ASYMPTOTIC NOTATIONS

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

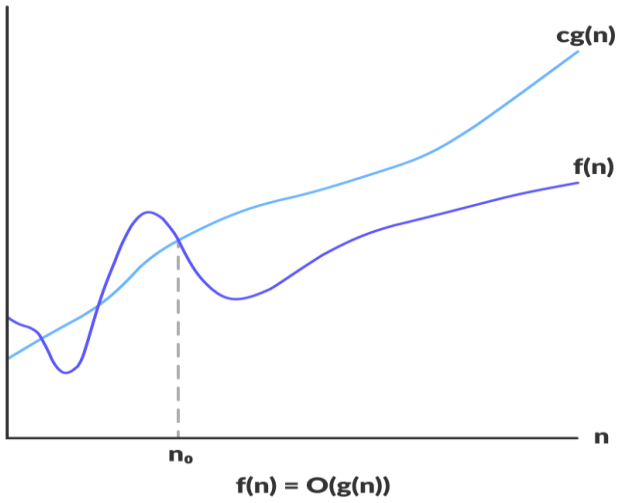
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm. Big-O gives the upper bound of a function



O(g(n)) = { f(n): there exist positive constants c and n_0

such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ }

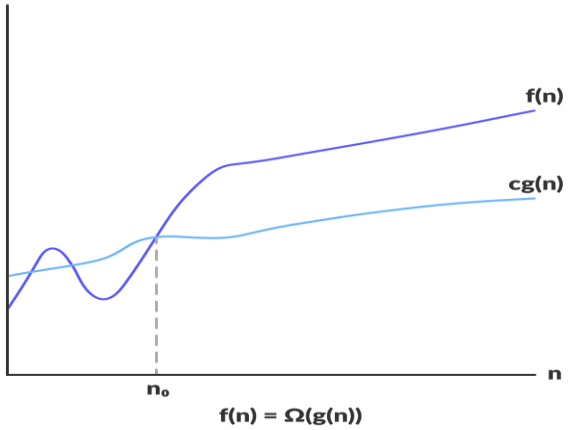
The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

Since it gives the worst-case running time of an algorithm, it is widely used to analyse an algorithm as we are always interested in the worst-case scenario.

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm. Omega gives the lower bound of a function



Ω(g(n)) = { f(n): there exist positive constants c and n_0

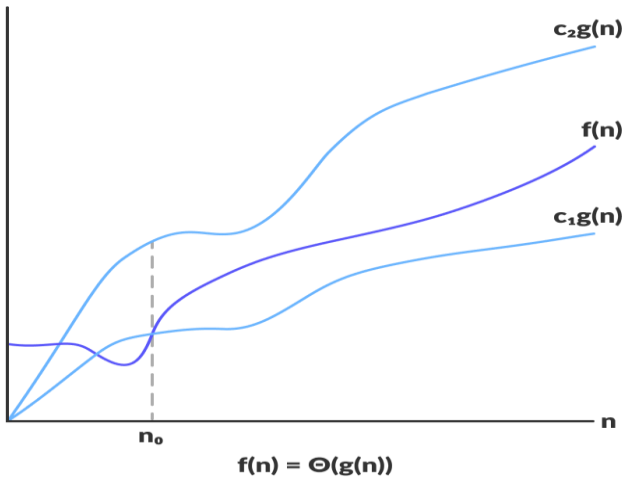
such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analysing the average-case complexity of an algorithm. Theta bounds the function within constants factors



For a function g(n), Θ(g(n)) is given by the relation:

Θ(g(n)) = { f(n): there exist positive constants c_1, c_2 and n_0

such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ }

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.

If a function f(n) lies anywhere in between $c_1 g(n)$ and $c_2 g(n)$ for all $n \geq n0$, then f(n) is said to be asymptotically tight bound.

MATHEMATICAL ANALYSIS FOR NON-RECURSIVE ALGORITHMS

General Plan for Analyzing the Time Efficiency of Non-recursive Algorithms:

1. Decide on a parameter (or parameters) indicating an input's size.

2. Identify the algorithm's basic operation (in the inner most loop).

3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional

property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.

4. Set up a sum expressing the number of times the algorithm's basic operation is executed.

5. Using standard formulas and rules of sum manipulation either find a closed form formula for the count or at the least, establish its order of growth.

EXAMPLE 1: Consider the problem of finding the value of the largest element in a list of n numbers. Assume that the list is implemented as an array for simplicity.

ALGORITHM Max Element(A[0..n − 1])

//Determines the value of the largest element in a given array

//Input: An array A[0..n − 1] of real numbers

//Output: The value of the largest element in A

Max val ←A[0]

for i ←1 to n − 1 do

if A[i]>maxval

maxval←A[i]

return maxval

Time complexity

• The measure of an input's size here is the number of elements in the array, i.e., n.

• **There are two operations in the for loop's body:**

1. The comparison A[i]> maxval and
2. The assignment max val←A[i].

• The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.

• The number of comparisons will be the same for all arrays of size n;

therefore, there is no need to distinguish among the worst, average, and best cases here.

• Let C(n) denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive. Therefore, the sum for C(n) is calculated as follows:

$$C(n)=\sum_{I=1}^{n-1} 1$$

n-1 (upper bond)

I=1 (lower bond)

n-1-1+1

=n-1 // Ignore constants(1)

TC=n

O(n)

EXAMPLE 2: Consider the element uniqueness problem: check whether all the elements in a given array of n elements are distinct.

ALGORITHM Unique Elements (A[0..n − 1])

//Determines whether all the elements in a given array are distinct

//Input: An array A[0..n − 1]

//Output: Returns "true" if all the elements in A are distinct and "false" otherwise

```
for i ←0 to n − 2 do
    for j ←i + 1 to n − 1 do
      if A[i]= A [j ] return false
return true
```

time complexity

1. Input size =n
2. Basic operation is comparison. So it is denoted by **C** which takes 1 unit of time.
3. The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only.
4. One comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits i + 1 and n − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and n −2.

$$C(n)=\sum_{I=0}^{n-2}\sum_{j=i+1}^{n-1}1$$

$$=\sum_{I=0}^{n-2}(n-1)-(i+1)+1$$

$$=\sum_{I=0}^{n-2}n-1-i-1+1$$

$$=\sum_{I=0}^{n-2}n-1-i$$

Substitute i=0,1,2,3…….n

So, n-1-i

n-1-0, n-1-1, n-1-2, n-1-3…….

n-1, n-2, n-3…..n-1-n+2

(n-1)+(n-2)+(n-3)+…..1

=n(n-1)/2

=n2-n //as n2 is large so consider only that

=n2

Example: 3

The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

```
ALGORITHM Binary(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ←1
    while n > 1 do
        count←count +1
        n←[n/2]
return count
```

Time complexity

• An input's size is n.

• The loop variable takes on only a few values between its lower and upper limits.

• Since the value of n is about halved on each repetition of the loop, the answer should be about log2 n.

• The exact formula for the number of times.

• The comparison n > 1 will be executed is actually [log2 n] +1.

MATHEMATICAL ANALYSIS FOR RECURSIVE ALGORITHMS

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.

4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

EXAMPLE 1: Compute the factorial function F(n) = n! for an arbitrary non negative integer n. Since n!= 1•. . . . • (n − 1) • n = (n − 1)! • n, for n ≥ 1 and 0!= 1 by definition, we can compute F(n) = F(n − 1) • n with the following recursive algorithm

ALGORITHM :Fact(n)

//Computes n! recursively

//Input: A nonnegative integer n

//Output: The value of n!

if n = 0 return 1

else

return F(n − 1) * n

Time complexity

• For simplicity, we consider n itself as an indicator of this algorithm's **input size. i.e.1.**

• **The basic operation of the algorithm is multiplication;** whose number of executions we denote **M(n).** Since the function F(n) is computed according to the formula F(n) = F(n −1)•n for n >0.

• base case: Second, by inspecting the pseudocode's exiting line, we can see that when n =0, the algorithm performs no multiplications.

Thus, the recurrence relation and initial condition for the algorithm's number of multiplications

M(n):

M(n) = M(n − 1) + 1 for n >0,

M(0)=0 for n =0.

Method of backward substitutions

M(n) = M(n − 1)+1 // if in case, we perform multiplication then.

substitute M(n − 1) = M(n − 2) +1

= [M(n − 2) + 1]+ 1

= M(n − 2)+2 substitute M(n − 2) = M(n − 3) +1

= [M(n − 3) + 1]+ 2

= M(n − 3) + 3

…

= M(n − i) + i

…

= M(n − n) + n

= n.

Therefore **M(n)=n**

EXAMPLE 2: consider educational workhorse of recursive algorithms: theTower of Hanoi puzzle. We have n disks of different sizes that can slide onto any of three pegs. Consider A (source), B (auxiliary), and C (Destination). Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top

Time complexity

Refer notes.