Assignment 1:

Case 1:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += i;
}
```

- **Expected Complexity**: O(n)
- **Reason**: A single loop running `n` times.

Case 2:

```
int result = 1;
for (int i = 1; i <= 5; i++) {
    result *= i;
}
```

- **Expected Complexity**: O(n)
- **Reason**: A single loop iterating from 1 to a constant number.

Case 3:

```
int x = 5;
int y = 10;
int max = (x > y) ? x : y;
```

- **Expected Complexity**: O(1)
- **Reason**: Simple conditional statement with no loops.

Case 4:

```
int a = 5;
int b = 3;
int result = a * b;
```

- **Expected Complexity**: O(1)
- **Reason**: Simple arithmetic operation with no loops.

Case 5:

```
int value = 10;
while (value > 0) {
    value--;
}
```

- **Expected Complexity**: O(n)
- **Reason**: While loop running `n` times.

Case 6:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        sum += i + j;
    }
}
```

- **Expected Complexity**: O(n^2) => O(1) because the 10 is constant and not changing.
- **Reason**: Two nested loops running `n` times each.

Explanation:

Outer Loop: The outer loop runs n times. Since n=10, so the outer loop will iterate from i=0 to i=9.

Inner Loop: The inner loop runs a fixed number of times (10) for each iteration of the outer loop. This means that for each value of i, the inner loop iterates exactly 10 times, independent of the value of i.

Even though the inner loop runs a constant 10 times for each iteration of the outer loop, it still results in a total of 10×10=100 iterations (or generally n×n if we don't fix n to 10).

In terms of Big-O notation:

Since there are n iterations of the outer loop and each of those n iterations includes another n (10 in this case) iterations in the inner loop, this results in n×10=n^2 in a general case when both loops have the same upper limit n. Thus, the time complexity is O(n^2), because as n grows, the total number of operations will grow quadratically due to the nested loop structure.

Case 7:

```
int count = 0;
for (int i = 1; i <= 16; i *= 2) {
    count++;
}
```

- **Expected Complexity**: O(log n)
- **Reason**: The loop’s variable doubles each iteration.

Case 8:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        sum += i;
    }
}
```

- **Expected Complexity**: O(n)
- **Reason**: A single loop with a constant-time condition inside.

Case 9:

```
int product = 1;
for (int i = 1; i <= 5; i++) {
    product *= i;
}
int result = product * 2;
```

- **Expected Complexity**: O(n)
- **Reason**: A loop and a constant-time multiplication.

Case 10:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 5; j++) {
        sum += i + j;
    }
}
```

- **Expected Complexity**: O(n * m) => O(1) because n and m are constants (they don't change).
- **Reason**: Two nested loops running `n` and `m` times respectively.

Explanation:

Outer Loop: The outer loop runs n times. In this case, n=10, so i iterates from 0 to 9. However, we'll consider it as n for the general case.

Inner Loop: The inner loop runs a fixed number of times, m=5, for each iteration of the outer loop.

Total Number of Iterations

Since the inner loop runs m times (5 times in this example) for each of the n iterations of the outer loop, the total number of executions is: $n \times m$ This gives a time complexity of $O(n \times m)$.

Case 11:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    for (int j = i; j < 10; j++) {
        sum += i + j;
    }
}
```

- **Expected Complexity**: O(n^2)
- **Reason**: Inner loop depends on the value of `i`.

Explanation:

Outer Loop: The outer loop runs n times. Here, n=10, so i iterates from 0 to 9.

Inner Loop: The inner loop's range depends on the current value of i. For each iteration of i, the inner loop starts from j=i and runs up to j=9. This means that the inner loop executes fewer times as i increases.

Let's break down the number of times the inner loop runs for each iteration of i:

- **When i=0**: The inner loop runs from j=0 to j=9, so it executes **10 times**.
- **When i=1**: The inner loop runs from j=1 to j=9, so it executes **9 times**.
- **When i=2**: The inner loop runs from j=2 to j=9, so it executes **8 times**.
- **When i=3**: The inner loop runs from j=3 to j=9, so it executes **7 times**.
- **When i=4**: The inner loop runs from j=4 to j=9, so it executes **6 times**.
- **When i=5**: The inner loop runs from j=5 to j=9, so it executes **5 times**.
- **When i=6**: The inner loop runs from j=6 to j=9, so it executes **4 times**.
- **When i=7**: The inner loop runs from j=7 to j=9, so it executes **3 times**.
- **When i=8**: The inner loop runs from j=8 to j=9, so it executes **2 times**.
- **When i=9**: The inner loop runs from j=9 to j=9, so it executes **1 time**.

Total Number of Executions

The total number of inner loop executions is the sum of these runs:

10+9+8+7+6+5+4+3+2+1

This is the sum of the first n integers. The formula for the sum of the first nnn integers is:

$$Sum = \frac{n(n+1)}{2}$$

For n=10, this gives:

$$\frac{10 \cdot 11}{2} = 55$$

Time Complexity

The total number of inner loop executions grows as $\frac{n(n+1)}{2}$, which simplifies to O(n^2) as nnn grows larger. Therefore, the time complexity of this code is O(n^2).

$$\frac{n(n+1)}{2} = \frac{n^2+n}{2} = \frac{n^2}{2} + \frac{n}{2}$$

Since $\frac{n^2}{2}$ is the term with the highest growth rate, the $\frac{n}{2}$ term grows more slowly and becomes negligible as n gets lareger. Now, we also know that $\frac{n^2}{2} = \frac{1}{2} \cdot n^2$), we ignore the constant factor of ½ since Big-O notation only cares about the order of growth, not the exact scaling factor, this leaves us with: O(n^2).

Case 12:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < i; j++) {
        sum += j;
    }
}
```

- **Expected Complexity**: O(n^2)
- **Reason**: Inner loop runs based on the increasing value of `i`.

Explanation:

Outer Loop: The outer loop runs n times. For this example, n=10, so the outer loop iterates from i=0 to i=9.

Inner Loop: The number of times the inner loop runs depends on the current value of i. Specifically, for each iteration of i, the inner loop will execute i times, running from j=0 up to j=i−1.

Let's break down the number of times the inner loop runs for each iteration of iii:

- **When i=0**: The inner loop does not execute because j<i is false (0 < 0 is false).
- **When i=1**: The inner loop runs 1 time (for j=0j).
- **When i=2**: The inner loop runs 2 times (for j=0 and j=1).
- **When i=3**: The inner loop runs 3 times (for j=0, j=1, and j=2).
- **When i=4**: The inner loop runs 4 times (for j=0, j=1, j=2, and j=3).
- **When i=5**: The inner loop runs 5 times (for j=0, j=1, j=2, j=3, and j=4).
- **When i=6**: The inner loop runs 6 times.
- **When i=7**: The inner loop runs 7 times.
- **When i=8**: The inner loop runs 8 times.
- **When i=9**: The inner loop runs 9 times.

So in this case we have:

$$Sum = \frac{n(n-1)}{2}$$

In Big-O notation, like with case 11, we have:

$$\frac{n(n-1)}{2} = \frac{n^2 - n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

We ignore the $\frac{n}{2}$, so we have: $\frac{n^2}{2} = \frac{1}{2} \cdot n^2$ since we ignore constants $(\frac{1}{2})$ we end up with n^2.

Case 13:

```
int i = 0;
int sum = 0;
while (i < 10) {
    sum += i;
    i++;
}
for (int j = 0; j < 5; j++) {
    sum += j;
}
```

- **Expected Complexity**: O(n + m)
- **Reason**: Two separate loops running `n` and `m` times each.

Case 14:

```
int x = 0;
for (int i = 0; i < 10; i++) {
    for (int j = i; j < 10; j++) {
        x += j;
    }
}
```

- **Expected Complexity**: O(n^2)
- **Reason**: Inner loop depends on `i`, making it a nested quadratic loop.

Case 15:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += i;
}
for (int j = 0; j < 5; j++) {
    sum += j;
}
```

- **Expected Complexity**: O(n + m)
- **Reason**: Two separate loops running `n` and `m` times each.

Assignment 2: **Write an O(n) complexity algorithm to find the sum of an array.**
Initial Variables: `int[] arr = {1, 2, 3, 4, 5}`
Solution:

```
int sum = 0;
for (int i = 0; i < arr.length; i++) {
    sum += arr[i];
}
```

Assignment 3: **Write an O(n^2) complexity algorithm to print pairs of numbers.**
Initial Variables: `int[] arr = {1, 2, 3}`
Solution:

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

Assignment 4: **Write an O(log n) complexity algorithm using a loop.**
Initial Variables: `int n = 16`
Solution:

```
int count = 0;
for (int i = 1; i < n; i *= 2) {
    count++;
}
```

Assignment 5: **Find a number in an array (O(n) and O(log n) solutions).**
Initial Variables: `int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`, `int target = 5`

- **O(n) Solution**:

```
boolean found = false;
for (int i = 0; i < arr.length; i++) {
    if (arr[i] == target) {
        found = true;
        break;
    }
}
```

- **O(log n) Solution** (Assume the array is sorted):

```
int left = 0, right = arr.length - 1;
boolean found = false;
while (left <= right) {
    int mid = (left + right) / 2;
    if (arr[mid] == target) {
        found = true;
        break;
    } else if (arr[mid] < target) {
        left = mid + 1;
    } else {
        right = mid - 1;
    }
}
```

Assignment 6: **Find the maximum element in an array (O(n) and O(1) solutions).**
Initial Variables: `int[] arr = {1, 3, 5, 7, 9, 11}`

- **O(n) Solution**:

```
int max = arr[0];
for (int i = 1; i < arr.length; i++) {
    if (arr[i] > max) {
        max = arr[i];
    }
}
```

- **O(1) Solution** (Assume array is sorted in ascending order):

```
int max = arr[arr.length - 1];
```

Explanation:

- For the O(n) solution, we iterate through the entire array to find the maximum value, making the complexity O(n).
- For the O(1) solution, if the array is already sorted, we can directly access the last element, which will be the maximum, resulting in O(1) complexity.