Big O

Big O definition in Wikipedia is the next :

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

- Do not worry, I know that is ambiguous for you, here I will make it more simple, Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.
- In other words, it is an approximation value that can gives us an idea about the performance of our algorithm. It helps us know about efficiency and complexity of an algorithm without the need to actually it with test inputs on a real machine.

Big O

- Big O is not a speed comparison, it does not allow you to compare the speed of two algorithms. Rather, it is used to determine what factor the number of operations will grow by as the amount of input increases.
- The notation is useful when comparing algorithms used for similar purposes. Seeing the Big O of multiplication algorithms would allow you to compare their relative complexity. It wouldn't be useful to see the Big O of a multiplication algorithm and an addition algorithm side-by-side.
- Big O is a simple approximation of the worst-case-scenario, it reduces an algorithm to the most significant portion of complexity. If an algorithm could be represented as $n^2 + n$, as the inputs became large, the + n would become meaningless in the overall complexity and could be dropped. The same logic holds true for n3 + $n^2 + n$, as this would be simplified to n3. In the next skill, we will understand more about how to be calculated.

How to calculate Big O.

Remember, all calculations of the Big O is for the worst-case-scenario.

This means it is possible that the algorithm would return results earlier than the approximation.

Let’s practice!

In the code box below, we will find an algorithm that returns true if a given element exists in a given array, and false if not.

```
FUNCTION seek_elt(tab : ARRAY_OF INTEGER , val : INTEGER) : BOOLEAN
VAR
   i : INTEGER;
BEGIN
  FOR i FROM 0 TO tab.length-1 STEP 1  DO
      IF (val = tab[i]) THEN
          RETURN TRUE
      END_IF
  END_FOR
  RETURN FALSE
END
```

As we can see, the tab.length is our n (or input). Now, the main idea behind the BigO is to find the worst-case scenario. Returning to our example, the worst-case scenario is that we browse the whole array and we do not find the searched element. So our complexity is O(n)

Let's take a second example. This time we will have a nested loop.

```
FUNCTION contain_dup(tab : ARRAY_OF INTEGER) : BOOLEAN
VAR
  i,j : INTEGER;
BEGIN
  FOR i FROM 0 TO tab.length-1 STEP 1  DO
       FOR j FROM 0 TO tab.length-1 STEP 1  DO
           IF (i<>j AND tab[i] = tab[j]) THEN
               RETURN TRUE
           END_IF
       END_FOR
  END_FOR
   RETURN FALSE ;
END
```

in the example above, we have a nested loop with each one of these loops has a complexity of O(n). So the whole complexity of our algorithm will be O(n * n) which equal to $O(n^2)$

Our third example will be the same algorithm as before but with a little change. We'll make the second loop not starting from the beginning of the array, but it will start from the i position.

```
FUNCTION contain_dup(tab : ARRAY_OF INTEGER) : BOOLEAN
VAR
  i,j : INTEGER;
BEGIN
  FOR i FROM 0 TO tab.length-1 STEP 1  DO
       FOR j FROM i+1 TO tab.length-1 STEP 1  DO
           IF (i<>j AND tab[i] = tab[j]) THEN
               RETURN TRUE
           END_IF
       END_FOR
  END_FOR
   RETURN FALSE ;
END
```

With this change, the first loop will keep the complexity of O(n), while the second loop will get a complexity of O(n - i).

This is what we call the logarithmic format. So the complexity of this algorithm will be O(n log n)

‹ Previous next ›