Analysis of Algorithms

Tutorial #2

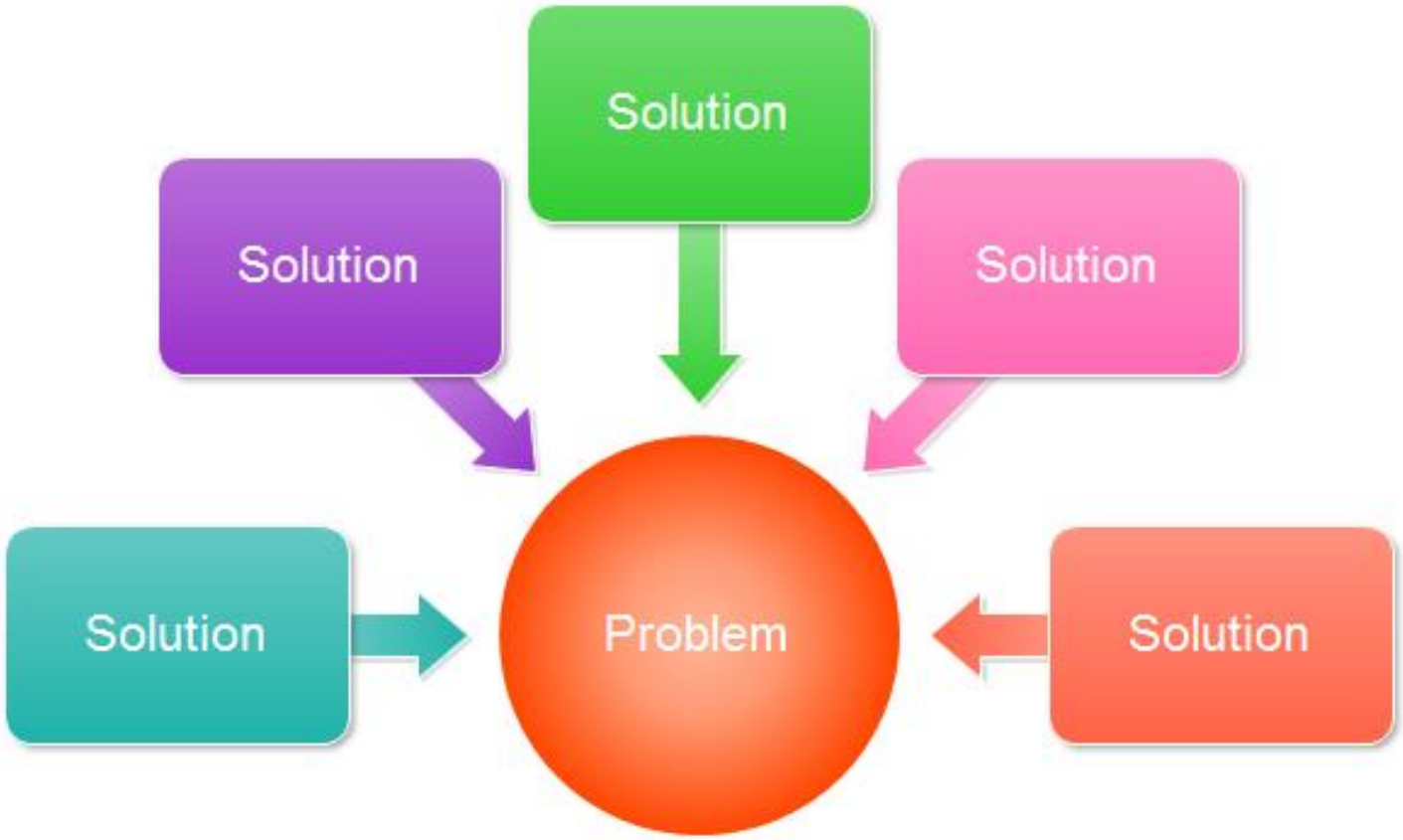
By

Mustafa Daraghmeh

Outlines

- **Complexity Analysis of Algorithms Review**
- **Calculate Time Complexity**
- **Calculate Space Complexity**

Complexity Analysis of Algorithms



What is the most efficient solution?

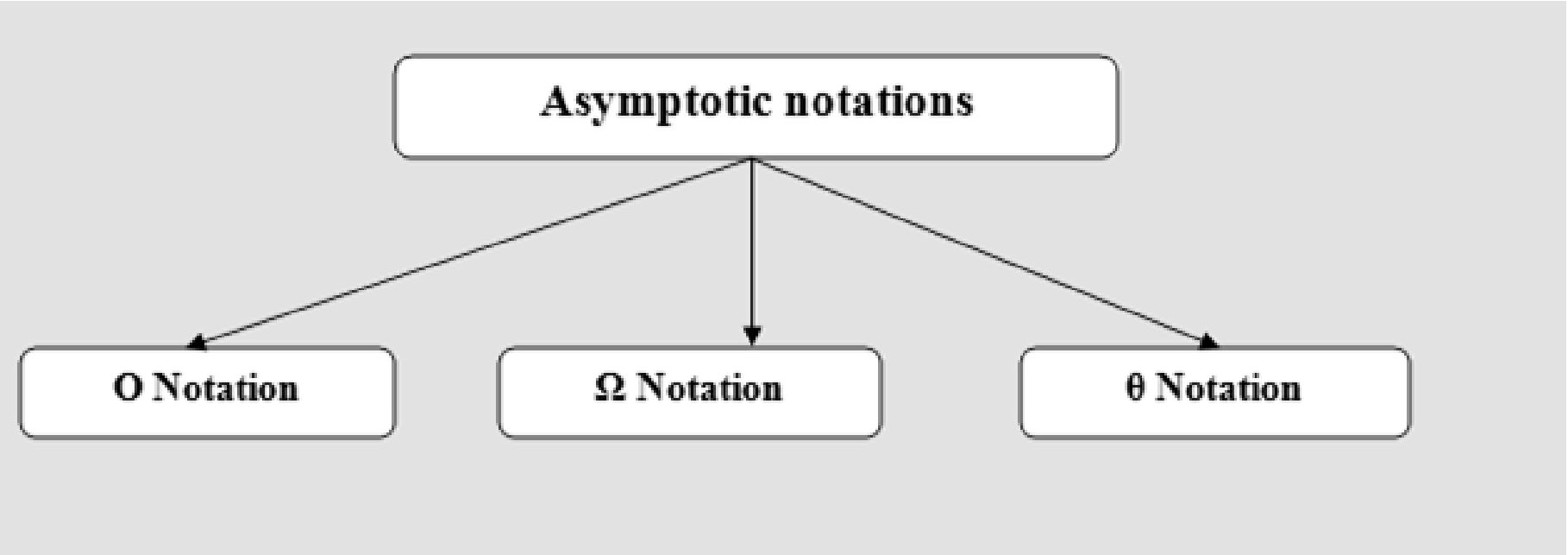
Complexity Analysis of Algorithms

- **Analysis of Algorithm**s is the determination of the amount of time, storage and/or other resources necessary to execute an Algorithm.

- Analyzing algorithms is called Asymptotic Analysis

- **Asymptotic Analysis** evaluate the performance of an algorithm.

Asymptotic Notations



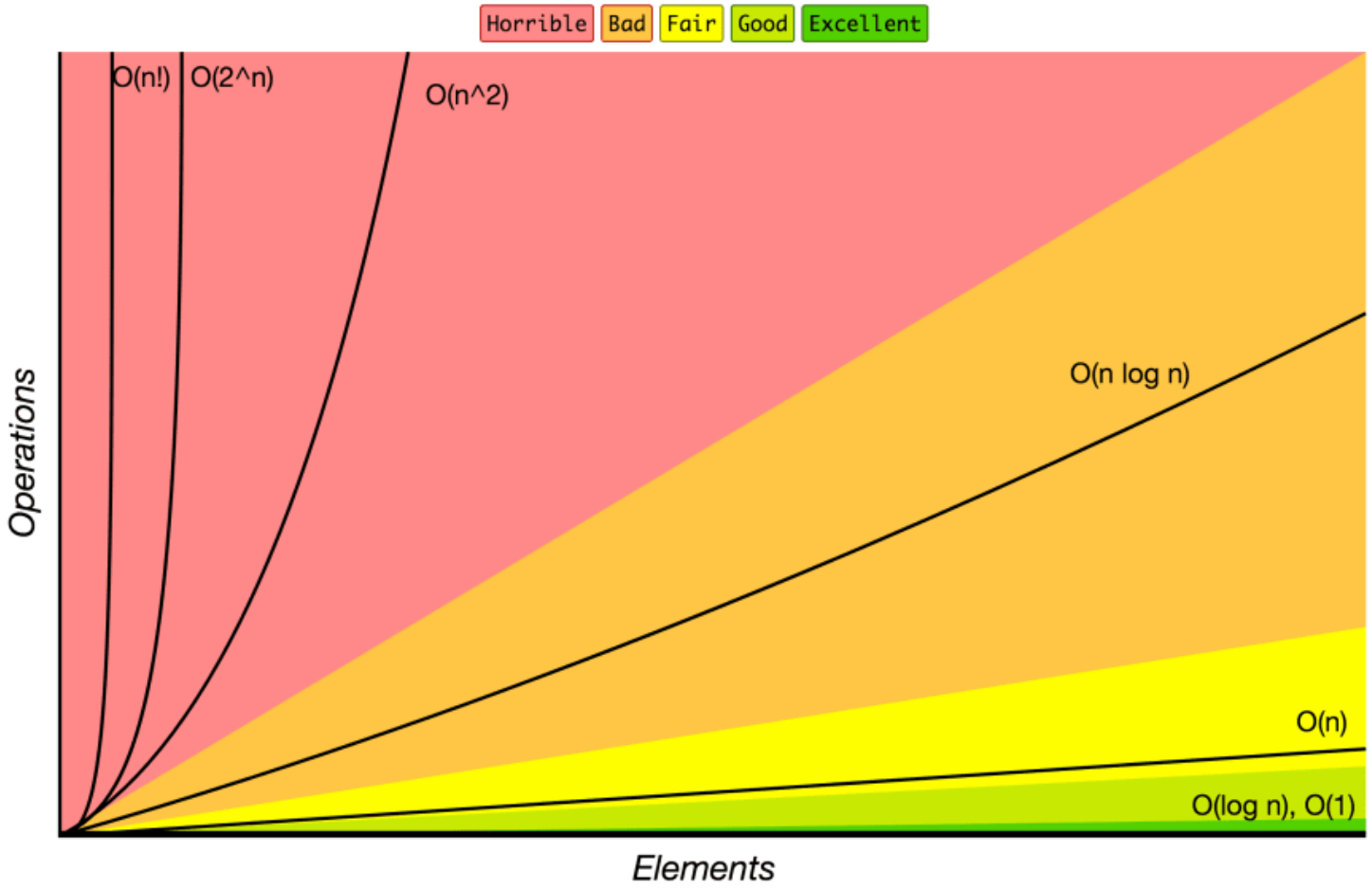
Time Complexity of Algorithms

- How does the computing time relate to the amount of input?
- Is it a linear relation?
- Does computing time rise exponentially for the doubling of input?

Time Complexity of Algorithms

- Time complexity of an algorithm quantifies the amount of time taken by an algorithm.

- **We can consider three cases:**
  1) Worst Case (Big O Notation)
  2) Average Case (Θ Notation)
  3) Best Case (Ω Notation)



Constant Time Complexity – O(1)

```java
public static void ConstantTimeFunction(int n){
    //Constant Complexity, then Constant Runtime
    System.out.println("**** Constant time ****");
    System.out.println("Your input is: " + n);
    System.out.println("Running time not dependent on input size!");
    System.out.println();
}
```

Note: A loop or recursion that runs a CONSTANT NUMBER OF TIMES is also considered as O(1)

Logarithmic Time Complexity – O(log n)

```java
public static void LogarithmicTimeFunction(int n) {
    //Logarithmic Complexity, then Logarithmic Runtime
    System.out.println("**** Logarithmic Time ****");
    int total=0;
    for (int i = 1; i < n; i = i * 2) {
        // Some O(1) expressions
        total++;
    }
    System.out.println("Loop 1, Total amount of times run: " + total);
    total=0;
    for (int i = n; i > 0; i = i / 2) {
        // Some O(1) expressions
        total++;
    }
    System.out.println("Loop 2, Total amount of times run: " + total);
    System.out.println();
}
```

Linear Time Complexity – O(n)

```java
public static void LinearTimeFunction(int n){
    //Linear Complexity, then Linear Runtime
    System.out.println("**** Linear Time ****");
    int total=0;
    for (int i = 0; i < n; i++) {
        // Some O(1) expressions
        total++;
    }
    System.out.println("Total amount of times run: " + total);
    System.out.println();
}
```

N Log N Time Complexity – O(n log n)

```java
public static void NLogNTimeFunction(int n) {
    // N Log N Complexity, then N Log N Runtime
    System.out.println("**** nlogn Time ****");
    int total = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j < n; j = j * 2) {
            // Some O(1) expressions
            total++;
        }
    }
    System.out.println("Total amount of times run: " + total);
    System.out.println();
}
```

Polynomial Time Complexity – $O(n^P)$

Quadratic Time Complexity - $O(n^2)$

```java
public static void QuadraticTimeFunction(int n) {
    // Quadratic Complexity, then Quadratic Runtime
    System.out.println("**** Quadratic Time ****");
    int total = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            // Some O(1) expressions
            total++;
        }
    }
    System.out.println("Total amount of times run: " + total);
    System.out.println();
}
```

Cubic Time Complexity - $O(n^3)$

```java
public static void CubicTimeFunction(int n) {
    // Cubic Complexity, then Cubic Runtime
    System.out.println("**** Cubic Time ****");
    int total = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            for (int k = 1; k <= n; k++) {
                // Some O(1) expressions
                total++;
            }
        }
    }
    System.out.println("Total amount of times run: " + total);
    System.out.println();
}
```

Exponential Time Complexity – $O(2^n)$

```java
public static void ExponentialTimeFunction(int n) {
    // Exponential Complexity, then Exponential Runtime
    System.out.println("**** Exponential Time ****");
    int total = 0;
    for (int i = 1; i <= Math.pow(2, n); i++) {
        // Some O(1) expressions
        total++;
    }
    System.out.println("Total amount of times run: " + total);
    System.out.println();
}
```

Factorial Time Complexity – O(n!)

```java
public static void FactorialTimeFunction(int n) {
    // Factorial Complexity, then Factorial Runtime
    System.out.println("**** Factorial Time ****");
    int total = 0;
    for (int i = 1; i <= factorial(n); i++) {
        // Some O(1) expressions
        total++;
    }
    System.out.println("Total amount of times run: " + total);
}

public static int factorial(int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial(   n - 1);
}
```

Time Complexity of Algorithms

- **Compute the elapsed time**

```
// TimeType.seconds, TimeType.milliSeconds, TimeType.nanoSeconds
StopWatch watch = new StopWatch(TimeType.nanoSeconds);
watch.start();
// some code
watch.stop();
System.out.println(watch.getTime());
```

```
long startMillis= System.currentTimeMillis();
{
    // some code
}
long endMillis= System.currentTimeMillis();
System.out.println("Elapsed Time in Millis is: "
        +(endMillis-startMillis));
```

```
long startNano=System.nanoTime();
{
    //some code
}
long endNano=System.nanoTime();
System.out.println("Elapsed Time in Nano is: "
        + (endNano-startNano));
```

Time Complexity Calculation – Problem 1

- Write the following function in java and find the time complexity:

```
function(int n)
{
    if (n==1)
       return;
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

Time Complexity Calculation – Problem 1

- **Solution:** Time Complexity O(n). Even though the inner loop is bounded by n, but due to break statement it is executing only once.

```
public static void Problem1(int n){
    if(n==1)
        return;
    /* Outer loop executes n times. */
    for(int i=1; i<=n; i++){
        /* Inner loop executes only one time due to break statement. */
        for(int j=1;j<=n;j++){
            System.out.print("*");
            break;
        }
    }
}
```

Time Complexity Calculation – Problem 2

- Write the following function in java and find the time complexity:

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)
        for (int j=1; j+n/2<=n; j = j++)
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Time Complexity Calculation – Problem 2

- **Solution:** Time $O(n^2 \log n)$

```
public static void Problem2(int n){
    int count =0;
    for(int i=n/2; i<=n; i++){ /* Outer loop executes n/2 times. */
        for(int j=1;(j+(n/2))<=n;j++){/* Middle loop executes n/2 times.*/
            for(int k=1;k<=n;k=k*2){ /* Inner loop executes log n times. */
                count++;
            }
        }
    }
    System.out.println("\n"+count);
}
```

Time Complexity Calculation – Problem 3

- Write the following function in java and find the time complexity:

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)
        for (int j=1; j<=n; j = 2 * j)
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Time Complexity Calculation – Problem 3

- **Solution:** Time $O(n \log^2 n)$

```java
public static void Problem3(int n){
    int count =0;
    for(int i=n/2; i<=n; i++){ /* Outer loop executes n/2 times. */
        for(int j=1;j<=n;j*=2){/* Middle loop executes log n times.*/
            for(int k=1;k<=n;k=k*2){ /* Inner loop executes log n times. */
                count++;
            }
        }
    }
    System.out.println("\n"+count);
}
```

Note: $\log^2(n) = (\log(n))^2 = (\log(n)) \times (\log(n))$

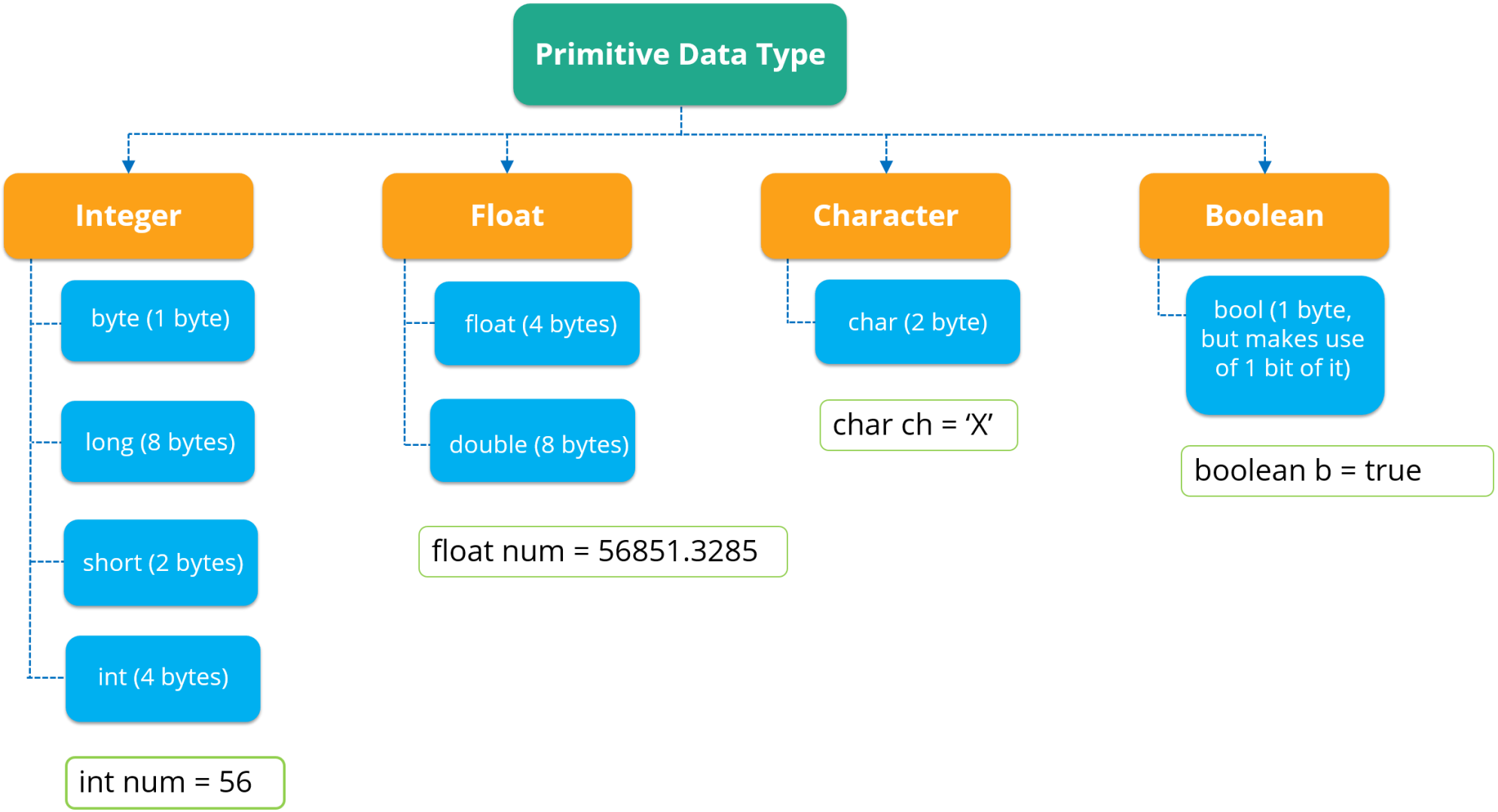
Space Complexity of Algorithms

- What is Space Complexity?
  - The total amount of memory space used by an algorithm including the space of input values for execution.
    - The lesser the space used, the faster it executes.

- Why do you need to calculate space complexity?
  - To determine the efficiency of an algorithm.
    - Ex, If a program takes up a lot of memory space, the compiler will not let you run it.

- How to calculate Space Complexity of an Algorithm?
  - By calculating the space occupied by the variables used in an algorithm.
    - **Space Complexity = Auxiliary space + Space use by input values**

The Size of Primitive Data Types



Space Complexity calculation through examples

- Example #1

Space complexity is O(1), or constant

```java
package COEN352.TUT_2.SpaceComplexity.Examples;
 // @author Mustafa Daraghmeh
public class Example_1 {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        int a = 10;
        int b = 99999999;
        int c = a + b;
        System.out.println(c);
    }
}
```

Constant Space Complexity occurs when the program doesn't contain any loops, recursive functions or call to any other functions.

Space Complexity calculation through examples

- Example #2

Space complexity is O(n) or linear

```java
package COEN352.TUT_2.SpaceComplexity.Examples;
import java.util.Scanner;
 // @author Mustafa Daraghmeh
public class Example_2 {
    public static void main(String[] args) {
        int n, i, sum = 0;
        Scanner in = new Scanner(System.in);
        System.out.println("Enter an integer");
        n = in.nextInt();

        int [] arr=new int [n];
        for(i = 0; i < n; i++){
            arr[i]=(int) Math.round(Math.random()*10);
            sum = sum + arr[i];
        }
        System.out.printf("SUM: %d\n", sum);
    }
}
```

Linear space complexity occurs when the program contains any loops.

ANY QUESTIONS