

Dr. Gina Bai

Spring 2023

Logistics

- ZY-7B and ZY-8Aon zyBook > Assignments
 - Due: Wednesday, April 5, at 11:59pm
- PA10 A, B on zyBook > Chap 11
 - Due: **Thursday, April 6**, at 11:59pm

Start Early!!!

- Midterm Exam 2 Regrade Requests
 - Due: Tuesday, April 11

Sorting

- Sorting is the process of arranging a list of items into either ascending (default in most cases) or descending order
 - Numerical order, alphabetical order

Sorting Algorithms

- Selection sort (in CS1101)
- Insertion sort (in CS1101)
- Bubble sort
- Merge sort
- Quick sort
- Heap sort

•

Selection Sort

zyBook Chap 7.11

Selection Sort

- Orders a list of values by repeatedly
 - selecting the smallest or largest value from the unsorted subarray, and
 - attaching it to the end of the sorted subarray

[0] [1] [2] [3] [4] [5] 21 18 -4 27 7 2

Initial Array

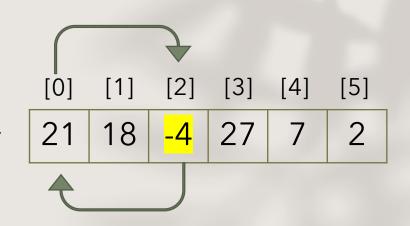
• The algorithm:

[0] [1] [2] [3] [4] [5] 21 18 -4 27 7 2

- The algorithm:
 - Traverse the array to find the smallest value

[0] [1] [2] [3] [4] [5] 21 18 <mark>-4</mark> 27 7 2

- The algorithm:
 - Traverse the array to find the smallest value



- The algorithm:
 - Traverse the array to find the smallest value
 - Swap it with the element at index 0

[0] [1] [2] [3] **Initial Array** 21 18 -4 27

- The algorithm:
 - Traverse the array to find the smallest value
 - Swap it with the element at index 0





[5]

[4]

[0] [1] [2] [3] [4] [5] 21 18 <mark>-4</mark> 27 7 2

Initial Array

• The algorithm:

- Traverse the array to find the smallest value
- Swap it with the element at index 0
- Traverse the array to find the 2ndsmallest value

After 1st iteration

-4 18 21 27 7 2

OR, generally, we can take it as:

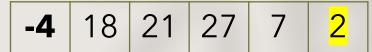
Traverse the unsorted part of the array to find the smallest value among the remaining elements

[0] [1] [2] [3] [4] [5] 21 18 <mark>-4</mark> 27 7 2

Initial Array

- The algorithm:
 - Traverse the array to find the smallest value
 - Swap it with the element at index 0
 - Traverse the array to find the 2ndsmallest value

After 1st iteration



OR, generally, we can take it as:

Traverse the unsorted part of the array to find the smallest value among the remaining elements

[0] [1] [2] [3] [4] [5] 21 18 <mark>-4</mark> 27 7 2

Initial Array

• The algorithm:

- Traverse the array to find the smallest value
- Swap it with the element at index 0
- Traverse the array to find the 2ndsmallest value
- Swap it with the element at index 1

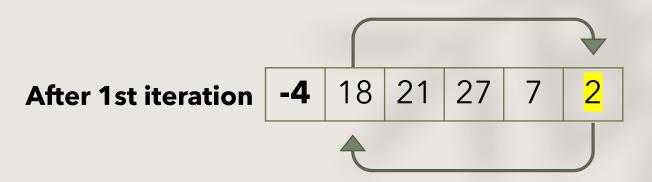
After 1st iteration

-4 18 21 27 7	2
----------------------	---

[0] [1] [2] [3] [4] [5]

Initial Array 21 18 -4 27 7 2

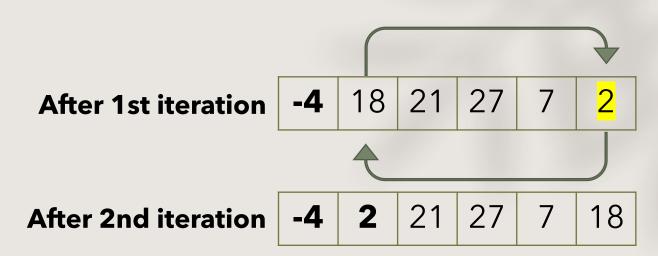
- The algorithm:
 - Traverse the array to find the smallest value
 - Swap it with the element at index 0
 - Traverse the array to find the 2ndsmallest value
 - Swap it with the element at index 1



[0] [1] [2] [3] [4] [5]

Initial Array 21 18 -4 27 7 2

- The algorithm:
 - Traverse the array to find the smallest value
 - Swap it with the element at index 0
 - Traverse the array to find the 2ndsmallest value
 - Swap it with the element at index 1



[0] [1] [2] [3] [4] [5] 21 18 -4 27 7 2

• The algorithm:

- Traverse the array to find the smallest value
- Swap it with the element at index 0
- Traverse the array to find the 2ndsmallest value
- Swap it with the element at index 1
- ...
- Repeat until all values are in their proper places

After 1st iteration

Initial Array



After 2nd iteration



[0] [1] [2] [3] [4] [5] 21 18 <mark>-4</mark> 27 7 2

Initial Array

• The algorithm:

- Traverse the array to find the smallest value
- Swap it with the element at index 0
- Traverse the array to find the 2ndsmallest value
- Swap it with the element at index 1
- ...
- Repeat until all values are in their proper places

After 1st iteration



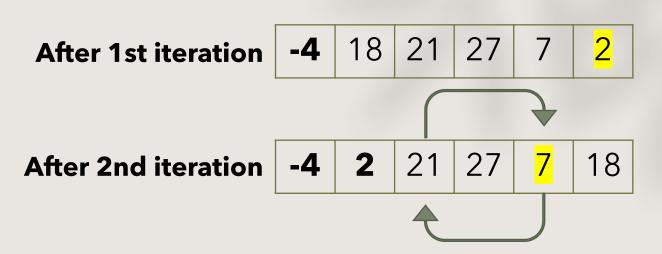
After 2nd iteration



[0] [1] [2] [3] [4] [5] 21 18 <mark>-4</mark> 27 7 2

• The algorithm:

- Traverse the array to find the smallest value
- Swap it with the element at index 0
- Traverse the array to find the 2ndsmallest value
- Swap it with the element at index 1
- ...
- Repeat until all values are in their proper places

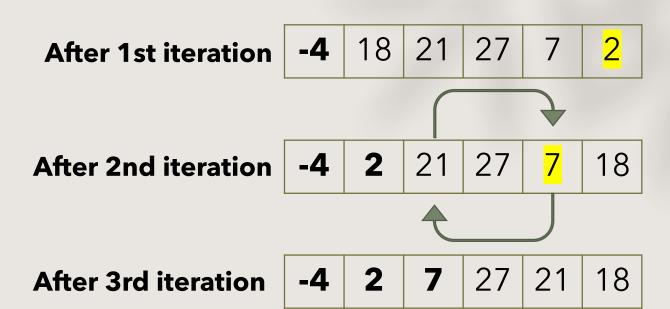


[0] [1] [2] [3] [4] [5] 21 18 -4 27 7 2

Initial Array

The algorithm:

- Traverse the array to find the smallest value
- Swap it with the element at index 0
- Traverse the array to find the 2ndsmallest value
- Swap it with the element at index 1
- ...
- Repeat until all values are in their proper places



[2] [5] [0] [1] [3] [4] 18

Initial Array

• The algorithm:

- Traverse the array to find the smallest value
- Swap it with the element at index 0
- Traverse the array to find the 2ndsmallest value
- Swap it with the element at index 1
- Repeat until all values are in their proper places

After 1st iteration

18 21

After 2nd iteration

2 21

After 3rd iteration

[0] [1] [2] [3] [4] [5] 21 18 <mark>-4</mark> 27 7 2

- The algorithm:
 - Traverse the array to find the smallest value
 - Swap it with the element at index 0
 - Traverse the array to find the 2ndsmallest value
 - Swap it with the element at index 1
 - ...
 - Repeat until all values are in their proper places







[5] [0] [1] [2] [3] [4]

Initial Array

18

- The algorithm:
 - Traverse the array to find the smallest value
 - Swap it with the element at index 0
 - Traverse the array to find the 2ndsmallest value
 - Swap it with the element at index 1

 - Repeat until all values are in their proper places











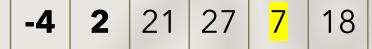
[0] [1] [2] [3] [4] [5] 21 18 <mark>-4</mark> 27 7 2

- The algorithm:
 - Traverse the array to find the smallest value
 - Swap it with the element at index 0
 - Traverse the array to find the 2ndsmallest value
 - Swap it with the element at index 1
 - ...
 - Repeat until all values are in their proper places

After 1st iteration



After 2nd iteration



After 3rd iteration



After 4th iteration



[0] [1] [2] [3] [4] [5] 21 18 <mark>-4</mark> 27 7 2

- The algorithm:
 - Traverse the array to find the smallest value
 - Swap it with the element at index 0
 - Traverse the array to find the 2ndsmallest value
 - Swap it with the element at index 1
 - ...
 - Repeat until all values are in their proper places

** Single element array is naturally sorted

After 1st iteration



After 2nd iteration



After 3rd iteration



After 4th iteration



After 5th iteration

```
import java.util.Arrays;
public class SelectionSort {
    public static void main(String[] args) {
                                                                              Selection Sort
        int[] arr = {21, 18, -4, 27, 7, 2};
        System.out.println("Initial Array: " + Arrays.toString(arr));
        selectionSort(arr):
                                                                              Implementation
    public static void selectionSort(int[] arr) {
        // Notice the range is arr.length - 1, as size 1 array is naturally sorted
        for (int i = 0; i < arr.length - 1; ++i) {</pre>
            // Let the front most element be the current smallest value
            int smallest = i:
            // Traverse the remaining part of the array (j = i + 1), and...
            for (int j = i + 1; j < arr.length; ++j) {</pre>
                // Compare each element with the current smallest value
                if (arr[i] < arr[smallest]) {</pre>
                    smallest = j; // Update the index the smallest value
            swap(arr, i, smallest); // Swap smallest to front
            System.out.println("After iteration #" + i + ": " + Arrays.toString(arr));
                                                                              $ javac SelectionSort.java
                                                                              $ java SelectionSort
    public static void swap(int[] arr, int i, int j) {
                                                                              Initial Array: [21, 18, -4, 27, 7, 2]
        int temp = arr[i];
                                                                              After iteration #0: [-4, 18, 21, 27, 7, 2]
        arr[i] = arr[j];
                                                                              After iteration #1: [-4, 2, 21, 27, 7, 18]
                                                                              After iteration #2: [-4, 2, 7, 27, 21, 18]
        arr[j] = temp;
                                                                              After iteration #3: [-4, 2, 7, 18, 21, 27]
                                                                              After iteration #4: [-4, 2, 7, 18, 21, 27]
```

Insertion Sort

zyBook Chap 7.10

Insertion Sort

- Orders a list of values by repeatedly
 - picking the first value from the unsorted subarray, and
 - inserting it into its proper place in a sorted subarray

[0]	[1]	[2]	[3]	[4]	[5]
21	18	-4	27	7	2

• The algorithm:

[0]	[1]	[2]	[3]	[4]	[5]
21	18	-4	27	7	2

- The algorithm:
 - Pick the first value from the unsorted portion of the array

** The first element is considered sorted

[0]	[1]	[2]	[3]	[4]	[5]
21	<mark>18</mark>	-4	27	7	2

- The algorithm:
 - Pick the first value from the unsorted portion of the array

** The first element is considered sorted

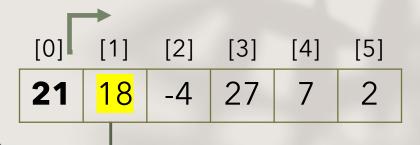
[0]	[1]	[2]	[3]	[4]	[5]
21	<mark>18</mark>	-4	27	7	2

- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value

[0] [1] [2] [3] [4] [5] **21** | 18 | -4 | 27 | 7 | 2

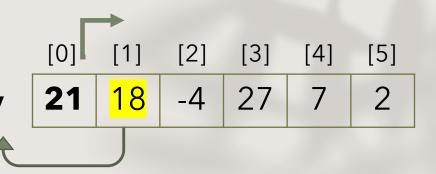
- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value

Insertion Sort (Ascending Order)



- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition

- ** Technically, the algorithm repeatedly swaps the target with the element on its left, until
- 1. It is no longer less than the element on its left, OR
- 2. It becomes the left-most element



- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - 2. Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - Shift items as needed to make room to insert the new addition

After 1st iteration

18 21 -4 27 7 2

[0] [1] [2] [3] [4] [5] **21** | 18 | -4 | 27 | 7 | 2

- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places

After 1st iteration

18 21 -4 27 7 2

[0] [1] [2] [3] [4] [5] **21** | 18 | -4 | 27 | 7 | 2

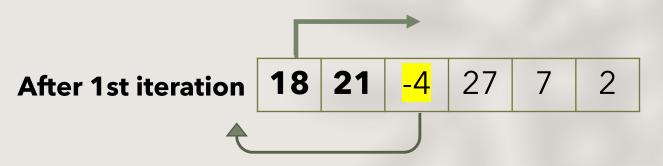
- The algorithm:
 - Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places

After 1st iteration

18 21 -4 27 7 2

[0] [1] [2] [3] [4] [5] **21 18** -4 27 7 2

- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places



[0] [1] [2] [3] [4] [5] **21 18** -4 27 7 2

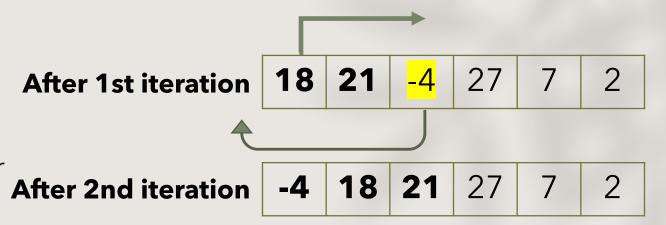
- The algorithm:
 - Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places

```
After 1st iteration 18 21 -4 27 7 2
```

Technically, the algorithm swaps arr[2] with arr[1] //[18, -4, 21, ... swaps arr[1] with arr[0] //[-4, 18, 21, ...

[0] [1] [2] [3] [4] [5] **21 18** -4 27 7 2

- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition
 - Repeat until all values are in their proper places



[0] [1] [2] [3] [4] [5] **21** | 18 | -4 | 27 | 7 | 2

- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places

After 1st iteration

18 21 -4 27 7 2

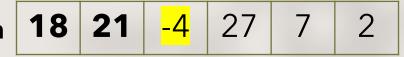
After 2nd iteration

-4 18 21 27 7 2

[0] [1] [2] [3] [4] [5] **21 18** -4 27 7 2

- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places

After 1st iteration



After 2nd iteration



After 3rd iteration



[0] [1] [2] [3] [4] [5] **21** | 18 | -4 | 27 | 7 | 2

- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places

After 1st iteration

18 21 -4 27 7 2

After 2nd iteration

-4 18 21 27 7 2

After 3rd iteration

-4 18 21 27 7 2

[0] [1] [2] [3] [4] [5] **21 18** -4 27 7 2

- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places







[0] [1] [2] [3] [4] [5] **21 18** -4 27 7 2

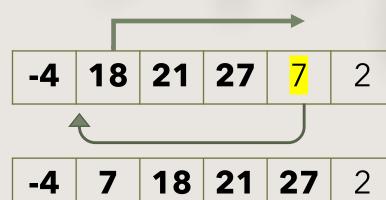
- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places











18

-4

[0] [1] [2] [3] [4] [5] **21 18** -4 27 7 2

- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places

After 1st iteration









7

18

21 | :

2



18 21 -4 27 7 2

-4 18 21 27 7 2

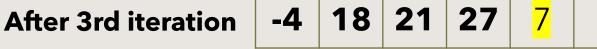
-4 18 21 27 7 2

[0] [1] [2] [3] [4] [5] **21 18** -4 27 7 2

- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places







After 4th iteration



[0] [1] [2] [3] [4] [5] **21 18** -4 27 7 2

- The algorithm:
 - 1. Pick the first value from the unsorted portion of the array
 - Traverse the sorted portion and look for an element smaller than what you just picked
 - If found, insert the new value after it
 - Otherwise, insert as the first value
 - 3. Shift items as needed to make room to insert the new addition
 - 4. Repeat until all values are in their proper places

After 1st iteration

18 21 -4 27 7 2

- After 2nd iteration
- ration | -4 | 18 | 21 | <mark>27</mark> | 7 | 2

After 3rd iteration

-4 | 18 | 21 | 27 | 7 | 2

- **After 4th iteration**
- er

After 5th iteration



```
import java.util.Arrays;
public class InsertionSort {
    public static void main(String[] args) {
                                                                              Insertion Sort
        int[] arr = {21, 18, -4, 27, 7, 2};
        System.out.println("Initial Array: " + Arrays.toString(arr));
        insertionSort(arr):
                                                                             Implementation
    public static void insertionSort(int[] arr) {
        // Notice the for loop starts at index 1, as the first element is naturally sorted
        for (int i = 1; i < arr.length; ++i) {
            // Starting with the first element in the unsorted part
            int j = i;
            // Traverse the sorted portion, and
            // repeatedly swap the target with the element on its left, until
            // 1. It is no longer less than the element on its left, OR
            // 2. It becomes the left-most element
            while (j > 0 && arr[j] < arr[j - 1]) {</pre>
                swap(arr, j, j - 1);
                --j;
            System.out.println("After iteration #" + i + ": " + Arrays.toString(arr));
    }
                                                                              $ javac SelectionSort.java
                                                                              $ java SelectionSort
    public static void swap(int[] arr, int i, int j) {
                                                                              Initial Array: [21, 18, -4, 27, 7, 2]
        int temp = arr[i];
                                                                              After iteration #1: [18, 21, -4, 27, 7, 2]
        arr[i] = arr[j];
                                                                              After iteration #2: [-4, 18, 21, 27, 7, 2]
        arr[j] = temp;
                                                                              After iteration #3: [-4, 18, 21, 27, 7, 2]
                                                                              After iteration #4: [-4, 7, 18, 21, 27, 2]
                                                                              After iteration #5: [-4, 2, 7, 18, 21, 27]
```

Q: Which sorting algorithm is used given the following output?

```
Initial Array: [25, -2, 9, 1, 8, 5, -5, 30]

After iteration #0: [-5, -2, 9, 1, 8, 5, 25, 30]

After iteration #1: [-5, -2, 9, 1, 8, 5, 25, 30]

After iteration #2: [-5, -2, 1, 9, 8, 5, 25, 30]

After iteration #3: [-5, -2, 1, 5, 8, 9, 25, 30]

After iteration #4: [-5, -2, 1, 5, 8, 9, 25, 30]

After iteration #5: [-5, -2, 1, 5, 8, 9, 25, 30]

After iteration #6: [-5, -2, 1, 5, 8, 9, 25, 30]
```

Selection Sort

Q: Sort the following array with 1) selection sort, and 2) insertion sort. Show the resulting array after each iteration.

[3, 6, 2, 10, 5, 1, 9]

```
Selection Sort:

After iteration #0: [1, 6, 2, 10, 5, 3, 9] After iteration #1: [3, 6, 2, 10, 5, 1, 9] After iteration #1: [1, 2, 6, 10, 5, 3, 9] After iteration #2: [2, 3, 6, 10, 5, 1, 9] After iteration #2: [1, 2, 3, 10, 5, 6, 9] After iteration #3: [2, 3, 6, 10, 5, 1, 9] After iteration #3: [1, 2, 3, 5, 10, 6, 9] After iteration #4: [2, 3, 5, 6, 10, 1, 9] After iteration #5: [1, 2, 3, 5, 6, 10, 9] After iteration #5: [1, 2, 3, 5, 6, 9, 10]
```

Binary Search

zyBook Chap 7.12

Binary Search

• Locates a **target value** in a **sorted** array by successively **eliminating half** of the array from consideration.

Binary Search (example – search for 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	67	72	85	98

$Binary\ Search\ (\text{example}-\text{search for 42})$

• Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)

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	67	72	85	98







- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!

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	67	72	85	98







Binary Search (example – search for 42)

- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!

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	<mark>30</mark>	36	42	50	56	67	72	85	98







- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!

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	<mark>30</mark>	36	42	50	56	67	72	85	98







- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!

Eliminate half of the array by resetting the lower or upper bound

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	<mark>30</mark>	36	42	50	56	67	72	85	98
	1								1								1

____ lowerBound midIndex

upperBound

- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!

Eliminate half of the array by resetting the lower or upper bound

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	<mark>30</mark>	36	42	50	56	67	72	85	98





- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!
- Step 3: Repeat until target is found or the range for consideration becomes empty

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	67	72	85	98



■ erBound

Binary Search (example – search for 42)

- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!
- Step 3: Repeat until target is found or the range for consideration becomes empty

value	-4	2	7	10	15	20	22	25	30	36	42	50	<mark>56</mark>	67	72	85	98
									love	erBo	und	m	1 idlad	OV			erBo

- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!
- Step 3: Repeat until target is found or the range for consideration becomes empty

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	<mark>56</mark>	67	72	85	98
										1			1				1

_ lowerBound

midIndex

upperBound

- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!
- Step 3: Repeat until target is found or the range for consideration becomes empty

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	<mark>56</mark>	67	72	85	98



Binary Search (example – search for 42)

- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!
- Step 3: Repeat until target is found or the range for consideration becomes empty

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	67	72	85	98



- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!
- Step 3: Repeat until target is found or the range for consideration becomes empty

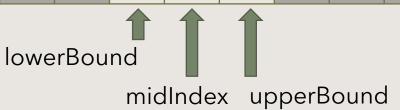
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	<mark>42</mark>	50	56	67	72	85	98
lowerBound																	

midIndex upperBound

- Step 1: Find the indices of 1) lower bound, 2) upper bound, 3) middle ((lower + upper) / 2)
- Step 2: Compare the target value with the element at the mid index
 - target < arr[midIndex], eliminate the subarray on the right-hand side (including the midIndex)
 - target > arr[midIndex], eliminate the subarray on the left-hand side (including the midIndex)
 - target == arr[midIndex], found!
- Step 3: Repeat until target is found or the range for consideration becomes empty

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	<mark>42</mark>	50	56	67	72	85	98

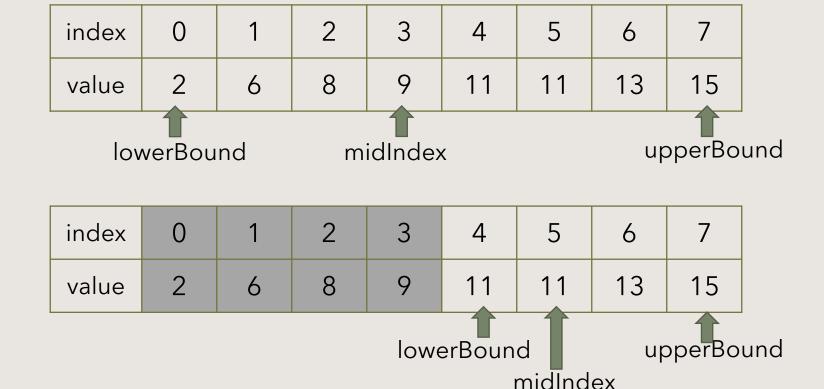
Return the midIndex



```
// Precondition: Elements in the array are in sorted order
public static int binarySearch(int[] arr, int target) {
   // The boundaries of the array indices
   int lowerBound = 0;
   int upperBound = arr.length - 1;
   // While this is not an empty array ( lowerBound == upperBound suggests a size 1 array )
   while (lowerBound <= upperBound) {</pre>
       // Set the midIndex given the lowerBound and upperBound
       int midIndex = (lowerBound + upperBound) / 2;
       // If the element at the midIndex is less than the target, that is,
       // the target is somewhere in the sub-array on the right-hand side
       if (arr[midIndex] < target) {</pre>
           // Set the lowerBound to be the first element of the RHS sub-array
           lowerBound = midIndex + 1;
       // If the element at the midIndex is greater than the target, that is,
       // the target is somewhere in the sub-array on the left-hand side
       else if (arr[midIndex] > target) {
           // Set the upperBound to be the last element of the LHS sub-array
           upperBound = midIndex - 1;
       // If the element at the midIndex equals the target, that is,
       // the target is found!
       else {
                                                                      Binary Search
           return midIndex;
                                                                      Implementation
    return -1; // If the target is not found
```

```
import java.util.Scanner;
import java.util.Arrays;
public class BinarySearch {
                                                                  Binary Search
    public static void main(String[] args) {
        int[] arr = {2, 6, 8, 9, 11, 11, 13, 15};
                                                                  Implementation
        Scanner input = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int target = input.nextInt();
        int foundAt = binarySearch(arr, target);
        if (foundAt !=-1) {
            System.out.println("Number " + target + " is found at index " + foundAt
                               + " in the array " + Arrays.toString(arr) + ".");
        } else {
            System.out.println("Number " + target + " cannot be found in the array "
                               + Arrays.toString(arr) + ".");
    public static int binarySearch(int[] arr, int target) {
        // See previous slide...
                                                  $ javac BinarySearch.java
                                                  $ java BinarySearch
                                                  Enter an integer: 12
                                                  Number 12 cannot be found in the array [2, 6, 8, 9, 11, 11, 13, 15].
                                                  $ java BinarySearch
                                                  Enter an integer: 11
                                                  Number 11 is found at index 5 in the array [2, 6, 8, 9, 11, 11, 13, 15].
```

Q: Determine if the number 11 exists in the following array with Binary Search. If yes, which index will this algorithm return? Visualize each iteration.



Found at index 5