Topics to cover:

1. Algorithm analysis in Java
2. Programming practice

1. Algorithm analysis

1.1 What is an algorithm?

An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations.According to its formal definition, an algorithm is a finite set of instructions carried out in a specific order to perform a particular task. It is not the entire program or code; it is simple logic to a problem represented as an informal description in the form of a flowchart or pseudocode.

1.2 How does an algorithm work?

Algorithms are step-by-step procedures designed to solve specific problems and perform tasks efficiently in the realm of computer science and mathematics. These powerful sets of instructions form the backbone of modern technology and govern everything from web searches to artificial intelligence. Here's how algorithms work:

- Input: Algorithms take input data, which can be in various formats, such as numbers, text, or images.
- Processing: The algorithm processes the input data through a series of logical and mathematical operations, manipulating and transforming it as needed.

- Output: After the processing is complete, the algorithm produces an output, which could be a result, a decision, or some other meaningful information.
- Efficiency: A key aspect of algorithms is their efficiency, aiming to accomplish tasks quickly and with minimal resources.
- Optimization: Algorithm designers constantly seek ways to optimize their algorithms, making them faster and more reliable.
- Implementation: Algorithms are implemented in various programming languages, enabling computers to execute them and produce desired outcomes.

1.3 Why do we need algorithms?

You require algorithms for mainly the following reasons:

Scalability

It aids in your understanding of scalability. When you have a sizable real-world problem, you must break it down into small steps to analyze it quickly.

Performance

The real world is challenging to break down into smaller steps. If a problem can be easily divided into smaller steps, it indicates that the problem is feasible.

After understanding what an algorithm is, why you need an algorithm, you will look at how to write one using an example.

1.4 What is algorithm analysis?

Algorithm analysis is an important part of **computational complexity theory**, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of **time** and **space** resources required to execute it.

1.4.1 Why is algorithm analysis important?

- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

1.4.2 Types of algorithm analysis

The two most important concepts in algorithm analysis are as follows:

i) **Time complexity:** It considers the number of operations performed,

ii) **Space complexity**: It considers the memory used by variables and data structures.

1.4.2.1 Time complexity

There are following three types of time-complexity in Algorithm Analysis:

- Best case
- Worst case
- Average case

1.4.2.1.1 Best case time complexity

It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time. In this case, the execution time serves as a lower bound on the algorithm's time complexity.

1.4.2.1.2 Average case time complexity

The average-case complexity of an algorithm is the amount of some computational resource (typically time) used by the algorithm, averaged over all possible inputs.

1.4.2.1.3 Worst case time complexity

It gives an upper bound on the resources required by the algorithm. In the case of running time, the worst-case time complexity indicates the longest running time performed by an algorithm given any input of size n, and thus guarantees that the algorithm will finish in the indicated period of time.

1.4.2.2 Asymptotic notation

Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n. There are three different notations: big O, big Theta (Θ), and big Omega (Ω).

1.4.2.2.1 Omega Notation (Ω-Notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm. The execution time serves as a lower bound on the algorithm’s time complexity. It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

1.4.2.2.2 Theta Notation (Θ-Notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm. You add the running times for each possible input combination and take the average in the average case.

1.4.2.2.3 Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

- It is the most widely used notation for Asymptotic analysis.
- It specifies the upper bound of a function.
- The maximum time required by an algorithm or the worst-case time complexity.
- It returns the highest possible output value(big-O) for a given input.
- Big-Oh(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.

1.4.2.2.4 A simple example for all the three cases

Let us consider the following array {1,2,3,4,5} and we are going to find element 1 here. We need just one operation here (or one unit of time) to search this element in the array as this is the first element (i.e., 0th index). The time complexity here will be the best-case complexity because this is the lower bound. In other words, the searching process here will take the lowest possible unit of time.

Now, suppose the integer 1 is in the second position (i.e., 1st index) in the array such as follows: {2,1,3,4,5}. In this case we need 2 units of time to search 1. Similarly, we need 3, 4 and 5 units of time to search for element 1 if it resides at position 3, 4 and 5, respectively. This will be called the average case and therefore the required amount of time on average will be calculated as follows.

(1+2+3+4+5)/5 = 3

In general, the equation stands as follows:

(1+2+.....+n)/n

n(n+1)/2n = (n+1)/2

Since this is a linear relation and hence the complexity is directly proportional to n.

Finally, the worst case scenario will happen when element 1 is at the end of the array such as {2,3,5,4,1}. Note that the arrangement of the first 4 elements does not matter as we only consider 1 to be at the end. In this situation, we will need 5 units of time to retrieve 1. Therefore, this depends on the input size (n) of the array. This is called the worst-case complexity which is directly proportional to n.

In summary, all the three cases will be as follows:

Best case:	Ω(1)
Average case:	Θ((n+1)/2)
Worst case:	O(n)

From now on, we will mostly talk about the worst case because it is very important to find out what is the maximum unit of time required for a certain algorithm to complete. In the above simple search algorithm, the worst case will take a maximum of 5 (n=5) units of time. Therefore, calculating the worst case time complexity ensures that a particular algorithm will not take more than a certain amount of time. It will never take more than that.

1.4.2.2.5 Illustration with simple programs

Consider the following part of a program:

```
public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

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

We will not consider the required time for creating a Scanner object because that is fixed for many similar types of programs. However, running several loops can significantly impact the runtime of the program and thus it is important to consider that. In the above code, the outer loop of i will run for n times (i.e., 0 to n-1). For each of its iterations, the inner loop of j will run for n times as well. Therefore the string “Hello” will be printed n^2 times, which means that the above nested loop will take no more than n^2 units of time (for n^2 operations). Now, the time-complexity (worst-case) here is $O(n^2)$.

Now consider another program as below.

```
public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        for(int i=0;i<n;i++){
                for(int j=0;j<m;j++){
                        System.out.println(“Hello”);
                }
        }
}
```

In the above code, the inner loop of j will run for m times for each iteration of the outer loop of i. Therefore it needs (n*m) operations and thus the time complexity here is $O(n*m)$.

Now consider our third program as below.

```
public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        for(int i=0;i<n;i++){
                System.out.println(“Hello”);
        }
        for(int j=0;j<m;j++){
                System.out.println(“Hello”);
        }
}
```

In the above code, each loop is independent of each other. The loop of i will run for n times and the loop of j will run for m times. We can simply add the time units here. Therefore it needs (n+m) operations and thus the time complexity here is $O(n+m)$.

Now, if either of n and m is significantly larger than the other then we can write the complexity equal to the larger number because the smaller number will not affect the runtime unless the

corresponding for loop consists of intensive calculations. For example, if n=100 and m=5, we can say that the time complexity here is O(n).

1.4.2.2.6 Calculating unit of time for different order of operations

Consider three algorithms with time complexities O(n), O(n²) and O(n³). The time required for different numbers of inputs for all these algorithms are shown in the following table.

Input size (n)	O(n)	O(n²)	O(n³)
1	1	1	1
2	2	4	8
3	3	9	27
4	4	16	64
5	5	25	125
…	…	…	…
…	…	…	…
10^5	10^5	10^10	10^15

As we can see that for input size 1, there is no difference in run time of all the above algorithms. However, as the input size increases, there is a sharp increase in the runtime in the second algorithm and even more in the third one.

1.4.2.2 Space complexity

When an algorithm is run on a computer, it necessitates a certain amount of memory space. The amount of memory used by a program to execute it is represented by its space complexity.

1.4.2.2.1 Example of space complexity

With an example, you will go over how to calculate space complexity in this section. Here is an example of computing the multiplication of array elements:

1. int mul, i

2. While i < = n do
3. mul <- mul * array[i]
4. i <- i + 1
5. end while
6. return mul

Let S(n) denote the algorithm's space complexity. In most systems, an integer occupies 4 bytes of memory. As a result, the number of allocated bytes would be the space complexity.

Line 1 allocates memory space for two integers, resulting in S(n) = 4 bytes multiplied by 2 = 8 bytes. Line 2 represents a loop. Lines 3 and 4 assign a value to an already existing variable. As a result, there is no need to set aside any space. The return statement in line 6 will allocate one more memory case. As a result, S(n)= 4 times 2 + 4 = 12 bytes.

Because the array is used in the algorithm to allocate n cases of integers, the final space complexity will be fS(n) = n + 12 = O (n).