Sure, here is a more structured and detailed version of the notes:

Big O Notation

Big O Notation is a mathematical way to compare the efficiency of two pieces of code. It’s not about how concise or readable the code is, but how well it performs, especially as the size of the input grows.

Key Points

1. **Not Based on Execution Time:**
    - Example:
        - Code 1: Executes in 15 seconds.
        - Code 2: Executes in 1 minute.
    - It might seem like Code 1 is more efficient based on time, but time alone isn’t a reliable measure. If we run the same code on a more powerful computer, it might execute faster due to better hardware, not because the code itself is better.
2. **Measured in Number of Operations:**
    - Instead of focusing on execution time, Big O considers the number of operations a computer must perform to execute the code. This is known as Time Complexity.
    - Additionally, Space Complexity measures the amount of memory the code consumes.

3. **Time vs. Space Trade-Off:**
    - Code 1 might execute faster but use more memory.
    - Code 2 might execute slower but use less memory.
    - Depending on your needs (speed vs. memory usage), one can prefer one code over the other.

Worst Case Big O

- **Omega (Ω):** Best case scenario.
- **Theta (Θ):** Average case scenario.
- **Omicron (O):** Worst case scenario, commonly referred to as Big O.

Example:

```
+---+---+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---+---+---+---+---+---+---+
  Ω             θ           O
```

- Finding an element in an array using a for loop:
    - 1st element: Best case (Ω)
    - Last element: Worst case (O)
    - Middle element: Average case (Θ)

Note: In Big O notation, there is no best case and average case; technically, they are Omega and Theta. Big O always represents the worst case.

Big O Complexity

It is measured based on n elements (input) with n operations.

Examples:

1. **Big O(n):**

```
def do_some_stuff(n):
    for i in range(n):
        print(i)
```

```
do_some_stuff(5)
```

- Here, for n elements, the loop goes over n operations. If n = 5, it is calculated as Big O(n).

2. **Big O(n^2):**

```
def do_some_stuff(n):
    for i in range(n):
        for j in range(n):
            print(i, j)

do_some_stuff(5)
```

- Here, for n elements, two nested loops go over $n * n$ operations. If n = 5, it is 5 * 5 = 25 operations, which is calculated as Big O(n^2).

3. **Big O(1):**

```
def do_some_stuff(n):
    return n + 5 + 20 + 1

do_some_stuff(5)
```

- Here, for n elements, only a single operation is performed, even though n is added with multiple constants. It is calculated as Big O(1).

4. **Big O(log n):**

```
# Binary search example
arr = [1, 2, 3, 4, 5, 6, 7, 8]
find_ele = 1

# 1st iteration
left_array = [1, 2, 3, 4]
right_array = [5, 6, 7, 8]

# 2nd iteration
left_array = [1, 2]
right_array = [5, 6]

# 3rd iteration
left_array = [1]
right_array = [5]
```

```
# Found in 3rd iteration on left
```

- To find element 1, it took 3 iterations. This is calculated as $\log_2(8) = 3$, so here 2 is raised to the power 3 to get $\log_2(8)$, which is calculated as log(n).

Dropping Constants

```python
def do_some_stuff(n):
    for i in range(n):
        print(i)

    for j in range(n):
        print(j)

do_some_stuff(5)
```

- Here, for n elements, two n operations are performed, which is `n + n = 2n`, i.e., Big O(2n). The constant 2 can be ignored, resulting in Big O(n).

Dropping Non-Dominants

```python
def do_some_stuff(n):
    for i in range(n):
        for j in range(n):
            print(i, j)

    for i in range(n):
        print(i)

do_some_stuff(5)
```

- Here, for n elements, one n*n operation and one n operation are performed, which is `n² + n`, i.e., Big O(n^2 + n). Since `n²` is dominant and n is non-dominant, the non-dominant term is ignored, resulting in Big O(n^2).

Different Terms for Input

```
def do_some_stuff(a, b):
    for i in range(a):
        print(a)
    for i in range(b):
        print(b)

do_some_stuff(10, 5)
```

- Here, for `a` and `b` elements, `a + b` operations are performed, which is Big O(a + b), not Big O(2n), because the operations for `a` and `b` are different.

This should provide a clearer and more detailed understanding of Big O Notation and its various aspects.