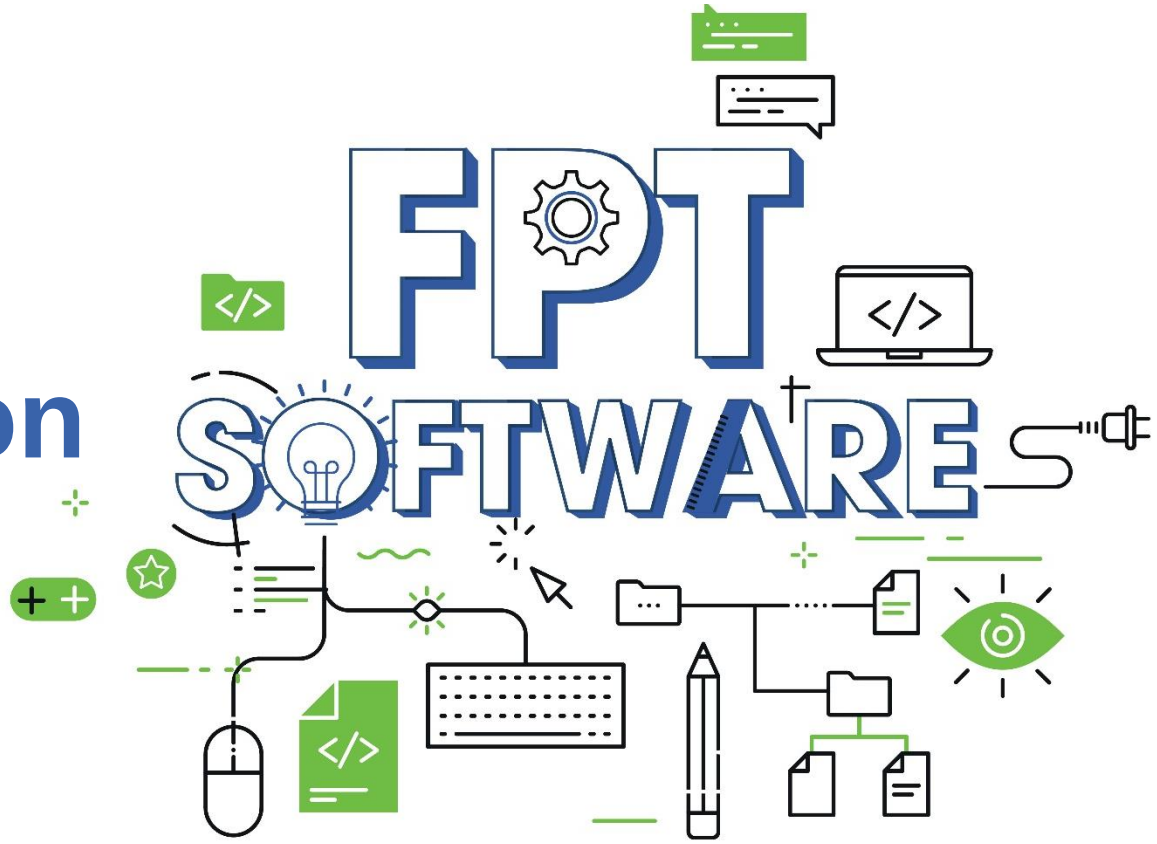
Array and Big O Notation

Fsoft Academy



Lesson Objectives

- **Understand** the basics of Big-O notation
- **Analyze** the time complexity of simple algorithms
- **Compare** the time complexity of different algorithms for the same problem
- **Apply** Big-O notation to real-world problems

Agenda

1. A Quick Review of Array in Java
2. Arrays in Memory
3. Big-O Notation
4. Big-O Values for Array Operations

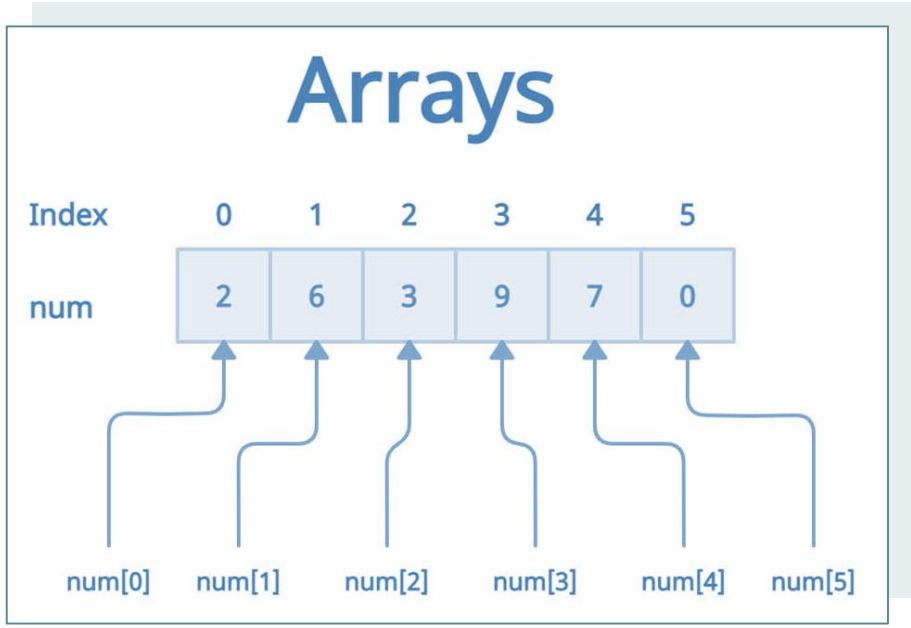
Section 1

A Quick Review of Array in Java

What are Arrays in Java?

Arrays in Java store multiple values of the same type in a single variable.

- **Arrays** can hold a fixed number of elements, and each element has an index used to access it.
- Arrays enable developers to manipulate data efficiently and perform operations like ***sorting***, ***searching***, ***insertion***, and ***deletion*** quickly and easily.
- There are multiple arrays applications, including *implementing mathematical operations*, *arrays as collections, and arrays as data structures.* You can also use arrays to *store objects.*

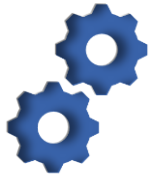


Uses of Arrays in Java

- **There are various uses of java array, such as:**

Store multiple elements of the same type in a single array.

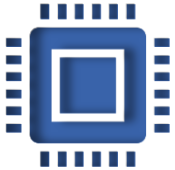
Sorting, searching, accessing, and manipulating data.



Provide memory efficiency, better performance, and faster execution time.

Solve complex problems (calculating the sum of finding values in an arrays).

Implement multi-dimensional arrays, useful for representing matrices and graphs.



Applications in computer programming, engineering, mathematics etc.

Types of Array in Java

Single Dimensional Array

- A type of linear array that stores data sequentially.
- Can only store values in one dimension.
- Declared using square brackets `[ ]` followed by the datatype.

Multi-Dimensional Array

- Contain *more than one* dimension, such as **rows**, **columns**, etc.
- Can be visualized as tables with rows and columns where each element is accessed through its position.

Advantages of Array in Java

- **Array can be very useful in Java because:**

Easy to use

Arrays require less coding than traditional data structures, making arrays a great choice for rapid development.

High performance

Arrays provide fast and efficient access to elements as compared to other data structures.

Memory efficient

Arrays can store multiple values in the same location. This reduces the amount of RAM required to store data and improves overall performance.

Random Access

Arrays support random access (elements can be accessed directly using their index). This makes arrays an ideal choice for applications requiring fast data access.

Disadvantages of Array in Java

- **However, Array also have several downsides as follow:**

disadvantage

Fixed-Sized

Arrays have a fixed size and are allocated at compile-time.

This means that their memory size cannot be changed during program execution. If you need to dynamically add or remove elements, use other data structures like Array List.

Lack of Flexibility

Arrays don't provide many features like ***sorting*** or ***searching***, which makes the data more accessible. The elements in an array is accessed using indexes.

Overhead

Arrays in Java can have certain overhead, including the time to look up a particular item, add or delete items, or rearrange arrays if needed.

Declaration an Array in Java

- **Java** follows a simple logic for ***declaring*** an array object - all you need to do is indicate the data type of each element and name the variable, along with adding some *square brackets* `[]` to signal that it's indeed an array.

❑***Example:***

```
int intArray[];
int[] intArray;
```

The first one is the older syntax, while the second one is preferred by most of Java developers.

Instantiating an Array in Java

- Arrays in Java are objects whose default initialization value is null;
- When an array is declared, only a reference of an array is created. To create or give memory to the array, you create an array like this: The general form of *new* as it applies to one-dimensional arrays appears as follows:

```
var-name = new type [size];
```

array is an object

- **Example:**

```
//declaring array
int intArray[];
// allocating memory to array
intArray = new int[20];
// combining both statements in one
int[] intArray = new int[20];
```

Initialization an Array in Java

- When arrays are instantiated, they already have values assigned to each element.
- ***Initialization*** only becomes necessary if you want to reassign new values to the array elements or add more elements through an initialization statement. This is done with a simple assignment statement:

```
intArray = {4,5,6};
```

This example initializes the arrays intArray with three new elements, 4, 5 and 6.

Scope of Arrays

The scope of arrays in Java is vast. They are ***one of the most fundamental concepts*** in programming languages.

- Arrays allow us to store and manipulate data efficiently.
- Arrays have a range of advantages, including their simplicity and flexibility for static and dynamic arrays.
- Arrays can be used in various applications such as database management systems, image processing, searching algorithms, and more. Besides this, cloud computing and big data rely heavily on arrays for their operations.

Challenges Faced While Using Arrays

Size Limitations:

Once an array is created, the number of elements stored within the array cannot be changed. This can be problematic if the number of elements that need to be stored is unknown or varies frequently.

Incompatibility:

Arrays are not always compatible with all data types and structures, meaning an array cannot necessarily store different types of objects.

Speed Issues:

Arrays are stored in a linear format, and thus searching for or manipulating specific elements requires going through each element one by one.

Data Limitations:

Arrays cannot store more complex data types, such as objects or structures with keys and values. This limits the applications of arrays.

Addition using Java Arrays

❑***Example***: *Find the sum of all the elements in an array.*

```
public class Main {
   public static void main(String[] args) {
      int my_array[] = { 3, 10, 15, 20, 8, 4, 10, 10 };
      int sum = 0;

      for (int i : my_array) {
         sum += i;
      }
      System.out.println("The sum is " + sum);
   }
}
```

```
The sum is 80
```

Multiplication using Java Arrays

❑***Example***: *Find the sum of all the elements in an array.*

```
public class Main {
   public static void main(String[] args) {
     int my_array[] = { 1, 2, 3, 4 };
     int mul = 1;

     for (int i : my_array) {
       mul *= i;
     }

     System.out.println("The product is " + mul);
   }
}
```

```
The product is 24
```

Copying using Java Arrays

- You can copy one array to another by using **Arrays.copyOf()** method.

```
import java.util.Arrays;
public class Main {
  public static void main(String args[]) {
    int a[] = { 1, 2, 3, 4, 5 };
    int b[] = new int[a.length];

    // copying one array to another
    b = Arrays.copyOf(a, a.length);

    // printing array
    for (int i = 0; i < b.length; ++i) {
       System.out.print(b[i] + " ");
    }
  }
}
```

*The above piece of code will store the elements of the array **"a"** in the newly created array **"b".***

Addition using Java Arrays

- Java supports object cloning with the help of the **clone()** method to create an exact copy of an object.

```
class Main {
  public static void main(String args[]) {
    int arr[] = { 33, 3, 4, 5 };
    System.out.println("Original array:");

    for (int i : arr) {
       System.out.println(i);
    }

    System.out.println("Clone of the array:");
    int clone_arr[] = arr.clone();

    for (int i : clone_arr) {
       System.out.println(i);
    }

    System.out.println("Are both equal?");
    System.out.println(arr == clone_arr);
  }
}
```

```
Original array:
33
3
4
5
Clone of the array:
33
3
4
5
Are both equal?
false
```

Section 2

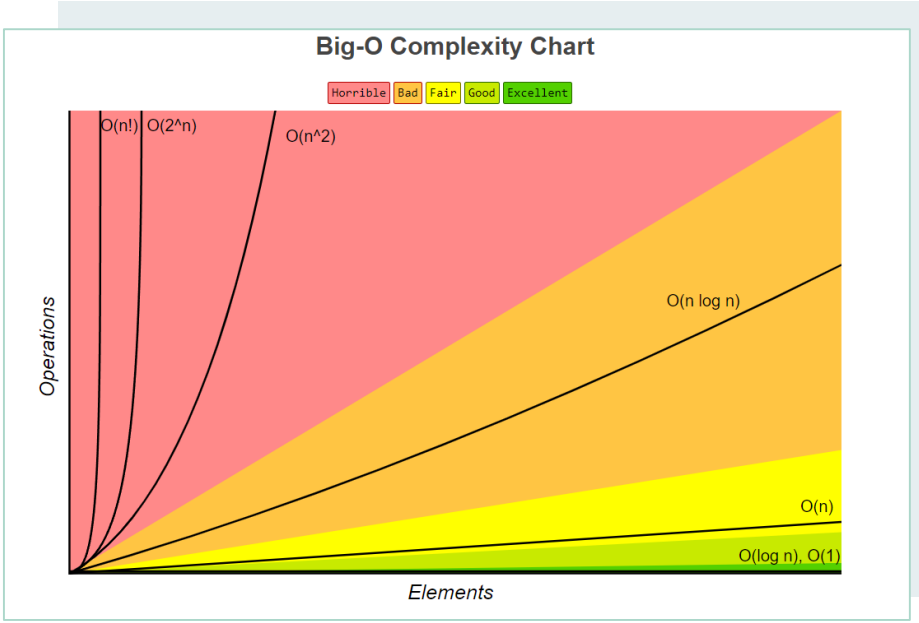
Big-O Notation

What is Big(O) Notion?

The Big-O Notation is a fundamental concept to help us measure the time complexity and space complexity of the algorithm.

- The Big-O notation is **NOT** 100% accurate. Instead, it's an estimate of how much time and space your algorithm will take.
- Another rule is that the Big-O notation will be calculated considering *the worst-case scenario.*

❖ *In any coding interview, you will be required to know Big(O) notation. That will help you to get your dream job.*



Asymptotic Notations Tiệm cận

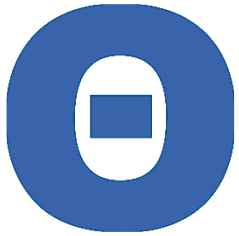
The asymptotic notation is a defined pattern to measure the performance and memory usage of an algorithm.

Asymptotic notation được dùng để biểu diễn "tốc độ tăng trưởng" của hàm số khi đầu vào (n) tiến đến vô hạn.

- Though the **Omega** and **Theta** notations are not very much used in coding interviews.

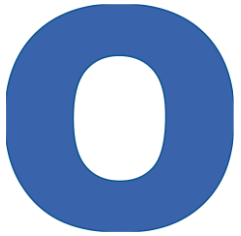
Omega Notation:

best-case scenario where the time complexity will be as optimal as possible based on the input.



Theta Notation:

average-case scenario where the time complexity will be the average considering the input.

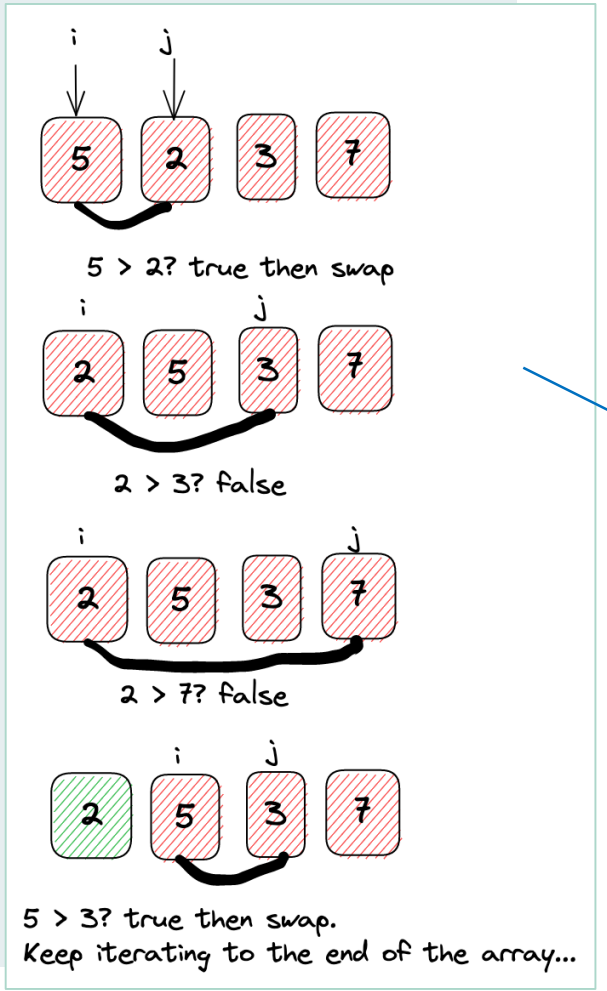


Big-O Notation:

worst-case scenario and the most used in coding interviews. It's the most important operator to learn because we can measure the worst-case scenario time complexity of an algorithm.

Asymptotic Notations – Bubble Sort Algorithm

- The basic idea of the bubble sort is to iterate over the array, iterate again and compare every element.
- If the element from the first looping is lower than the other looping element, then swap the elements.



***Almost all elements** will be compared and that's why it's so inefficient in terms of performance.*

Asymptotic Notations – Bubble Sort Algorithm

The best-case scenario (Omega notation Ω)

- the array is already **sorted.**
- In this case, it would be necessary to run through the array only once.
- Therefore, the time complexity would be Ω(n).

The average-case scenario (Theta notation Θ)

- the array has only **a couple of elements unordered.**
- In this case, it wouldn't be necessary to run through the whole algorithm.
- Even though this is a bit better, the time complexity is still considered as O(n ^ 2).

The worst-case scenario (Big O notation)

- the array is **fully unsorted and the whole array has to be traversed.**
- It would be necessary to sort the whole array.
- the time complexity would be O(n ^ 2).

Asymptotic Notations – Bubble Sort Algorithm

- The space complexity from the Bubble sort algorithm will be always O(1) since we only change values in place from an array.

Changing values in place (tại chỗ) means that we don't need to create a new array, we only change array values.

Constant – O(1)

- In practice, we need to measure the Big(O) notation by checking the number of an elementary operation. But notice that this is a constant time. It doesn't depend on any external number.
- When assigning a variable this will take the time complexity of O(1) because it's only one operation. Remember that the Big (O) notation will not measure precisely the performance of the algorithm.

➢ Therefore, the O(1) time complexity is an abstract way to measure code performance.

```
int number = 7; // O(1) time and space complexity
```

Constant – O(1)

❑ ***Example:***

```
public class O1Example {

 public static void main(String[] args) {

 int num1 = 10;
 int num2 = 10;
 System.out.println(num1 + num2);

 }
}
```

- Notice that even though the code above would be considered something around O(3), the number 3 is irrelevant because it doesn't make much of a difference.
- Also, notice that two values are being stored in two variables. Even though this is actually O(2) as space complexity, we can consider it as O(1) as well.
- If we have a code with many operations and we are not using any kind of external parameter that will change the time complexity, it will still be considered as O(1).

Accessing an Array by Index

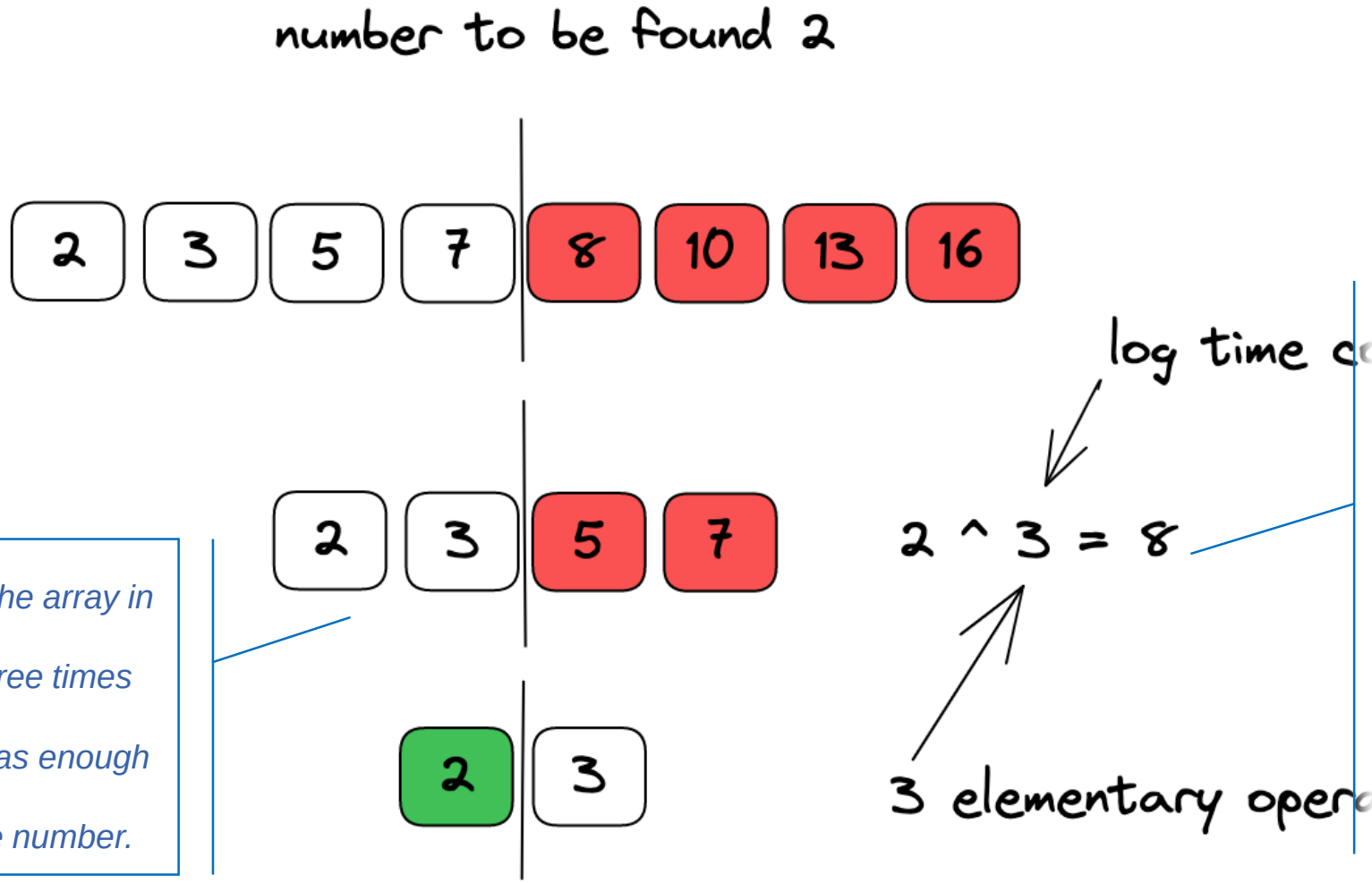
- Accessing an index of an array has also constant time complexity.
- Once the array is created in memory, we won't need to traverse the array, or do any other special operation.
- We just need to access it and the time complexity for that is O(1) or constant time:

```
int[] numbers = { 1, 2, 3};
System.out.println(numbers[1]); // O(1) here
```

Logarithmic – O(log n)

- The classic algorithm that uses the O(log n) complexity is **the binary search.**
- It's not necessary to run through the whole array. Instead, we get the number in the middle and check if it's lower or greater than the number to be found.
- Then we break the array in two, we break it again until the number is found. That's only possible because a binary search must have the array already sorted.

Logarithmic – O(log n) Example



We break the array in two for three times and that was enough to find the number.

*That's because 2 ^ 3 is the same as the size of our array. In other words, 2 * 2 * 2 = 8. Therefore, 3 is the log complexity of our number.*

Logarithmic – O(log n) Binary Search Example

```java
public class BinarySearch {
   public static int binarySearch(int[] array, int target) {
      int middle = array.length / 2;
      var leftPointer = 0;
      var rightPointer = array.length - 1;

      while (leftPointer <= rightPointer) {
         if (array[middle] < target) {
            leftPointer = middle + 1;
         } else if (array[middle] > target) {
             rightPointer = middle - 1;
         } else {
            return middle;
         }
         middle = (leftPointer + rightPointer) / 2;
      }
      return -1;
   }

   @Test
   public void testCaseLog3() {
      int[] array = { 2, 3, 5, 7, 8, 10, 13, 16 };
      System.out.println(binarySearch(array, 2));
   }

   @Test
   public void testCaseLog20() {
      int[] array = new int[1048576];
      int number = 0;
      for (int i = 0; i < array.length; i++) {
         array[i] = ++number;
      }
      System.out.println(binarySearch(array, 1));
   }
}
```

```
testCaseLog3:

Iterations count: 3

0
```

```
testCaseLog20:

Iterations count: 20

0
```

Linear – O(n)

- When we use looping we have a linear complexity. It doesn't matter if it's the ***for***, ***while***, ***for-each***, ***do-while***. Any of those loopings will have a linear complexity.
- If we have two loopings that depend on the same input, in our case, we will still have the time complexity of **O(n).**

Linear – O(n) Example

```
public void printNumbers(int n) {
  for (int i = 0; i < n; i++) {
    System.out.println(i);
  }
}
```

*The code will print the number **i** that is incremented in each iteration. Since we run through the looping n times, we will have the time complexity of O(n).*

Linear – O(n) Example

```
public static void printNumbers2Looping(int n) {
  for (int i = 0; i < n; i++) { // O(n)
    System.out.println(i);
  }

  for (int j = 0; j < n; j++) { // O(n)
    System.out.println(j);
  }
}
```

At first we might think that the time complexity is more than O(n) but it's actually still O(n) because both loopings are using n as the size number.

Two Inputs – O(m + n)

- To measure the time complexity we have to pay attention to the input numbers the algorithm is using. That will change the time complexity.
- If a looping depends on two inputs to be executed, the time complexity won't be **O(n),** instead, it will be **O(m + n)** if we use two variables as the looping length.
- **Example:**

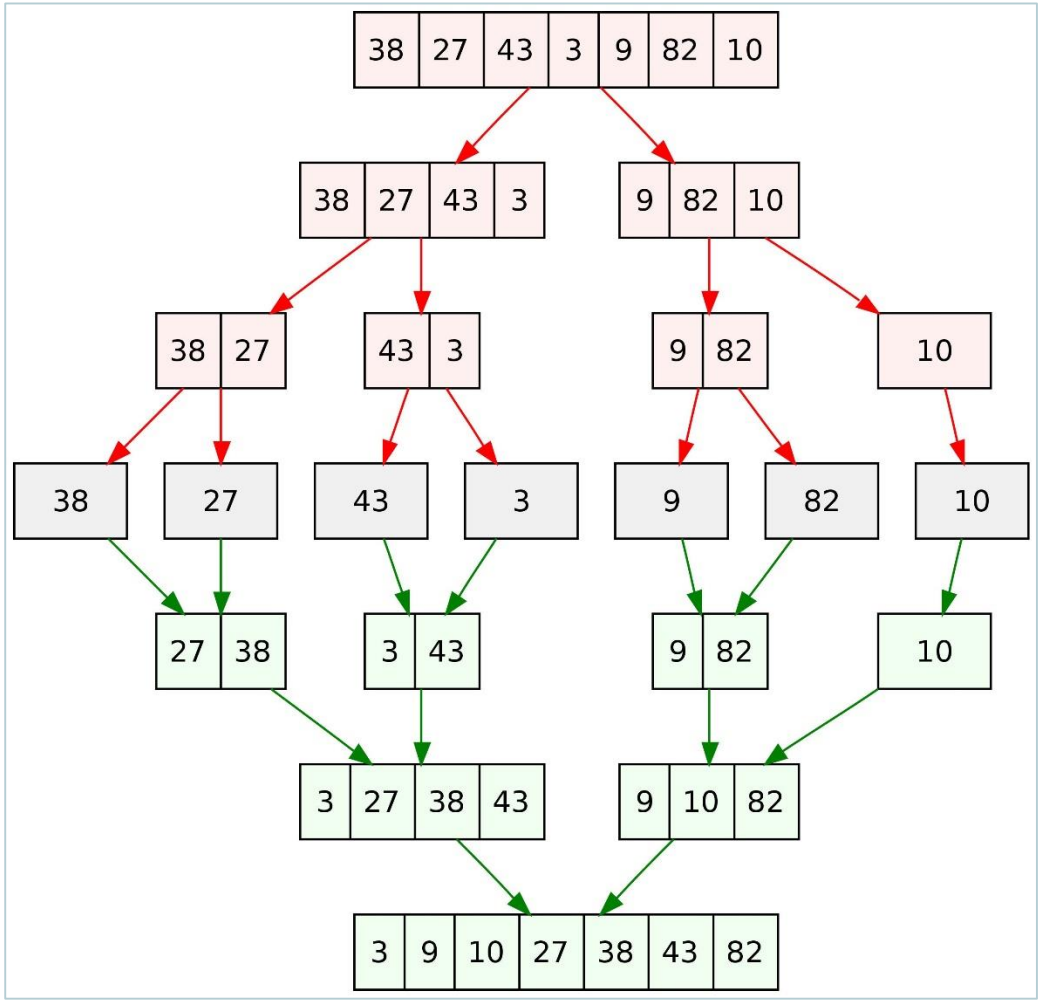
```
public static void printNumbers(int m, int n) {
  for (int i = 0; i < m; i++) {
    System.out.println(i);
  }

  for (int i = 0; i < n; i++) {
    System.out.println(i);
  }
}
```

*It doesn't matter if the size of n is greater than m or vice-versa, the Big(O) notation will **always** be O(m + n) because m or n might be any number. Therefore, it might dramatically impact the time complexity of either m or n.*

Log-linear – O(n log n)

- The time complexity O(n log n) is a bit slower than O(log n) and O(n).
- The log-linear complexity is present in algorithms that use the divide-and-conquer strategy.
- The time complexity of O(n log n) is scalable, it can handle a great number of elements effectively.



Quadratic – O(n²)

```
public class On2Example {
  public static void main(String[] args) {
    int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int countOperations = countOperationsOn2(numbers);
    System.out.println(countOperations);
  }

  public static int countOperationsOn2(int[] numbers) {
    int countOperations = 0;
    for (int i = 0; i < numbers.length; i++) {
      for (int j = 0; j < numbers.length; j++) {
        countOperations++;
      }
    }
    return countOperations;
  }
}
```

Output:
100

*As you can see in the code above, the iteration depends on the size of the array. We also have a for loop inside another one which makes the time complexity be multiplied by 10 * 10.*

*Notice that the size of the array we are passing is 10. Therefore, the **countOperations** variable will have a value of 100.*

Cubic – O(n ^ 3)

- The cubic complexity is similar to the quadratic one but instead of having two nested loopings, we will have 3 nested loopings.

```
public static int countOperationsOn3(int[] numbers) {
  int countOperations = 0;
  for (int i = 0; i < numbers.length; i++) {
    for (int j = 0; j < numbers.length; j++) {
      for (int k = 0; k < numbers.length; k++) {
        countOperations++;
      }
    }
  }
  return countOperations;
}
```

Notice that the cubic complexity will happen when we use 3 nested loopings.

This time complexity is very slow.

Factorial O(n!)

- The concept of factorial is simple. Suppose we have the factorial of ***5!*** then this will equal *1 * 2 * 3 * 4 * 5* which results in *120*.
- A good example of an algorithm that has factorial time complexity is the array permutation (sự hoán vị). In this algorithm, we need to check how many permutations are possible given the array elements. For example, if we have 3 elements A, B, and C, there will be 6 permutations. Let's see how it works:

> ABC, BAC, CAB, BCA, CAB, CBA – Notice that we have 6 possible permutations, the same as 3!.

- If we have 4 elements, we will have 4! which is the same as 24 permutations, if 5! 120 permutations, and so on.

Lesson Summary

- A Quick Review of Array in Java
- Arrays in Memory
- Big-O Notation
- Big-O Values for Array Operations

THANK YOU!