



Java Programming

3-4

Sorting and Searching



Overview

This lesson covers the following topics:

- Recognize the sort order of primitive types and objects
- Trace and write code to perform a simple Bubble Sort of integers
- Trace and write code to perform a Selection Sort of integers
- Trace and write code to perform a Binary Search of integers
- Compare and contrast search and sort algorithms
- Analyze the Big-O for various sort algorithms

Sorting Arrays

- There are many different algorithms for sorting arrays, some are easier to code, and some have a faster computation time.

An algorithm is a logical computational procedure that if correctly applied, ensures the solution to a problem.

- When searching for elements in an array, it is beneficial to sort your array in lexicographical order to reduce time spent searching through the array.

Lexicographical order is an order based on the ASCII value of characters. The table of values can be found at <http://www.asciitable.com/>

- ASCII values can be used to sort any character(letter, **ORACLE** number or symbol).

Main Sorting Algorithms for Arrays

- There are many sorting algorithms for arrays.
- Some of the common algorithms are:
 - Selection sort
 - Bubble sort
 - Insertion sort
 - Merge sort
 - Quick sort
- We will look at the pros and cons of these in the remaining slides.
- These algorithms are transferrable between languages.

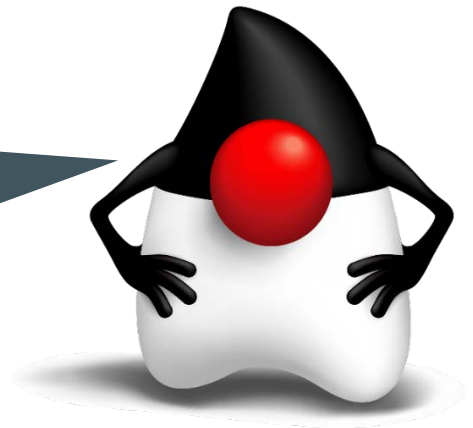


Selection Sort

Selection sort is a simple sorting algorithm that works as follows:

- Find the minimum value in the list.
- Swap it with the value in the starting position.
- Repeat the steps above for the remainder of the list, starting at the second position and advancing each time.

Before reading on try
and list what is good
and bad about this
sorting technique.

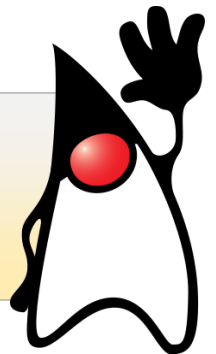


Selection Sort

This algorithm:

- Is inefficient on large arrays because it has to iterate through the entire array each time to find the next smallest element.
- Is preferred for smaller arrays because of its simplicity.

Not only is it simple, but it also uses very little additional memory. So if you were on a system that had access to very little memory then this would produce little overhead.



Steps to the Selection Sort Algorithm

- To sort the following array:

{5, 7, 2, 15, 3}

- Step 1: Find the smallest value, which is 2.
- Step 2: Swap the value with the starting position:

{2, 7, 5, 15, 3}

- Repeat steps 1 and 2 starting with the next position in the list. The next smallest number is 3.
- When swapped for the next position, the array is:

{2, 3, 5, 15, 7}

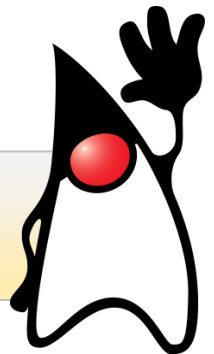
Steps to the Selection Sort Algorithm

- These steps are repeated until the array is sorted, or to be more specific, when `array.length - 1` is reached.
- At that point, if the second last element was bigger than the last, it would swap and the array would be sorted.
- Otherwise, the array is sorted.

{2, 3, 5, 7, 15}

Watch this Selection Sort video:

<https://www.youtube.com/watch?v=Ns4TPTC8whw>





Selection Sort Example

1. Create a project named sortexample.
2. Create a SortExample class that includes the following methods and declares an array of integers.

```
public class SortExample {  
    public static void main(String[] args) {  
        int[] numbers = {40, 7, 59, 4, 1};  
    }//end method main  
  
    static void selectionSort(int[] numbers) {  
    }//end method selectionSort  
  
    static void displayValues(int[] numbers) {  
    }//end method displayValues  
  
}//end class SearchExample
```



Selection Sort Example

3. Use a traditional for loop to display the contents of the array in the displayValues method.
4. Call the displayValues() method from main.

```
public static void main(String[] args) {  
    int[] numbers = {40, 7, 59, 4, 1};  
    displayValues(numbers);  
} //end method main  
  
static void selectionSort(int[] numbers) {  
} //end method selectionSort  
  
static void displayValues(int[] numbers) {  
    for(int i = 0; i < numbers.length; i++){  
        System.out.print(numbers[i] + " ");  
    } //endfor  
    System.out.println("\n");  
} //end method displayValues  
} //end class SearchExample
```



Selection Sort Example

5. Add the selection sort code to the selectionSort() method.

```
static void selectionSort(int[] numbers) {  
    int indexMin = 0; //the index of the smallest number  
    for(int i = 0; i < numbers.length; i++){  
        indexMin = i;  
        for(int j = i + 1; j < numbers.length; j++){  
            if(numbers[j] < numbers[indexMin]){//if we find a smaller int,  
                indexMin = j;                //set it as the min  
            }  
        }  
        //we have the index of the smallest int and can swap the values  
        int temp = numbers[i]; //use temp to store the value  
        numbers[i] = numbers[indexMin];  
        numbers[indexMin] = temp;  
    }  
}
```



Selection Sort Example

6. Update the main method to display the array, then call the selection sort and display the sorted array.

```
public class SortExample {  
  
    public static void main(String[] args) {  
        int[] numbers = {40, 7, 59, 4, 1};  
  
        displayValues(numbers);  
        selectionSort(numbers);  
        displayValues(numbers);  
    } //end method main
```

- Output

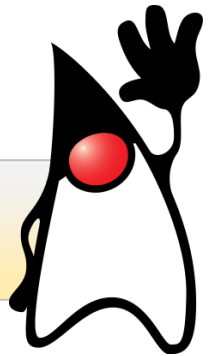
40 7 59 4 1

1 4 7 40 59

Bubble Sort

- Bubble sort, also known as exchange sort, is another sorting algorithm that is simple but inefficient on large lists.
- The algorithm works as follows:
 - Compare 2 adjacent values (those at indexes 0 and 1).
 - If they are in the wrong order, swap them.
 - Continue this with the next two adjacent values (those at indexes 1 and 2) and on through the rest of the list.
 - Repeat steps 1 through 3 until array is sorted.

Bubble sort is sometimes referred to as sinking sort. Try to think what would be good and bad about this algorithm.



Bubble Sort Video

Watch this Bubble Sort video:

— <http://www.youtube.com/watch?v=lyZQPjUT5B4>



Bubble Sort Example

- To sort the following array:

```
{40, 7, 59, 4, 1}
```

- First, compare and swap the first two values:

```
{7, 40, 59, 4, 1}
```

- Then, compare and swap the next two values:

```
{7, 40, 59, 4, 1}
```

- Notice that the swap did not occur in the second comparison because they are already in correct order.
- Repeat these steps until the array is sorted.

```
{1, 4, 7, 40, 59}
```


Bubble Sort Array Example

Step by step, the array would look as follows.

```
{40, 7, 59, 4, 1} //Starting array
{7, 40, 59, 4, 1} //7 and 40 swapped
{7, 40, 59, 4, 1} //40 and 59 did not swap
{7, 40, 4, 59, 1} //59 and 4 swapped
{7, 40, 4, 1, 59} //1 and 59 swapped
//First pass through array is complete
{7, 40, 4, 1, 59} //7 and 40 did not swap
{7, 4, 40, 1, 59} //40 and 4 swapped
{7, 4, 1, 40, 59} //40 and 1 swapped
{7, 4, 1, 40, 59} //40 and 59 did not swap
//Second pass is complete
{4, 7, 1, 40, 59} //7 and 4 swapped
{4, 1, 7, 40, 59} //7 and 1 swapped
{4, 1, 7, 40, 59} //7 and 40 did not swap
//Third pass is complete
{1, 4, 7, 40, 59} //1 and 4 swapped
{1, 4, 7, 40, 59} //4 and 7 did not swap
//Fourth pass is complete and the array is sorted
```



Bubble Sort Example

1. In your SortExample class create a method under the main method named bubbleSort().

```
//end method main

static void bubbleSort(int[] numbers) {
    for(int j =0 ; j < numbers.length; j++){
        for(int i = 0; i < numbers.length-1; i++){
            //if the numbers are not in order
            if(numbers[i] > numbers[i+1]){
                //swap the numbers
                int temp = numbers[i];
                numbers[i] = numbers[i+1];
                numbers[i+1] = temp;
            }//endif
        }//endfor
    }//endfor
}//end method bubbleSort
```

The array is sorted when you pass through all of the elements and none are swapped.



Bubble Sort Example

2. Update the main method to call the `bubbleSort()` instead of the `selection sort()`.

```
public class SortExample {  
    public static void main(String[] args) {  
        int[] numbers = {40, 7, 59, 4, 1};  
  
        displayValues(numbers);  
        //selectionSort(numbers);  
        bubbleSort(numbers);  
        displayValues(numbers);  
    } //end method main  
}
```

Bubble sort is a simple sorting method but suffers greatly with performance.

- Output

40 7 59 4 1

1 4 7 40 59



Optimized Bubble Sort Example

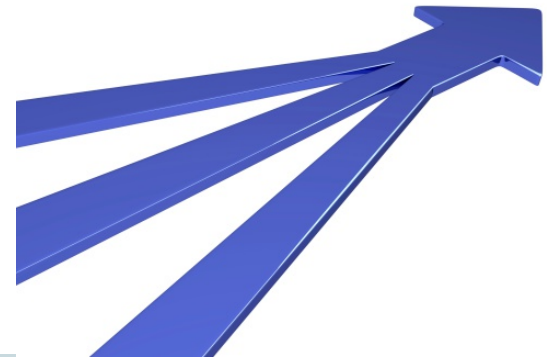
```
static void bubbleSort(int[] numbers) {  
    for(int i=0; i<numbers.length; i++)  
    {  
        boolean flag = false;  
        for(int j=0; j<numbers.length-i-1; j++)  
        {  
            if(numbers[j]>numbers[j+1])  
            {  
                flag = true;  
                int temp = numbers[j+1];  
                numbers[j+1] = numbers[j];  
                numbers[j] = temp;  
            }  
        }  
        if(!flag){//No Swapping happened, array is sorted. Exit.  
            return;  
        }  
    }  
}
```

It can be modified so that it breaks out of the for loops if there is no swap detected.

Merge Sort

Merge sort:

- Is more complex than the previous two sorting algorithms.
- Can be more efficient because it takes advantage of parallel processing.
- Takes on a "divide and conquer" technique, which allows it to sort arrays with optimal speed.



Merge Sort Algorithm

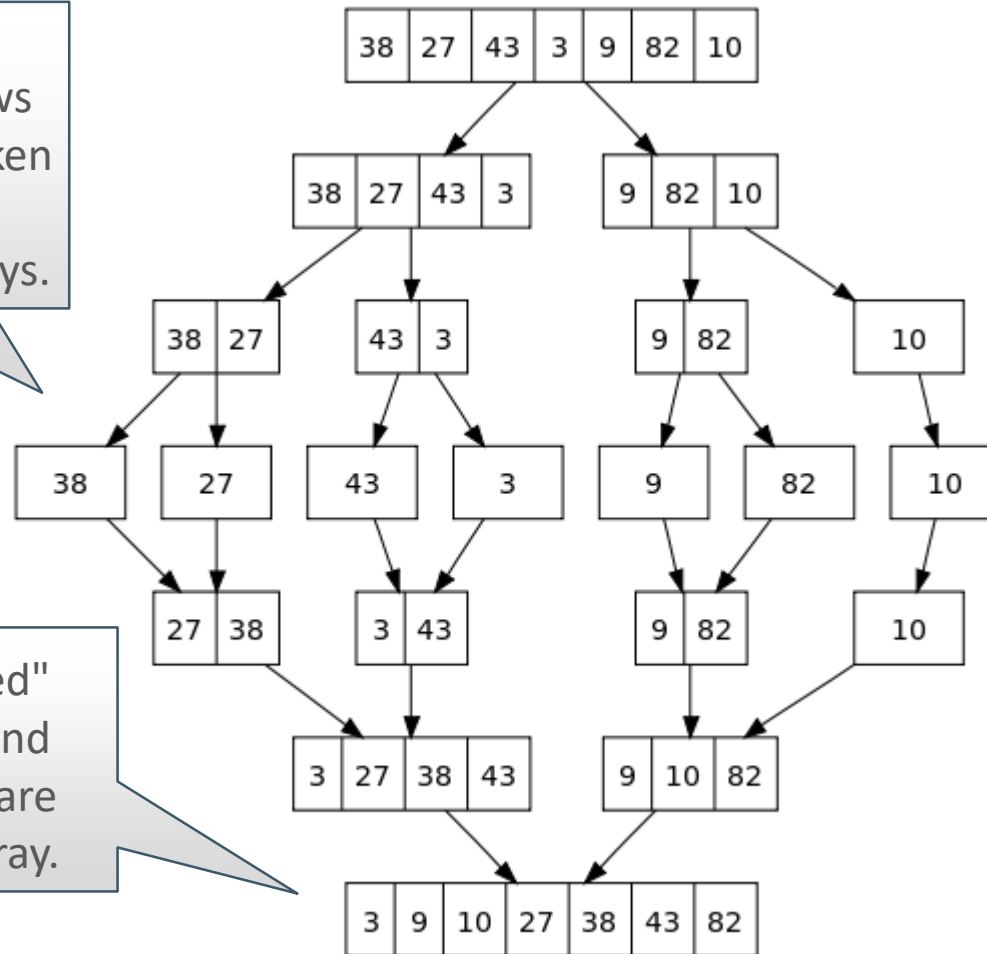
The algorithm works as follows:

- Divide the unsorted list into sub-lists, each containing one element (a list of one element is considered sorted).
- Repeatedly merge sub-lists to produce new sub-lists until there is only one sub-list remaining.
- This will be the sorted list.

Visual Representation of Merge Sort

This visual representation shows the array being broken down into the one element sorted arrays.

They are "merged" together again and again until they are in one sorted array.



Learn More About Merge Sort

- This lesson focuses on the theory of merge sort since it is so complex.
- Watch the following Merge Sort video:
 - https://www.youtube.com/watch?v=XaqR3G_NVoo
- You can find good examples of merge sort implementations online.
- If you look at the code you will see it is a lot more complicated than the selection and bubble sorts.

To see the ones covered and a few others a good resource is - <http://www.sorting-algorithms.com/> This demonstrates each sorting pass for the algorithms visually.



Searching Through Arrays

- Sorting an array makes searching faster and easier.
- There are two basic searching methods:
 - Sequential/Linear searches
 - Binary searches
- For example, in a sorted array of student names, you want to know if Juan is in your class.
- You could search the array, or find out exactly where his name is in alphabetical order with the other students.

Within programming you often find yourself having to search through a data structure to locate a particular element.



Sequential Search


- A sequential search is an iteration through the array that stops at the index where the desired element is found.
- If the element is not found, the search stops after all elements of the list have been iterated through.
- This method works for sorted or unsorted arrays, but is not efficient on larger arrays.
- However, a sequential search is not more efficient if the list is sorted.



Sequential Search

- A sequential search starts at the first element, then the second and so on.
- If the value being searched for is near the start, then it will be found quicker than if it was at the end.

Index	0	1	2	3	4	5	6	7	8	9	10
Value	25	10	18	47	13	6	57	38	81	85	7



13 X X X X ✓



Sequential Search Example

1. Create a method called `sequentialSearch` in the `SortExample` class.

```
    }//end method main

    static void sequentialSearch(int[] numbers, int value) {
        for(int i = 0; i < numbers.length; i++) {
            if(numbers[i] == value) {
                System.out.println("The number " + value + " is at position "
                                   + i + " in the list");

                return;
            }//endif
        }//endfor
        System.out.println("The number " + value + " is not in the list");
    }//end method sequentialSearch
```

The value of `i` could have been returned if the value was going to be updated somewhere else in the code.





Sequential Search Example

2. Update the main method to match the following example which carries out two searches.

```
public static void main(String[] args) {  
    int[] numbers = {40, 7, 59, 4, 1};  
  
    displayValues(numbers);  
    sequentialSearch(numbers, 13);  
    sequentialSearch(numbers, 7);  
    //selectionSort(numbers);  
    bubbleSort(numbers);  
    displayValues(numbers);  
} //end method main
```

- Output
40 7 59 4 1
The number 13 is not in the list
The number 7 is at position 1 in the list
1 4 7 40 59

Binary Search

- Binary searches can only be performed on sorted data.
- To see how a binary search works:
 - Search for the target value 76 in the array.
 - Step 1: Identify the low, middle, and high elements in the array.
 - Low is 0, high is array.length = 11, middle is $(\text{high} - \text{low}) / 2 = 5$
 - Step 2: Compare the target value with the middle value.
 - 76 is greater than 56.

Index	0	1	2	3	4	5	6	7	8	9	10
Value	2	10	16	34	37	56	57	76	81	83	85

76



Using Binary Search With Sorted Arrays

- To see how a binary search works:
 - Step 3: Since the target value is greater than the middle value, it is to the right of the middle. Set the low index to middle + 1, then calculate the new middle index.
 - Middle is $((5 + 1) + 10) / 2 = 8$
 - Step 4: Compare the target value with the middle value.
 - 76 is less than 81.

Index	0	1	2	3	4	5	6	7	8	9	10
Value	2	10	16	34	37	56	57	76	81	83	85

76



Using Binary Search With Sorted Arrays

- To see how a binary search works:
 - Step 5: Since the target value is less than the middle value, it is to the left of the middle. Set the high index to middle - 1 then calculate the new middle index.
 - Middle is $((8 - 1) + 6) / 2 = 7$
 - Step 6: Compare the target value with the middle value.
 - 76 is equal to 76.
 - Step 7: The target value has been found at index 7.

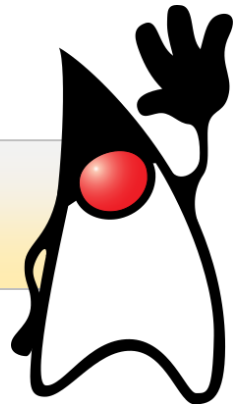
Index	0	1	2	3	4	5	6	7	8	9	10
Value	2	10	16	34	37	56	57	76	81	83	85

76

Decide What to Return in a Binary Search Method

- When writing a binary search method you will have to decide what to return.
 - Index
 - Boolean
 - Value
 - Entire object
 - One field of the object

What would you return if the value was not found?





Binary Search Example

```
static public void binarySearch(int[] numbers, int value){
    int low = 0;
    int high = numbers.length - 1;
    while(high >= low){
        int middle = (low + high)/2; // Middle index
        if(numbers[middle] == value){
            System.out.println("The number "+ value + " is at position "
                               + middle + " in the list");
            return; //Target value was found
        }//endif
        if(numbers[middle] < value){
            low = middle + 1;
        }//endif
        if(numbers[middle] > value){
            high = middle - 1;
        }//endif
    }//endwhile
    System.out.println("The number "+ value + " is not in the list");
    //The value was not found
} //end method binarySearch
```

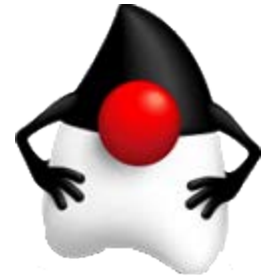
1. Create a method called binarySearch that will display a value if found.



Binary Search Example

2. Update the main method to match the following.
3. The first binary search will not work as it is being executed on an unsorted array!

```
public static void main(String[] args) {  
    int[] numbers = {40, 7, 59, 4, 1};  
  
    displayValues(numbers);  
    //sequentialSearch(numbers, 13);  
    //sequentialSearch(numbers, 7);  
    binarySearch(numbers, 7);  
    selectionSort(numbers);  
    binarySearch(numbers, 7);  
    //bubbleSortOptimized(numbers);  
    //displayValues(numbers);  
} //end method main
```



Binary Search Task

Quick Task: return values from a binary search.

1. Update the `binarySearch` method to return a true value if the value is found or false if it is not.
2. Use the return value from the functional method in an if statement in main to say the value is found or the value is not found.
3. Update the `binarySearch` method to return the index of the value if it is found or -1 if it is not.
4. Use the integer return value to display the position of the value or value not found message.

Binary Search Suggested Solution



```
static boolean binarySearch(int[] numbers, int value){  
    int low = 0;  
    int high = numbers.length - 1;  
  
    while(high >= low){  
        int middle = (low + high)/2; // Middle index  
  
        if(numbers[middle] == value){  
            return true; // Target value was found  
        }//endif  
        if(numbers[middle] < value){  
            low = middle + 1;  
        }//endif  
        if(numbers[middle] > value){  
            high = middle - 1;  
        }//endif  
    }//endwhile  
    return false; // The value was not found  
}//end method binarySearch
```

1. Binary search
that returns a
Boolean value.



Binary Search Suggested Solution

2. Use the return value from the functional method in an if statement in main to say the value is found or the value is not found.

```
public static void main(String[] args) {  
    int[] numbers = {40, 7, 59, 4, 1};  
  
    displayValues(numbers);  
    selectionSort(numbers);  
    if(binarySearch(numbers, 7))  
        System.out.println("The value is found");  
    else  
        System.out.println("The value is not found");  
    //endif  
} //end method main
```

Binary Search Suggested Solution



```
static int binarySearch(int[] numbers, int value){
    int low = 0;
    int high = numbers.length - 1;

    while(high >= low){
        int middle = (low + high)/2; // Middle index

        if(numbers[middle] == value){
            return middle; // Target value was found
        }//endif
        if(numbers[middle] < value){
            low = middle + 1;
        }//endif
        if(numbers[middle] > value){
            high = middle - 1;
        }//endif
    }//endwhile
    return -1; // The value was not found
}//end method binarySearch
```

3. Update the binarySearch method to return the index of the value if it is found or -1 if it is not.



Binary Search Suggested Solution

4. Use the integer return value to display the position of the value or value not found message.

```
public static void main(String[] args) {  
    int[] numbers = {40, 7, 59, 4, 1};  
  
    displayValues(numbers);  
    selectionSort(numbers);  
    int found = binarySearch(numbers, 7);  
    if(found != -1)  
        System.out.println("The value is found at position " + found);  
    else  
        System.out.println("The value is not found");  
    //endif  
} //end method main
```


Comparison of Sorting Algorithms

- From https://en.wikipedia.org/wiki/Big_O_notation

Name	Best	Average	Worst	Memory	Stable	Method
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends	Partitioning
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends; worst case is n	Yes	Merging
In-place Merge sort	—	—	$n (\log n)^2$	1	Yes	Merging
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection
Insertion sort	n	n^2	n^2	1	Yes	Insertion
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection
Selection sort	n^2	n^2	n^2	1	No	Selection
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging
Shell sort	n	$n(\log n)^2$ or $n^{3/2}$	Depends on gap sequence; best known is $n(\log n)^2$	1	No	Insertion
Bubble sort	n	n^2	n^2	1	Yes	Exchanging
Binary tree sort	n	$n \log n$	$n \log n$	n	Yes	Insertion
Cycle sort	—	n^2	n^2	1	No	Insertion
Library sort	—	$n \log n$	n^2	n	Yes	Insertion
Patience sorting	—	—	$n \log n$	n	No	Insertion & Selection
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection
Strand sort	n	n^2	n^2	n	Yes	Selection
Tournament sort	—	$n \log n$	$n \log n$	$n^{[5]}$		Selection
Cocktail sort	n	n^2	n^2	1	Yes	Exchanging
Comb sort	n	$n \log n$	n^2	1	No	Exchanging
Gnome sort	n	n^2	n^2	1	Yes	Exchanging
Bogosort	n	$n \cdot n!$	$n \cdot n! \rightarrow \infty$	1	No	Luck

Big-O Notation

- Big-O Notation is used in Computer Science to describe the performance of Sorts and Searches on arrays.
- The Big-O is a proportion of speed or memory usage to the size of the array.
- In the previous slide, Big-O examples are:
 - n
 - $n \log n$
 - n^2
- Compare the values of these sorts.
 - Which sort(s) is/are quickest when n is 100?
 - Which sort(s) is/are quickest when n is 1 Billion?



Big-O Notation

- $O(n)$ represents linear time. That the size of n affects the growth of the algorithm proportionally. So for a for loop that iterates n times, then as n grows the number of iterations grows proportionally.
- n^2 would be the same as having a nested loop where both iterate n times. So for every iteration of the first loop, the nested loop would run n times.



Big-O Notation

- $O(\log n)$ – means that you don't have to look at all the inputs. This is when we don't always have to iterate through a whole data structure. Think of a phone book and you want to look up someone with a name starting with N. You would start roughly at the middle and see if you are before or after N. Then repeat this step again, depending on where you are (binary search idea). You have not had to search through all of the elements. So on average its performance is better than a linear search.



Terminology

Key terms used in this lesson included:

- Array
- Binary search
- Bubble Sort
- Lexicographical order
- Merge Sort
- Selection Sort
- Sequential search

Summary

In this lesson, you should have learned how to:

- Recognize the sort order of primitive types and objects
- Trace and write code to perform a simple Bubble Sort of integers
- Trace and write code to perform a Selection Sort of integers
- Trace and write code to perform a Binary Search of integers
- Compare and contrast search and sort algorithms
- Analyze the Big-O for various sort algorithms

