🎓 Student Material: Algorithm Analysis & Arrays (with Java)

⏱ Session 1: Time & Space Complexity (2 hours)

🧠 1. What Is an Algorithm?

An algorithm is a step-by-step procedure to solve a problem. But not all algorithms are equal — some are faster or use less memory than others.

We measure this using:

- Time Complexity → how long it takes to run.
- Space Complexity → how much memory it uses.

⚙ 2. Time Complexity

We measure growth rate as input size `n` increases.

Big O Notation	Meaning	Example
O(1)	Constant time	Accessing an array element
O(n)	Linear time	Iterating through a list
$O(n^2)$	Quadratic time	Nested loops
O(log n)	Logarithmic	Binary Search

💻 Java Examples

```java
// O(1) - Constant Time
int getFirst(int[] arr) {
    return arr[0];
}

// O(n) - Linear Time
int sum(int[] arr) {
    int total = 0;
    for (int x : arr)
        total += x;
    return total;
}
```

```java
// O(n^2) - Quadratic Time
void printPairs(int[] arr) {
    for (int i = 0; i < arr.length; i++)
        for (int j = 0; j < arr.length; j++)
            System.out.println(arr[i] + ", " + arr[j]);
}
```

---

3. Space Complexity

Space complexity measures extra memory required beyond the input data.

Example:

```java
// O(1) space (no extra memory)
int findMax(int[] arr) {
    int max = arr[0];
    for (int n : arr)
        if (n > max) max = n;
    return max;
}

// O(n) space (recursion stack)
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

Exercises

1. Write a Java method to find the average of an integer array and analyze its time complexity.
2. Identify the time complexity of this snippet:

   ```java
   for (int i = 0; i < n; i++)
       for (int j = 0; j < 10; j++)
           System.out.println(i + j);
   ```

3. Write a recursive function to compute Fibonacci numbers and determine its time and space complexity.

🕑 Session 2: Arrays and Sorting Algorithms (2 hours)

🧠 1. Arrays in Java

An array stores a fixed number of elements of the same type.

Example:

```java
int[] numbers = {3, 1, 4, 1, 5};
System.out.println("First element: " + numbers[0]);
```

Common Operations

- Traversal (looping through all elements)
- Searching for an element
- Finding min/max/average

✏️ Array Exercises

1. Write a program to find the maximum and minimum elements in an array.
2. Compute the sum and average of array elements.
3. Count how many elements are greater than a given number.

⚙️ 2. Bubble Sort

Idea: Repeatedly compare adjacent elements and swap if they're in the wrong order. After each pass, the largest element "bubbles up" to the end.

💻 Java Example

```java
void bubbleSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        boolean swapped = false;
        for (int j = 0; j < arr.length - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) break; // optimization
```

```
    }
}
```

Complexity:

- Best: O(n)
- Worst: O(n^2)
- Space: O(1)

⚙️ 3. Selection Sort

Idea: Find the smallest element and place it at the beginning. Repeat for the rest.

💻 Java Example

```java
void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[minIdx])
                minIdx = j;
        }
        int temp = arr[minIdx];
        arr[minIdx] = arr[i];
        arr[i] = temp;
    }
}
```

Complexity:

- Best: O(n^2)
- Worst: O(n^2)
- Space: O(1)

✏️ Sorting Exercises

1. Implement Bubble Sort and print the array after each pass.
2. Modify Selection Sort to sort in descending order.
3. Compare the number of swaps in Bubble Sort vs Selection Sort for the same array.

🕒 Session 3: Searching Algorithms (2 hours)

🧠 1. Sequential (Linear) Search

Idea: Check each element one by one until the key is found.

💻 Java Example

```java
int linearSearch(int[] arr, int key) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}
```

Complexity:

- Best: O(1)
- Worst: O(n)
- Space: O(1)

🧠 2. Binary Search

Idea: Only works on sorted arrays. Repeatedly divide the search range in half.

💻 Java Example

```java
int binarySearch(int[] arr, int key) {
    int low = 0, high = arr.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key)
            return mid;
        if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
```

Complexity:

- Best: O(1)

- Worst: O(log n)
- Space: O(1)

🧩 3. Sorting + Searching Integration

You can use sorting algorithms to prepare data for binary search:

```
int[] data = {4, 2, 9, 1, 6};
bubbleSort(data);
int index = binarySearch(data, 6);
System.out.println("6 found at index: " + index);
```

✏️ Searching Exercises

1. Write a Linear Search that returns how many times the key appears in the array.
2. Modify Binary Search to count how many comparisons are made.
3. Combine Selection Sort + Binary Search to find an element efficiently.

📊 Summary Chart

Algorithm	Best Case	Worst Case	Average	Space	Sorted Input?
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	No
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No
Linear Search	O(1)	O(n)	O(n)	O(1)	No
Binary Search	O(1)	O(log n)	O(log n)	O(1)	Yes

🎯 Expected Learning Outcomes

By the end of these sessions, students should be able to:

- Analyze the time and space complexity of algorithms.
- Implement sorting algorithms (Bubble, Selection).
- Implement searching algorithms (Sequential, Binary).
- Choose appropriate algorithms based on efficiency and data conditions.
