log_b n$?

A way to think about logs:

To what power must I raise b to get n?

What is $\log_{10} 1000$?

What is $\log_5 625$?

What is $\log_2 32768$?

Exercise: Runtime Analysis

Fill in the following chart with runtimes using binary search

	sorted list (no repeats)
search for the smallest value	
search for the largest value	
search for the median value	
search for a value that doesn't exist	
search for some random value that does exist (worst case)	

`grades (int[])`

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Exercise: Runtime Analysis

Fill in the following chart with runtimes using binary search

	sorted list (no repeats)
search for the smallest value	O(log(n))
search for the largest value	O(log(n))
search for the median value	O(1)
search for a value that doesn't exist	O(log(n))
search for some random value that does exist (worst case)	O(log(n))

`grades (int[])`

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Exercise: Runtime Analysis

Fill in the following chart with runtimes using binary search

	sorted list (no repeats)
search for the smallest value	~~O(log(n))~~ O(1)
search for the largest value	~~O(log(n))~~ O(1)
search for the median value	O(1)
search for a value that doesn't exist	O(log(n))
search for some random value that does exist (worst case)	O(log(n))

`grades (int[])`

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Runtime Analysis

Compare and contrast - where are we doing worse vs better? Is the tradeoff worth it?

	sorted list (linear search)	**sorted list (binary search)**
search for the smallest value	O(1)	~~O(log(n))~~ O(1)
search for the largest value	O(1)	~~O(log(n))~~ O(1)
search for the median value	O(1)	O(1)
search for a value that doesn't exist	O(n)	O(log(n))
search for some random value that does exist (worst case)	O(n)	O(log(n))

`grades (int[])`

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Runtime Analysis

What value(s) are you looking for…

	sorted list (linear search)	**sorted list (binary search)**
best case		
worst case		

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Runtime Analysis

What value(s) are you looking for…

	sorted list (linear search)	sorted list (binary search)
best case	40	
worst case		

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Runtime Analysis

What value(s) are you looking for…

	sorted list (linear search)	sorted list (binary search)
best case	40	
worst case	100	

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Runtime Analysis

What value(s) are you looking for…

	sorted list (linear search)	sorted list (binary search)
best case	40	82
worst case	100	

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Runtime Analysis

What value(s) are you looking for…

	sorted list (linear search)	sorted list (binary search)
best case	40	82
worst case	100	40, 64, 68, 77, 83, 87, 95, 100

40	49	64	65	68	70	77	82	83	86	87	88	95	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Code: Binary Search (iterative)

```
private static int binarySearch(int arr[], int toFind) {
    int begin = 0;
    int end = arr.length - 1;

    while (begin <= end) {
      int mid = (begin + end) / 2;   // Find the midpoint

      if (arr[mid] == toFind) {  // Found it!
        return mid;
      } else if (arr[mid] < toFind) {  // mid value too small
        begin = mid + 1;
      } else { /* arr[mid] > toFind */  // mid value too large
        end = mid - 1;
      }
    }

    return -1;  // Failed search
}
```

Code: Binary Search (recursive)

```java
private static int binarySearch(int arr[], int toFind) {
    return binSearchHelper(arr, toFind, 0, arr.length - 1);
}

private static int binSearchHelper(int arr[], int toFind, int begin, int end) {
    if (begin > end) {
        return -1;  // Failed search
    }
    int mid = (begin + end) / 2;  // Find the midpoint
    if (arr[mid] == toFind) {  // Found it!
        return mid;
    } else if (arr[mid] < toFind) {  // mid value too small
        return binSearchHelper(arr, toFind, mid + 1, end);
    } else { /* arr[mid] > toFind */ // mid value too large
        return binSearchHelper(arr, toFind, begin, mid - 1);
    }
}
```

Searching on Data Structures

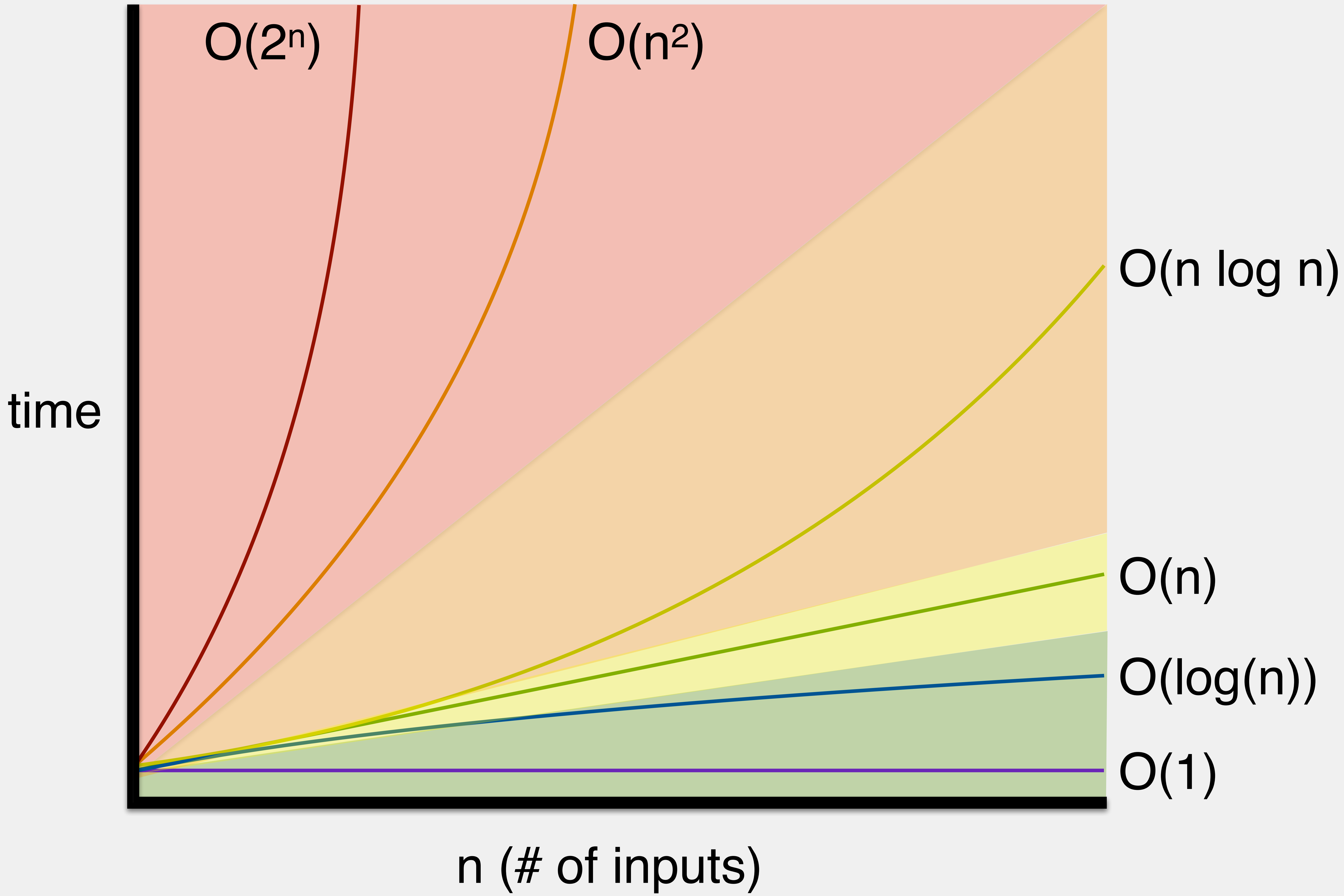
- Discussed linear and binary search on arrays
  - runtimes would be comparable for an array list
- What about a singly linked list?
  - can linear search be performed on one?
  - what about binary search on a singly linked list?
  - what would the runtimes be like?

Linked Lists & Search

Consider the following chart for a singly linked list

	unsorted linked list (linear search)	**sorted linked list (linear search)**	***sorted array (linear search)***	**sorted linked list (binary search)**	***sorted array (binary search)***
search for the smallest value	O(n)	O(1)	*O(1)*	O(1)	*O(log(n))*
search for the largest value	O(n)	**O(n)**	*O(1)*	**O(n)**	*O(log(n))*
search for the median value	O(n)	**O(n)**	*O(1)*	**O(n)**	*O(1)*
search for a value that doesn't exist	O(n)	O(n)	*O(n)*	**O(n log(n))**	*O(log(n))*
search for some random value that does exist (worst case)	O(n)	O(n)	*O(n)*	**O(n log(n))**	*O(log(n))*

Big O Notation



O(n log n) is significantly larger than O(n) and O(log n) as n gets large…

Linked Lists & Search

	unsorted linked list (linear search)	**sorted linked list (linear search)**	***sorted array (linear search)***	**sorted linked list (binary search)**	***sorted array (binary search)***
search for the median value	O(n)	**O(n)**	*O(1)*	**O(n)**	*O(1)*