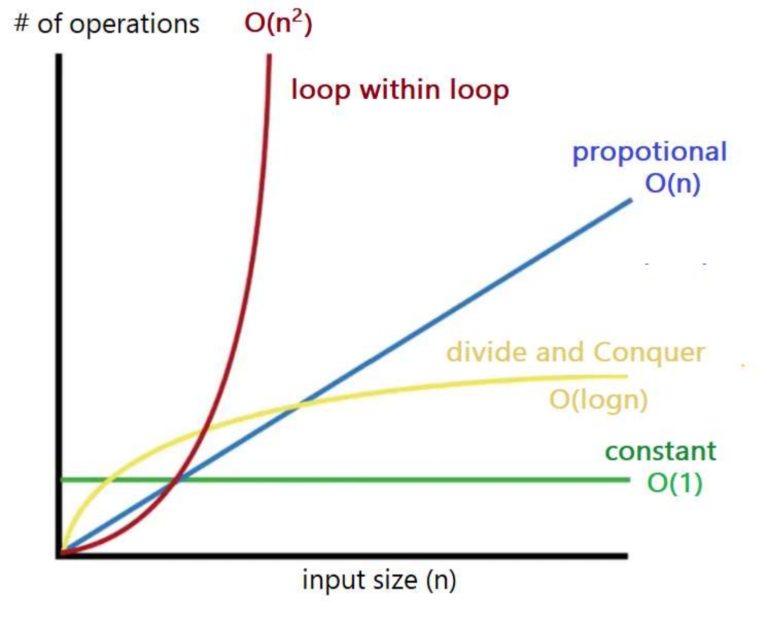
Big O

- A Measure to the worst-case scenario for a
- A way of comparing two sets of code mathematically about how efficient they run.
- We measure time complexity. But it is not measured in time, but in number of operations it takes to complete something.

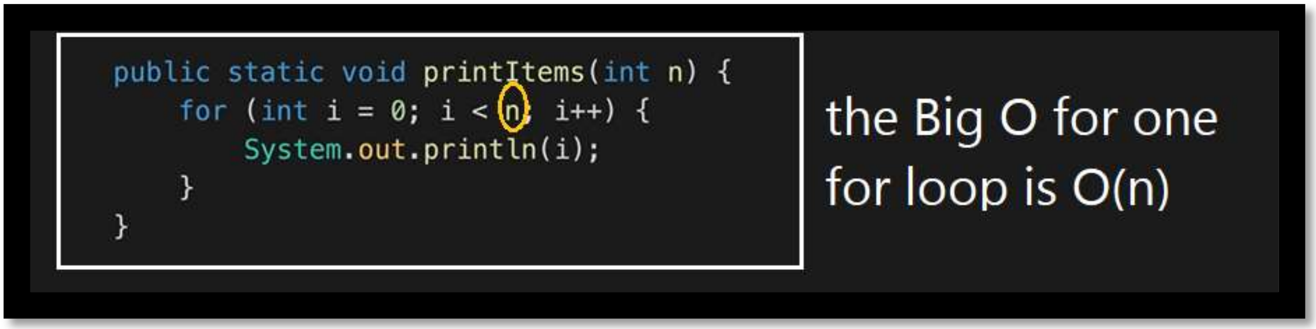


Rules for simplifications for Big O

1. **Drop constants:**
   If Big O = O(n+n) = O(2n)
   It can be simplified to O(n)
2. **Drop non-dominant:**
   If Big O= $O(n^2)+O(n) = O(n^2+n)$
   It can be simplified to $O(n^2)$
3. **Different terms for inputs**
   If Big O=O(a) + O(b) it cannot be simplified further as a <> b

Examples on how to calculate Big O:

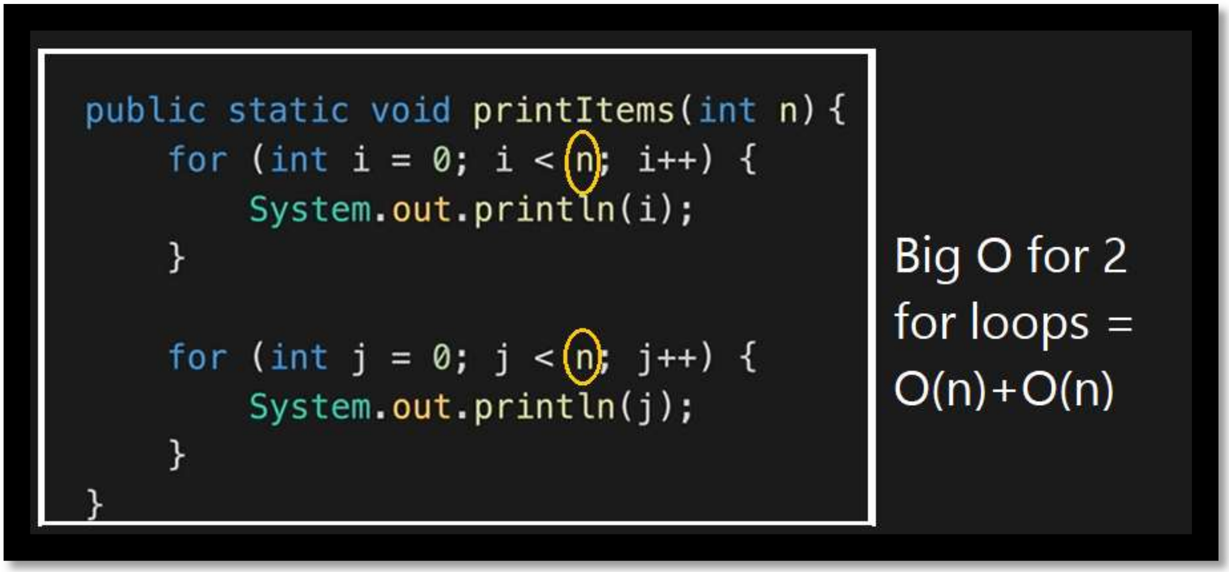
1- If we have one loop, Big O= O(n), n is the max number of operations this loop takes.



```
public static void printItems(int n) {
    for (int i = 0; i < n; i++) {
        System.out.println(i);
    }
}
```

the Big O for one for loop is O(n)

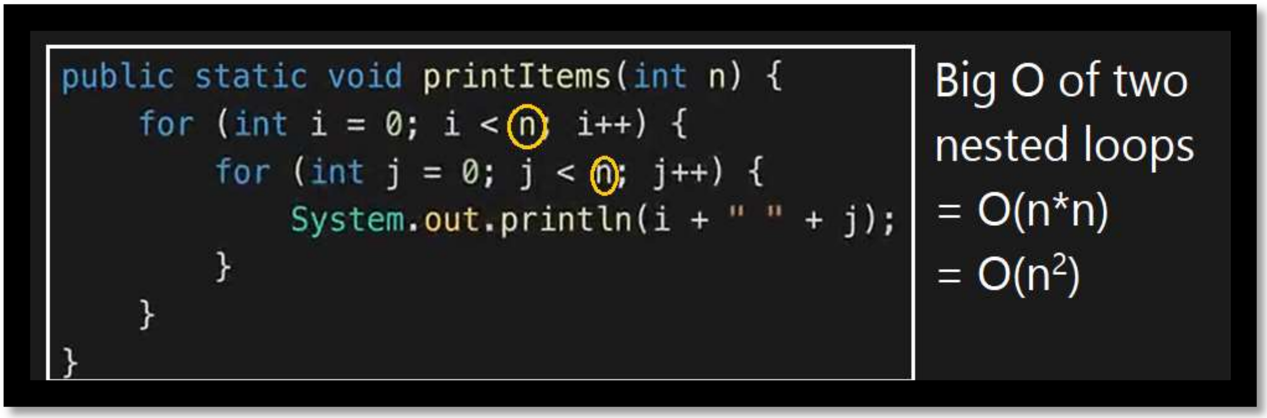
2- If we have 2 loops, Big O= O(n)+O(n)=O(2n)=O(n)



```
public static void printItems(int n){
    for (int i = 0; i < n; i++) {
        System.out.println(i);
    }

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

Big O for 2 for loops = O(n)+O(n)

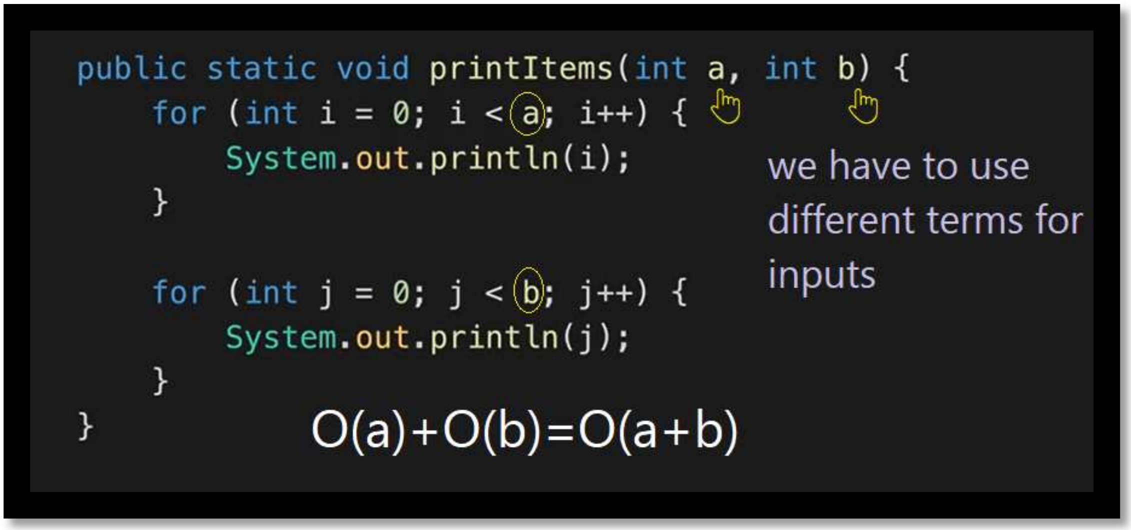
3- If we have 2 nested loops, Big O=O(n*n)=O(n^2)



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

Big O of two nested loops = O(n*n) = O(n^2)

4- If we have 2 loops of different lengths (inputs), Big O= O(a)+O(b)



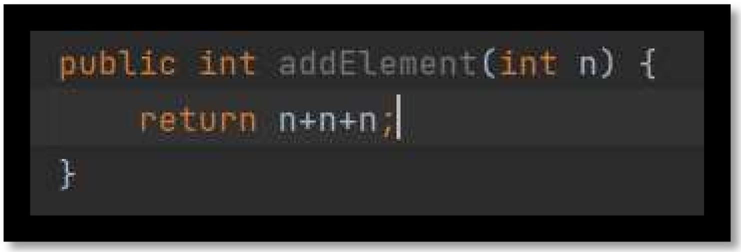
```java
public static void printItems(int a, int b) {
    for (int i = 0; i < a; i++) {
        System.out.println(i);
    }

    for (int j = 0; j < b; j++) {
        System.out.println(j);
    }
}
```

we have to use different terms for inputs

O(a)+O(b)=O(a+b)

4. Big O= O(1)

O (1) does not mean that there will be only one operation; but it means that as n grows, the number of operations stays constant.

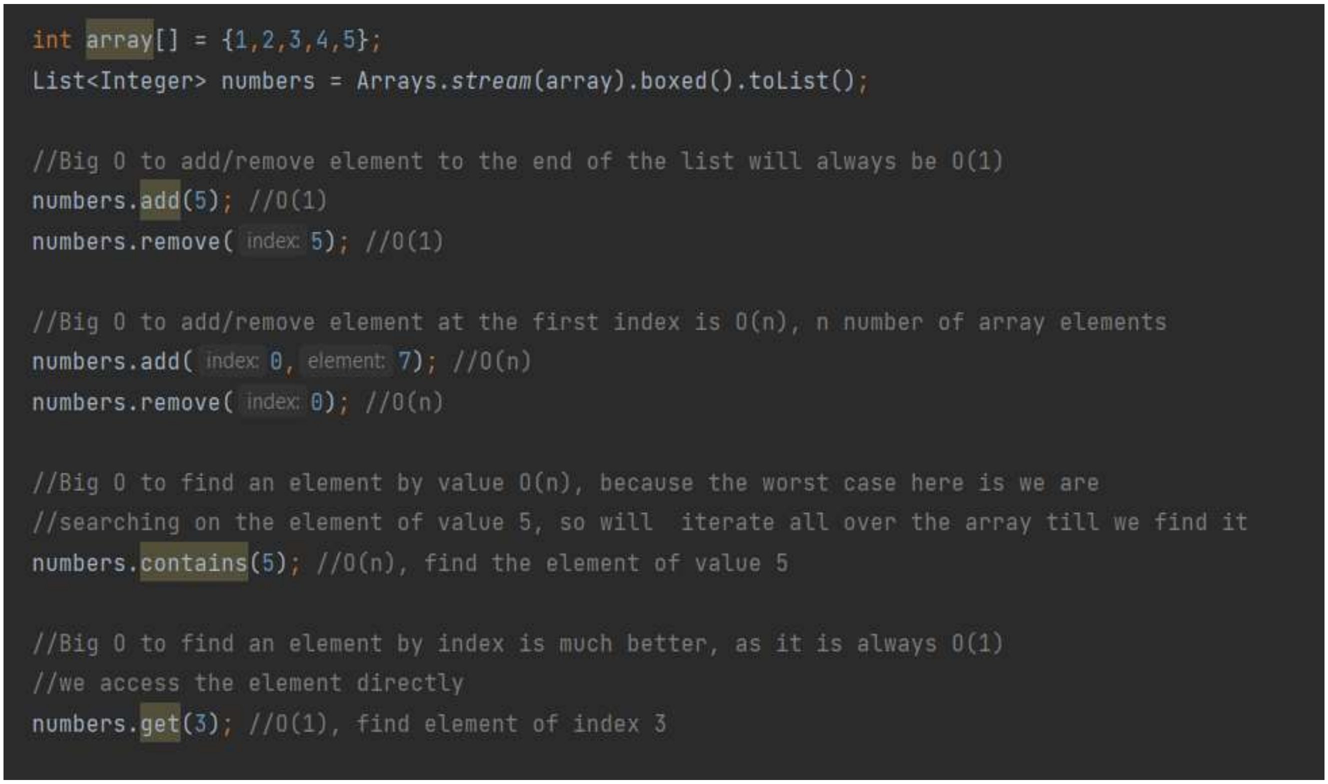


```java
public int addElement(int n) {
    return n+n+n;
}
```

O(1) is the most efficient Big O

Big O for array Lists:

- If we want to add/remove an element to the end of the list, then no re-indexing needed so Big O = O(1)
- If we want to add/remove element from the beginning of the list, so we need to re-index the whole list, Big O= O(n) , n is the arraylist length
- If we want to add a new element to in the middle of the arrayList, Big O= O(n-i) = O(n), i is the index at which we will insert the new element and start re-indexing the remaining list.
- If we search for an element by index, Big O = O(1)
- If we seatch for an element by value, Big O= O(n)



```java
int array[] = {1,2,3,4,5};
List<Integer> numbers = Arrays.stream(array).boxed().toList();

//Big O to add/remove element to the end of the list will always be O(1)
numbers.add(5); //O(1)
numbers.remove( index: 5); //O(1)

//Big O to add/remove element at the first index is O(n), n number of array elements
numbers.add( index: 0, element: 7); //O(n)
numbers.remove( index: 0); //O(n)

//Big O to find an element by value O(n), because the worst case here is we are
//searching on the element of value 5, so will  iterate all over the array till we find it
numbers.contains(5); //O(n), find the element of value 5

//Big O to find an element by index is much better, as it is always O(1)
//we access the element directly
numbers.get(3); //O(1), find element of index 3
```