**Informal Definition of Time Efficiency Notations**

Dr. Byun

08/04/2020

When we express the **time efficiency** (**or time complexity**) of an algorithm, we typically use the Big-Oh (O) notation or Big-Theta () notation. There is another notation, Big-Omega (), but we do not use it much in our class. In the document, I introduce the informal definition of the two notations of Big-Oh (O) and Big-Omega (). You will learn more formal mathematical definition later.

Basically, **algorithm analysis** can be defined as **identifying the time complexity category** of an algorithm. Most popular categories of time complexity are 1 (= constant), log n, n, n\*log n, n2, n3, 2n, and n!. So, if an algorithm’s execution takes linear time (for example, a single for loop with the index from one to n), we express it as O(n) or (n).

**Big-Oh (O) Notation**

To get the basic idea of the Big-Oh (O) notation, let’s consider the *SequentialAdd*  algorithm as an example.

1. Algorithm *SequentialAdd* (A[0..n-1])
2. // Input: An array A with *n* numbers from the index 0 to n-1
3. // Output: Sum of the numbers in the array A
4. sum ← 0
5. i ← 0
6. while ( i < n ) do
7. sum ← sum + A[i]
8. i ← i + 1
9. return sum

To express the algorithm’s time efficiency in Big-Oh notation, we should know the **number of executions of all operations in the algorithm**. The following table presents the number of executions of operations for the general array size n.

Table 1. Number of operations executed in *SequentialAdd*.

|  |  |  |
| --- | --- | --- |
| Operation | Line # | Array size: *n* |
| < | 6 | n+1 |
| + | 7 | n |
| + | 8 | n |
| return | 9 | 1 |

We can represent the total execution time of the algorithm by adding all numbers like this:

T(n) = (n + 1) + n + n + 1

= 3\*n + 2

Now, to represent the execution time T(n) in the Big-Oh notation, we have to **consider only the highest order in the formula T(n) and remove coefficients**. Note that T(n) has two terms “3\*n” and “2\*1”. If we remove the coefficients in the terms, we have just “n” and “1”. Since “n” (= linear) is higher than “1” (= constant), we can say that T(n) belongs to O(n). We can represent it as below

T(n) = 3\*n + 2

O(n)

As another example, let’s assume that there’s an algorithm with multiple operations and the number of executions of all operations is 2\*n2 + 5\*n + 20. Then, how can you represent the time efficiency of the algorithm in the Big-Oh notation?

Note that 2\*n2 + 5\*n + 20 has three terms such as “2\*n2”, “5\*n”, and “20\*1”. Again, we should remove the coefficients. After that, we should consider only the highest order among them. Since “n2”is the highest order, we can represent the algorithm’s efficiency as O(n2). In other words,

T(n) = 2\*n2 + 5\*n + 20

O(n2)

You may think that this approach is a very simple approximation. However, remember that **the main purpose of the O notation is to find the dominant factor (or order) in algorithm execution**. So, you can simply ignore coefficients and lower terms (= orders).

**Example**

Let’s assume that the number of executions of operations in an algorithm is 0.5\*n3 + 500\*n2 + 100\*n + 700. Represent the time efficiency of the algorithm in the O notation.

**Solution**

Since the dominant order is n3, we can say that it’s O(n3). The coefficients of n2 and n are 500 and 100, which are quite large when compared to the coefficients of n3 (= 0.5). But it does not matter. We only consider the highest order.

Another issue is that the sum of all executions of all operations is not trivial. For the *SequentialAdd* algorithm, there are only four operations, and the sum of all executions is not difficult. However, if an algorithm is large and there are many operations, it would be very difficult to add the numbers of executions of all operations. To make the calculation simple, you **don’t need to consider all operations in an algorithm**. You only need to **consider the number of executions of the basic operation**. For example, in the *SequentialAdd* algorithm, the < operation of the line number 6 is the basic operation and executed “n + 1” times. We know that the time efficiency of “n+1” is O(n). Recall that the basic operation of an algorithm is the operation that contributes the most to the total running time. Therefore, the number of executions of the basic operation clearly indicates the main factor (= order) of the algorithm.

**Exercise**

For the following pseudocode *Display\_2D\_Stars*, represent its time efficiency in the O notation.

1. Algorithm Display\_2D\_Stars (n)
2. // Input: A positive integer number n
3. // Output: Display the \* symbol in two dimensions.
4. i ← 0
5. while ( i < n ) do
6. j ← 0
7. while ( j < n ) do
8. write '\*'
9. j ← j + 1
10. write '\n'
11. i ← i + 1
12. return 0

**Solution**

In the algorithm, the < operation in the line number 7 is the basic operation. The dominant factor of the basic operation is n2, so the time efficiency of the algorithm can be expressed as O(n2).

**Big-Theta () Notation**

We generally use the O notation and occasionally Big-Theta () notation to indicate the time efficiency of the algorithm. For instance, you can use either O(n) or (n) to represent the time complexity of the *SequentialAdd* algorithm. However, they are not the same notation. There is a situation where the notation can not be used. For example, consider the following sequential search algorithm:

1. Algorithm *SequentialSearch*(A[0..n − 1], K)
2. // Searches for a key value K in an array A using sequential search
3. // Input: An array A[0..n − 1] and a search key K
4. // Output: The index of the first element in A that matches K
5. // or −1 if there are no matching elements
6. i ←0
7. while ((i < n) AND (A[i] K)) do
8. i ←i + 1
9. if i < n
10. return i
11. else
12. return −1

The basic operation of the algorithm is the comparison operation() in the line number 7. Note that the execution time of the basic operation depends on the input array A and K value. In other words, its best case time efficiency and worst time efficiency are different. If there are no matching numbers or the first matching number happens to be the last one on the array, the algorithm makes the largest number of key comparisons and the basic operation will be executed *n* times. However, if the first number in the array is the search key value *K*, the basic operation will be executed only one time. For the algorithm, we can say that the time efficiency is O(n). Note that **the O notation should represent the worst case time efficiency of an algorithm if worst case and best case time efficiencies are different**. However, **the notation can not be used if the worst case and the best case have different time efficiencies**.

That is, the notation can be used only if an algorithm has the same time complexity in all cases. However, the O notation can be used for an algorithm which has different best and worst cases. The **O notation** indicates the **upper bound** of the algorithm.

For example, if the algorithm time efficiency is O(n2), then the algorithm will take n2 or less orders. However, the notation indicates that an algorithm’s time efficiency always takes the same order. Therefore, if an algorithm’s time efficiency is (n2), it means that the algorithm execution time always takes the order of n2 irrespective of the best case or the worst case.

**Example**

Let’s consider a sample algorithm below. Can the time efficiency be expressed as O (n)? How about (n)?

1. Algorithm *Average* (A[0..n-1])
2. // Input: An array A with *n* numbers from the index 0 to n-1
3. // Output: Average of the numbers in the array A
4. sum ← A[0]
5. for i ← 1 to n - 1 do
6. sum ← sum + A[i]
7. avg ← sum / n
8. return avg

**Solution**

You can represent it as both O(n) and (n). Mathematically, (n) is more accurate because the algorithm always takes a linear order. However, many computer scientists commonly use O(n) and (n) interchangeably in this case.

**Example**

Let’s consider a sample algorithm below. Can you represent the algorithm’s time efficiency as O(n2)? How about (n2)?

1. Algorithm *CheckDuplicate* (A[0..n-1])
2. // Input: An array A with *n* numbers from the index 0 to n-1
3. // Output: The algorithm checks for duplicates in the array A.
4. // It returns false if there are duplicates.
5. for i ← 0 to n - 1 do
6. for j ← 0 to n - 1 do
7. if ( (i j) AND (A[i] = A[j]) )
8. return false
9. return true

**Solution**

You can represent it as O(n2) but you can not represent it as (n2). Note that the algorithm’s best case and worst case are different.

**More Examples for Notations**

In this section, you will learn more examples of the O and notations.

**Question 1**

Suppose that there is a programming assignment in the homework and the instructor requires the program to have O(n2) time efficiency. If the number of executions of the basic operation in your program is “3\*n2 + 2”, does your program meet the requirement of the instructor?

**Answer**

Your program meets the requirement of O(n2) because the O notation indicates the upper limit (or upper bound) of an algorithm. Because the instructor requires O(n2), your algorithm should be in the category of n2 or less order. “3\*n2 + 2” belongs to the n2 category. You can represent this situation like below

3\*n2 + 2 O(n2)

**Question 2**

This question has the same situation as the question 1. But if your program runs “4\*n + 5” times, does it meet the requirement of O(n2)?

**Answer**

Yes, your program still meets the requirement of O(n2). Again, the O notation indicates the upper bound of the algorithm, and O(n2) means n2 or less. Since “4\*n + 5” belongs to the n (= linear) category, you can represent it as

4\*n + 5 O(n2)

**Question 3**

This question has the same situation as the question 1. But if your program runs “4\*n3 + 5\*n + 2” times, does it meet the requirement of O(n2)?

**Answer**

No, your program doesn’t meet the requirement of O(n2) because the program has the category of n3 which is higher than n2. You can represent this situation as below:

4\*n3 + 5\*n + 2 O(n2)

**Question 4**

This question has the same situation as the question 1. But let’s assume that the instructor requires(n2) and your program runs “4\*n + 5” times. Does your program meet the requirement of (n2)?

**Answer**

No, your program doesn’t meet the requirement of (n2). Recall that the Big Theta notation is used to indicate the same order. Because the program has the category of nwhich is less than n2, your program fails the requirement. You can represent this situation like:

4\*n + 5 (n2)