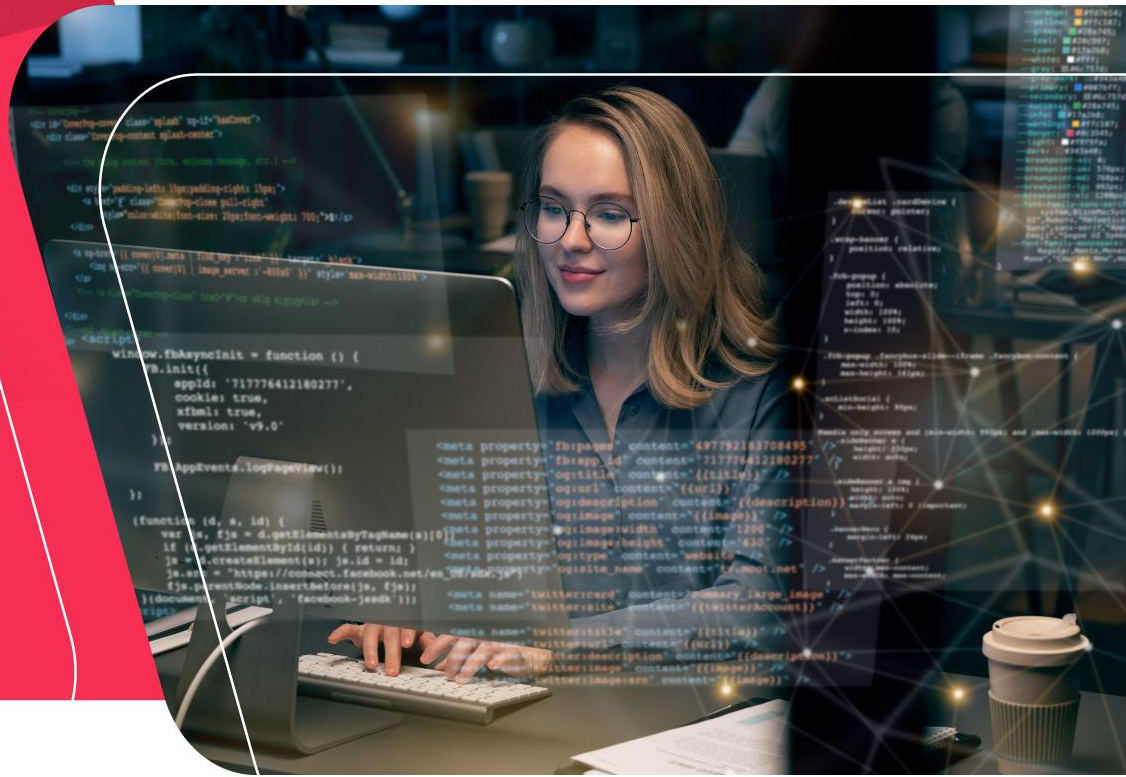
TIA-Academy

# Java Bootcamp

## Day 08



# Technologies will be Use

- JDK 8/**11**/15
- JRE 8/**11**/15
- Writing Code using Notepad++ (For Windows) or Vim (For Linux and Mac)
- Compiling with Command Prompt (For Windows) or Terminal (For Linux and Mac)
- ***Forbidden of using IDE Based Development Tools as of Intellij IDEA or Eclipse***

TIA-Academy

# Java Arrays

# Java Arrays

- An array is a **collection of similar types of data**.
- For example, if we want to store the names of 100 people then we can create an array of the string type that can store 100 names.

```
String[] array = new String[100];
```

- Here, the above array cannot store more than 100 names. The number of **values in a Java array is always fixed**

# How to declare an array in Java?

- In Java, here is how we can declare an array.

```
dataType[] arrayName;
```

- dataType - it can be **primitive data types like int, char, double, byte**, etc. or **Java objects**
- arrayName - it is an identifier
- For example,

```
double[] data;
```

- Here, **data** is an **array that can hold values of type double**.

# How to declare an array in Java?

- **But, how many elements can array this hold?**
- Good question! To **define the number of elements that an array can hold**, we have to allocate memory for the array in Java. For example,

```
// declare an array
double[] data;

// allocate memory
data = new double[10];
```

- Here, the array **can store 10 elements**. We can also say that the **size or length of the array is 10**.
- In Java, we can declare and allocate the memory of an array in one single statement. For example,

```
double[] data = new double[10];
```

# How to Initialize Arrays in Java?

- In Java, we can initialize arrays during declaration. For example,

```
//declare and initialize and array
int[] age = {12, 4, 5, 2, 5};
```

- Here, we have **created an array named age** and **initialized it with the values inside the curly brackets**.
- Note that **we have not provided the size of the array**. In this case, the Java compiler automatically specifies the size by **counting the number of elements in the array** (i.e. 5).

# How to Initialize Arrays in Java?

- In the Java array, **each memory location is associated with a number**. The **number is known as an array index**. We can also initialize arrays in Java, using the index number. For example,

```
// declare an array
int[] age = new int[5];

// initialize array
age[0] = 12;
age[1] = 4;
age[2] = 5;
..
```

```
//declare and initialize and array
int[] age = {12, 4, 5, 2, 5};
```


Java Arrays initialization

- **Array indices always start from 0**. That is, **the first element of an array is at index 0**.
- If the size of an array is **n**, then the last element of the array will be at **index n-1**.

# How to Access Elements of an Array in Java?

- We can access the element of an array using the index number. Here is the syntax for accessing elements of an array,

```
// access array elements
array[index]
```

- Let's see an example of accessing array elements using index numbers.

# How to Access Elements of an Array in Java?

- Example: Access Array Elements
- Output :

```
Accessing Elements of Array:
First Element: 12
Second Element: 4
Third Element: 5
Fourth Element: 2
Fifth Element: 5
```

```
class Main {
 public static void main(String[] args) {

   // create an array
   int[] age = {12, 4, 5, 2, 5};

   // access each array elements
   System.out.println("Accessing Elements of Array:");
   System.out.println("First Element: " + age[0]);
   System.out.println("Second Element: " + age[1]);
   System.out.println("Third Element: " + age[2]);
   System.out.println("Fourth Element: " + age[3]);
   System.out.println("Fifth Element: " + age[4]);
 }
}
```

# How to Access Elements of an Array in Java?

- The syntax of the if...else statement is:

```
if (condition) {
  // codes in if block
}
else {
  // codes in else block
}
```

- Here, the program will do one task (codes inside if block) if the condition is true and another task (codes inside else block) if the condition is false.

# Looping Through Array Elements

- In Java, we can also loop through each element of the array. For example,
- Example: Using For Loop
- Output :

```
Using for Loop:
12
4
5
```

```
class Main {
 public static void main(String[] args) {

   // create an array
   int[] age = {12, 4, 5};

   // loop through the array
   // using for loop
   System.out.println("Using for Loop:");
   for(int i = 0; i < age.length; i++) {
     System.out.println(age[i]);
   }
 }
}
```

# Looping Through Array Elements

- In the above example, we are using the for Loop in Java to iterate through each element of the array. Notice the expression inside the loop,

```
age.length
```

- Here, we are using the **length property of the array to get the size of the array**.

# Looping Through Array Elements

- We can also use the for-each loop to iterate through the elements of an array. For example,
- Example: Using the for-each Loop
- Output :

```
Using for Loop:
12
4
5
```

```
class Main {
 public static void main(String[] args) {

   // create an array
   int[] age = {12, 4, 5};

   // loop through the array
   // using for loop
   System.out.println("Using for-each Loop:");
   for(int a : age) {
     System.out.println(a);
   }
 }
}
```

# Looping Through Array Elements

- Example: Compute Sum and Average of Array Elements
- Output :

```
Sum = 36
Average = 3.6
```

```
class Main {
 public static void main(String[] args) {

   int[] numbers = {2, -9, 0, 5, 12, -25, 22, 9, 8, 12};
   int sum = 0;
   Double average;

   // access all elements using for each loop
   // add each element in sum
   for (int number: numbers) {
     sum += number;
   }

   // get the total number of elements
   int arrayLength = numbers.length;

   // calculate the average
   // convert the average from int to double
   average =  ((double)sum / (double)arrayLength);

   System.out.println("Sum = " + sum);
   System.out.println("Average = " + average);
 }
}
```

# Java if...else (if-then-else) Statement

- In the above example, we have created an array of named numbers. We have used the for...each loop to access each element of the array.
- Inside the loop, we are calculating the sum of each element. Notice the line,

```
int arrayLength = number.length;
```

- Here, we are using the length attribute of the array to calculate the size of the array. We then calculate the average using:

```
average = ((double)sum / (double)arrayLength);
```

- As you can see, we are converting the int value into double. This is called type casting in Java. To learn more about typecasting, visit Java Type Casting.

TIA-Academy

# ASSIGNMENT 01

TIA-Academy

# Java Multidimensional Arrays

# Java Multidimensional Arrays

- In this tutorial, we will learn about the Java multidimensional array using 2-dimensional arrays and 3-dimensional arrays with the help of examples.
- A multidimensional array is an array of arrays. Each element of a multidimensional array is an array itself. For example,

```
int[][] a = new int[3][4];
```

- Here, we have created a multidimensional array named a. It is a **2-dimensional array**, that can hold a **maximum of 12 elements**,

# Java Multidimensional Arrays

| | Column 1 | Column 2 | Column 3 | Column 4 |
|---|---|---|---|---|
| Row 1 | a[0][0] | a[0][1] | a[0][2] | a[0][3] |
| Row 2 | a[1][0] | a[1][1] | a[1][2] | a[1][3] |
| Row 3 | a[2][0] | a[2][1] | a[2][2] | a[2][3] |

2-dimensional Array

# Java Multidimensional Arrays

- Remember, Java uses **zero-based indexing**, that is, indexing of arrays in Java **starts with 0** and not 1.
- Let's take another example of the multidimensional array. This time we will be **creating a 3-dimensional array**. For example,

```
String[][][] data = new String[3][4][2];
```

- Here, data is a **3d array that can hold a maximum of 24 (3*4*2)** elements of type **String**.

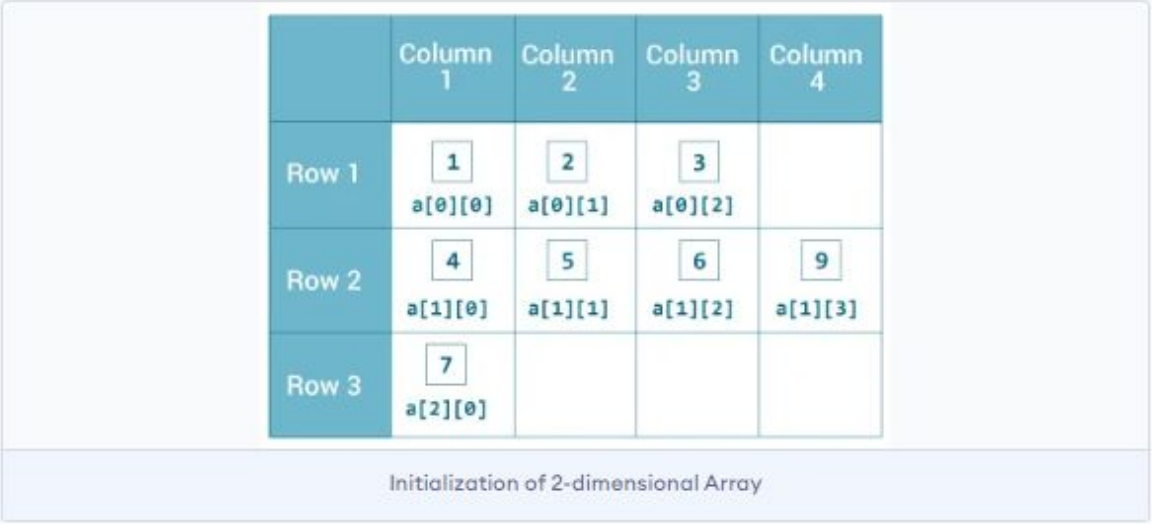
# How to initialize a 2d array in Java?

- Here is how we can initialize **a 2-dimensional array** in Java.

```
int[][] a = {
        {1, 2, 3},
        {4, 5, 6, 9},
        {7},
};
```

- As we can see, each element of the multidimensional array is an array itself. And also, unlike C/C++**, each row of the multidimensional array in Java can be of different lengths.**

# How to initialize a 2d array in Java?



| | Column 1 | Column 2 | Column 3 | Column 4 |
|---|---|---|---|---|
| Row 1 | 1 a[0][0] | 2 a[0][1] | 3 a[0][2] | |
| Row 2 | 4 a[1][0] | 5 a[1][1] | 6 a[1][2] | 9 a[1][3] |
| Row 3 | 7 a[2][0] | | | |

Initialization of 2-dimensional Array

# How to initialize a 2d array in Java?

- Example: 2-dimensional Array
- Output :

```
Length of row 1: 3
Length of row 2: 4
Length of row 3: 1
```

```
class MultidimensionalArray {
    public static void main(String[] args) {

        // create a 2d array
        int[][] a = {
            {1, 2, 3},
            {4, 5, 6, 9},
            {7},
        };

        // calculate the length of each row
        System.out.println("Length of row 1: " + a[0].length);
        System.out.println("Length of row 2: " + a[1].length);
        System.out.println("Length of row 3: " + a[2].length);
    }
}
```

# How to initialize a 2d array in Java?

- In the above example, we are creating a multidimensional array named a. Since each component of a multidimensional array is also an array (a[0], a[1] and a[2] are also arrays).
- Here, we are using the length attribute to calculate the length of each row.

# How to initialize a 2d array in Java?

- Example: Print all elements of 2d array Using Loop
- Output :

```
a[0][0] : 1
a[0][1] : -2
a[0][2] : 3

a[1][0] : -4
a[1][1] : -5
a[1][2] : 6
a[1][3] : 9

a[2][0] : 7
```

```
class MultidimensionalArray {
    public static void main(String[] args) {

        int[][] a = {
            {1, -2, 3},
            {-4, -5, 6, 9},
            {7},
        };

        for (int i = 0; i < a.length; ++i) {
            for(int j = 0; j < a[i].length; ++j) {
                System.out.println("a["+i+"]["+j+"] : "+a[i][j]);
            }
          System.out.println();
        }
    }
}
```

# How to initialize a 2d array in Java?

- We can also use the for...each loop to access elements of the multidimensional array. For example,
- Example: Print all elements of 2d array Using Loop
- Output :

```
1
-2
3
-4
-5
6
9
7
```

```
class MultidimensionalArray {
    public static void main(String[] args) {

        int[][] a = {
            {1, -2, 3},
            {-4, -5, 6, 9},
            {7},
        };

        for (int i = 0; i < a.length; ++i) {
            for(int j = 0; j < a[i].length; ++j) {
                System.out.println("a["+i+"]["+j+"] : "+a[i][j]);
            }
          System.out.println();
        }
    }
}
```

# How to initialize a 2d array in Java?

- In the above example, we are have created a 2d array named a. We then used for loop and for...each loop to access each element of the array.

# How to initialize a 3d array in Java?

- Let's see how we can use a 3d array in Java. We can initialize a 3d array similar to the 2d array. For example,

```
// test is a 3d array
int[][][] test = {
        {
          {1, -2, 3},
          {2, 3, 4}
        },
        {
          {-4, -5, 6, 9},
          {1},
          {2, 3}
        }
};
```

- Basically, a 3d array is an array of 2d arrays. The rows of a 3d array can also vary in length just like in a 2d array.

# How to initialize a 3d array in Java?

- Example: 3-dimensional Array
- Output :

```
1
-2
3
2
3
4
-4
-5
6
9
1
2
3
```

```
class ThreeArray {
    public static void main(String[] args) {

        // create a 3d array
        int[][][] test = {
            {
              {1, -2, 3},
              {2, 3, 4}
            },
            {
              {-4, -5, 6, 9},
              {1},
              {2, 3}
            }
        };

        // for..each loop to iterate through elements of 3d array
        for (int[][] array2D: test) {
            for (int[] array1D: array2D) {
                for(int item: array1D) {
                    System.out.println(item);
                }
            }
        }
    }
}
```

TIA-Academy

# ASSIGNMENT 02

TIA-Academy

# Java Copy Arrays

# Copying Arrays Using Assignment Operator

- Let's take an example,
- Output :

```
1, 2, 3, 4, 5, 6
```

- In the above example, we have used the assignment operator (=) to copy an array named numbers to another array named positiveNumbers.

```
class Main {
    public static void main(String[] args) {

        int [] numbers = {1, 2, 3, 4, 5, 6};
        int [] positiveNumbers = numbers;    // copying arrays

        for (int number: positiveNumbers) {
            System.out.print(number + ", ");
        }
    }
}
```

# Copying Arrays Using Assignment Operator

- This technique is the easiest one and it works as well.
- However, there is a problem with this technique. If we change elements of one array, corresponding elements of the other arrays also change. For example,
- Output :

```
1, 2, 3, 4, 5, 6
```

```
class Main {
    public static void main(String[] args) {

        int [] numbers = {1, 2, 3, 4, 5, 6};
        int [] positiveNumbers = numbers;    // copying arrays

        // change value of first array
        numbers[0] = -1;

        // printing the second array
        for (int number: positiveNumbers) {
            System.out.print(number + ", ");
        }
    }
}
```

# Copying Arrays Using Assignment Operator

- Here, we can see that **we have changed one value of the numbers array**. **When we print the positiveNumbers array**, we can see that **the same value is also changed**.
- It's because **both arrays refer to the same array object**. This is because of the shallow copy.
- Now, to make **new array objects** while copying the arrays, we need **deep copy** rather than a shallow copy.

# Using Looping Construct to Copy Arrays

- Let's take an example:
- Output :

```
[1, 2, 3, 4, 5, 6]
```

```
class Main {
    public static void main(String[] args) {

        int [] numbers = {1, 2, 3, 4, 5, 6};
        int [] positiveNumbers = numbers;    // copying arrays

        // change value of first array
        numbers[0] = -1;

        // printing the second array
        for (int number: positiveNumbers) {
            System.out.print(number + ", ");
        }
    }
}
```

# Using Looping Construct to Copy Arrays

- In the above example, we have used the for loop to iterate through each element of the source array. In each iteration, we are copying elements from the source array to the destination array.
- Here, the source and destination array refer to different objects (deep copy). Hence, if elements of one array are changed, corresponding elements of another array is unchanged.
- Notice the statement,

```
System.out.println(Arrays.toString(destination));
```

- Here, the toString() method is used to convert an array into a string.

# Copying Arrays Using arraycopy() method

- In Java, the System class contains a method named arraycopy() to copy arrays. This method is a better approach to copy arrays than the above two.
- The arraycopy() method allows you to copy a specified portion of the source array to the destination array. For example,

```
arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

- Here,
  1. src - source array you want to copy
  2. srcPos - starting position (index) in the source array
  3. dest - destination array where elements will be copied from the source
  4. destPos - starting position (index) in the destination array
  5. length - number of elements to copy

# Copying Arrays Using arraycopy() method

- Let's take an example:
- Output

```
n2 = [2, 3, 12, 4, 12, -2]
n3 = [0, 12, 4, 0, 0]
```

```
// To use Arrays.toString() method
import java.util.Arrays;

class Main {
    public static void main(String[] args) {
        int[] n1 = {2, 3, 12, 4, 12, -2};

        int[] n3 = new int[5];

        // Creating n2 array of having length of n1 array
        int[] n2 = new int[n1.length];

        // copying entire n1 array to n2
        System.arraycopy(n1, 0, n2, 0, n1.length);
        System.out.println("n2 = " + Arrays.toString(n2));

        // copying elements from index 2 on n1 array
        // copying element to index 1 of n3 array
        // 2 elements will be copied
        System.arraycopy(n1, 2, n3, 1, 2);
        System.out.println("n3 = " + Arrays.toString(n3));
    }
}
```

# Copying Arrays Using arraycopy() method

- In the above example, we have used the arraycopy() method,
  - System.arraycopy(n1, 0, n2, 0, n1.length) - entire elements from the n1 array are copied to n2 array
  - System.arraycopy(n1, 2, n3, 1, 2) - 2 elements of the n1 array starting from index 2 are copied to the index starting from 1 of the n3 array
- As you can see, the default initial value of elements of an int type array is 0.

# Copying Arrays Using copyOfRange() method

- We can also use the copyOfRange() method defined in Java Arrays class to copy arrays. For example,
- Output

```
destination1 = [2, 3, 12, 4, 12, -2]
destination2 = [12, 4, 12]
```

```
// To use toString() and copyOfRange() method
import java.util.Arrays;

class ArraysCopy {
    public static void main(String[] args) {

        int[] source = {2, 3, 12, 4, 12, -2};

        // copying entire source array to destination
        int[] destination1 = Arrays.copyOfRange(source, 0, source.length);
        System.out.println("destination1 = " + Arrays.toString(destination1));

        // copying from index 2 to 5 (5 is not included)
        int[] destination2 = Arrays.copyOfRange(source, 2, 5);
        System.out.println("destination2 = " + Arrays.toString(destination2));
    }
}
```

# Copying Arrays Using copyOfRange() method

- In the above example, notice the line,

```
int[] destination1 = Arrays.copyOfRange(source, 0, source.length);
```

- Here, we can see that we are creating the destination1 array and copying the source array to it at the same time. We are not creating the destination1 array before calling the copyOfRange() method.

# Copying 2d Arrays Using Loop

- Similar to the single-dimensional array, we can also copy the 2-dimensional array using the for loop. For example,
- Output

```
[[1, 2, 3, 4], [5, 6], [0, 2, 42, -4, 5]]
```

```
import java.util.Arrays;

class Main {
    public static void main(String[] args) {

        int[][] source = {
              {1, 2, 3, 4},
              {5, 6},
              {0, 2, 42, -4, 5}
              };

        int[][] destination = new int[source.length][];

        for (int i = 0; i < destination.length; ++i) {

            // allocating space for each row of destination array
            destination[i] = new int[source[i].length];

            for (int j = 0; j < destination[i].length; ++j) {
                destination[i][j] = source[i][j];
            }
        }

        // displaying destination array
        System.out.println(Arrays.deepToString(destination));

    }
}
```

# Copying 2d Arrays Using Loop

- In the above program, notice the line,

```
System.out.println(Arrays.deepToString(destination);
```

- Here, the deepToString() method is used to provide a better representation of the 2-dimensional array. To learn more, visit Java deepToString().

# Copying 2d Arrays using arraycopy()

- Similar to the single-dimensional array, we can also copy the 2-dimensional array using the for loop. For example,
- Output

```
[[1, 2, 3, 4], [5, 6], [0, 2, 42, -4, 5]]
```

```
import java.util.Arrays;

class Main {
    public static void main(String[] args) {

        int[][] source = {
              {1, 2, 3, 4},
              {5, 6},
              {0, 2, 42, -4, 5}
              };

        int[][] destination = new int[source.length][];

        for (int i = 0; i < source.length; ++i) {

             // allocating space for each row of destination array
             destination[i] = new int[source[i].length];
             System.arraycopy(source[i], 0, destination[i], 0, destination[i].length);
        }

        // displaying destination array
        System.out.println(Arrays.deepToString(destination));
    }
}
```

# Copying 2d Arrays using arraycopy()

- Here, we can see that we get the same output by replacing the inner for loop with the arraycopy() method.

TIA-Academy

# Searching and Sorting

# Searching and Sorting

- TWO ARRAY PROCESSING TECHNIQUES that are particularly common are searching and sorting.
- Searching here refers to finding an item in the array that meets some specified criterion.
- Sorting refers to rearranging all the items in the array into increasing or decreasing order (where the meaning of increasing and decreasing can depend on the context).
- An array is not just used to store elements but also access those stored values later on.
- We may also need to search for a particular element in the array. Sorting the array may also be required at times.
- We will now look into how these things are done using different algorithms. And we shall also see what the big O notation is and use it to determine the efficiency of these algorithms.

# Simple Search or Linear Search

- The simplest way to search for an element is to iterate through the entire array and compare each value of that array with the target element.
- When a match is found, we may break out of the loop if we are sure that there exist no duplicate values.
- However, if duplicate values do exist in the array, we might have to continue searching even when a match is found. Since we search for the elements in a linear order from left to right ( by convention, you may also search from the end of the array to the beginning), this algorithm is named as linear search.

# Simple Search or Linear Search

```
int[] a = { 3, 34, 5, 91, 100}; // an array not containing duplicates
int target = 91; // the element to be searched
for( int i=0; i<a.length; i++) {
    if(a[i]==target) {
        System.out.println ( "Element found at index "+i);
        break; // break should be omitted if the array contains duplicates
    }
}
```

# Simple Search or Linear Search

- Before we move onto the next algorithm, we will look at what efficiency means in the context of algorithms.
- There are two things that have to be taken care of when we write algorithms. **The first is the execution time** and **the second is the memory requirement**.
- Execution time is determined by the numbers of **statements that are excited** by the algorithm while memory requirement is determined by the **additional variables that we use.**

# Simple Search or Linear Search

- In the above program, we have used only one variable, target, which falls under memory requirement.
- However, the execution time cannot be specified directly as such even if we consider every statement to take the same time to execute.
- The reason is that, in this particular algorithm, if the target is in the beginning of the array, then this search would require only a single comparison which is known as the best case.

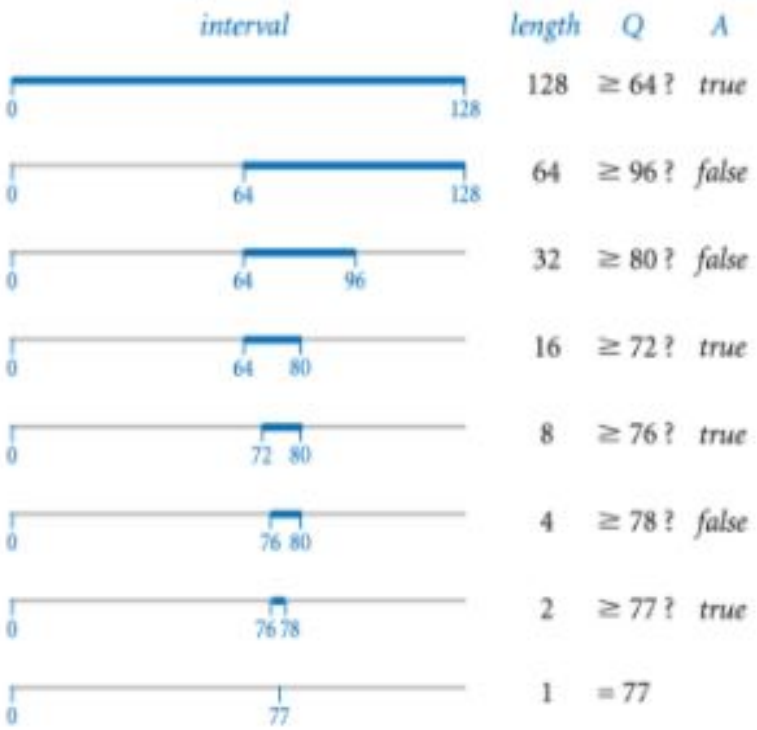
# Simple Search or Linear Search

- **However if the target is located at the end of the array,** the number of comparisons required would be equal to the length of the array. This is the worst case.
- The average execution time would occur when the target is located in the middle of the array. So, one thing that we can conclude is that the efficiency of algorithms depends on the input data too.
- Here arises the need for a standardised comparison of efficiencies. And one solution is the Big O notation.

# Simple Search or Linear Search

- **The big O notation is gives us a relation between the number of data items contained in the array and the number of comparisons required**. It takes the worst case into account.
- It determines how hard an algorithm has to work to obtain the result. In this particular linear search algorithm, if the number of data items are n, then the number of compressions required are also n. This is the order of the algorithm. Hence, linear search is said to be an algorithm of order n.

# Binary Search




- Binary search is an efficient algorithm which can be used to search in a sorted array. Note that the array has to be sorted in either ascending or descending order for the algorithm to worth.
- Initially, the range of the array to be searched begins at the first element of the array and extends up to the last element.
- This range reduces by half in every iteration. We locate the middle element of the array and compare it with the target. If the target equals the middle element, our search is completed, otherwise the range has to be adjusted accordingly.

# Binary Search

- For now, assume that the array is sorted in ascending order. If the middle element is smaller than the target, then the target cannot be found in the left half of the array as each of those elements would also be smaller than the target.

```
[0,1,2, … 63,64,65 …,126,127,128]
```

- Hence, we can narrow down our search to the right half of the array. On the other hand, if the middle element is larger than the target, we narrow our search to the left auld of the array. Clearly, the range of elements to be searched has been reduced by half.

# Binary Search

- We now perform the same operation of finding the middle element of the new range and reduce the range accordingly. In the second iteration, the range of elements to be searched reduces to one fourth of the original array length. Similarly, with the third iteration, the range of elements reduce to one eighth.

```
[64,65,66,… 95,96,97 …,126,127,128]
```

# Binary Search

- We now have to represent this algorithm programmatically. For this purpose, we take two variables left and right which represent the bounds of the array to be searched. Left indicates the left bound of the range and right indicates the right bound of the range.
- Initially, the left bound is set to 0 and the right bound is set to array length -1. We then use a while loop to perform the repetitive task of finding the middle element and comparing it with the target event.
- We shall look into the termination condition of while shortly. For now, the various things to be included in the while loop are statements to find the middle element, compare it with the target and set the left and right bounds accordingly.
- Now coming to the condition in the while loop, this process of finding the middle element and comparing it with the target would end when we are left with just a single element. This happens when the value of left and right becomes identical. On moving further, either right would become less than left or left would become more than right. At this point of time, the search needs to be stopped. Hence the loop condition is left<=right.

# Binary Search

```
int[] a = {3, 7, 10, 15, 91, 110, 150}; // a sorted array not containing duplicates
int target = 91; // the element to be searched
int left = 0;
int middle;
int right = a.length - 1;
while (left <= right) {
    middle = (left + right) / 2;
    if (a[middle] == target) {
        System.out.println("Element found at index " + middle);
        break;
    } else if (a[middle] < target) {
        left = middle + 1;
    } else if (a[middle] > target) {
        right = middle - 1;
    }
}
```

# Sorting

- Now we move to sorting of arrays. Sorting refers to arranging the elements of an array in either ascending or descending order. We will in the code snippets that follow arrange the elements in ascending order. The code may be suitably modified to sort in descending order. We will deal with two algorithms: Simple or selection sort and insertion or bubble sort.

- The selection sort algorithm is rather quiet simple. **We start from the left of the array and search for the smallest element and then place is at index 0 by swapping the two elements**. In order to find the smallest element, we first assume that the element at index zero is the smallest and assign the index 0 to a variable, say minPos.

- We then **compare each of the successive elements with the value held at minPos**. If the new value in smaller than what minPos corresponds to, the index stored in minPos is updated. In this way, when we reach the end of the array, the variable minPos will be holding the index of the smallest element. Now, we exchange the number stored at index 0 with the number stored at index minPos.

# Sorting

- After this first iteration, we bought the smallest element to index 0. Now, we need to search for the second smallest element.
- To do it, we assume that the element at index 1 is the smallest and proceed as before. In this way, we sort the entire array. Given below is the example for **selection sort.**

# Sorting

- After this first iteration, we bought the smallest element to index 0. Now, we need to search for the second smallest element.
- To do it, we assume that the element at index 1 is the smallest and proceed as before. In this way, we sort the entire array. Given below is the example for **selection sort.**

# Sorting

```
int[] a = {4, 85, 7, 1, 0, 36, -5, 48};
int minPos;

for (int i = 0; i < a.length; i++) {
            minPos = i;

    for (int j = i + 1; j < a.length; j++) {
         if (a[j] < a[minPos]){
              minPos = j;
         }
    }

    int temp = a[i];
    a[i] = a[minPos];
    a[minPos] = temp;

}
```

# Sorting

- Note that the above code uses two for loop, one nested within the other. The outer loop keeps track of the position from which the elements are to be searched for the smallest data item. Initially, this position is 0.

- After we bring the smallest item to index 0, this position in the second iteration becomes 1. The inner loop is used to find the smallest elements starting from the position defined by the outer loop. After finding the smallest element, the two values are swapped.

- This algorithm has an order of n2. The outer loop executes for n times. And the inner loop executes for variable number of times deepening on the value of i. When i is 0, the inner lope executes n-1 times. When i is 1, the inner loops excites n-2 times and so on. Therefore the total number of comparisons made come out to be (n-1)+(n-2)+(n-3)+....(n-(n-1)) which equals (n-1)*n-(1+2+3....n-1).

# Bubble Sort

- The last of the algorithms that we are gaining to deal with is the bubble sort method. This algorithm has its name derived from the water bubbles which surface to the top. In a similar way, in this algorithm, we make larger numbers move to the top of the array when we wish to sort in ascending order.

- We iterate through the array from the left to the right and compare each pair of successive values in turn. If they are in the right order, we leave them as they are. However, if a larger item is on the left of a smaller item, we swap them. Note that only successive values are compared. In this way, when we move to the rightmost end of the array, the largest value is guaranteed to have been bubbled to the top of the array.

# Bubble Sort

- For example, if the largest element was at index 0 initially, then in each step of comparison, this element is moved up ultimately bringing it to the top. In this way when we repeat this process for n-1 times, the entire array is guaranteed to be sorted.

- We can further improve this algorithm by setting proper loop termination conditions. This algorithm requires two nested loops. Within the inner loop, the loop condition can be stated to be a relation with the outer counter variable rather than the end of the array.

- For example, if it is the fifth iteration, then the four largest elements have already been bubbled to the top of the array. So, there is no need of comparing those values gain. So, the loop condition can be specified as j<x.length-1-i. Given below is the code for bubble sort.

# Bubble Sort

- The bubble sort algorithm also has an order of n2.

```
int[] a = {4, 85, 7, 1, 0, 36, -5, 48};
for (int i = 0; i < a.length - 1; i++) {
    for (int j = 0; j < a.length - 1 - i; j++) {
        System.out.println("a[j + 1] = "+ a[j + 1] )
        System.out.println("a[j] = "+ a[j] )
        if (a[j + 1] < a[j]) {
            int temp = a[j];
            a[j] = a[j + 1];
            a[j + 1] = temp;
        }
    }
}
```

TIA-Academy

# ASSIGNMENT 03
# (HOME ASSIGNMENT)

# Sources

- https://www.programiz.com/java-programming

TIA-Academy

# Thank You