ASYMPTOTIC NOTATIONS AND PROPERTIES

Asymptotic notation is a notation, which is used to take meaningful statement bout the efficiency of a program. The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency.

To compare and rank such orders of growth, computer scientists use three notations, they are:

- O-Big oh notation
- Ω-Big omega notation
- Θ-Big theta notation

Let *t(n)*and *g(n)*can be any nonnegative functions defined on the set of natural numbers. The algorithm running time *t(n)*usually indicated by its basic operation count *C(n),* and *g(n),* Some simple function to compare with the count.

(i) O-Big oh notation

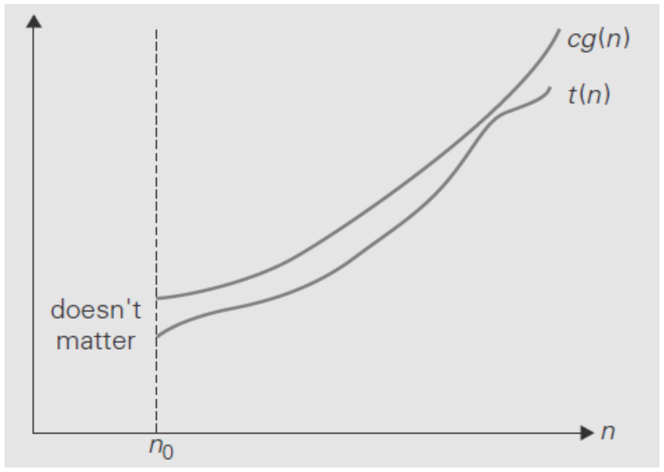
This Notation is used to express the upper bound of an algorithm running time ,It represents the worst case of an algorithm time complexity

The longest amount of time an algorithm can possibly take complete

A function t*(n)* is said to be in *O(g(n)),* denoted $t\ (n)\in O(g(n))$, if *t (n)* is bounded above by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant *c* and some nonnegative integer n_0 such that

$$(n)\leq cg(n) for all\ n\geq n_0.$$

Where *t(n)*and *g(n)*are non negative functions defined on the set of natural numbers. O = Asymptotic upper bound = Useful for worst case analysis = Loose



Equation:

f(n) ≤c.g(n) $forall\ n \geq n_0$.

Example:

Given f(n)=100n+5 prove that f(n)=o(n)

Solution:

Let f(n)=100n+5

Replace 5 by n

f(n) 100+n

f(n)=101n

100n+5=101n

f(n) ≤c.g(n) $forall\ n \geq n_0$.

100n+5≤101n forall ≥5

C=101,n_0=5, g(n)=n,

f(n)=o(n) proved

(i) Ω-Big omega notation

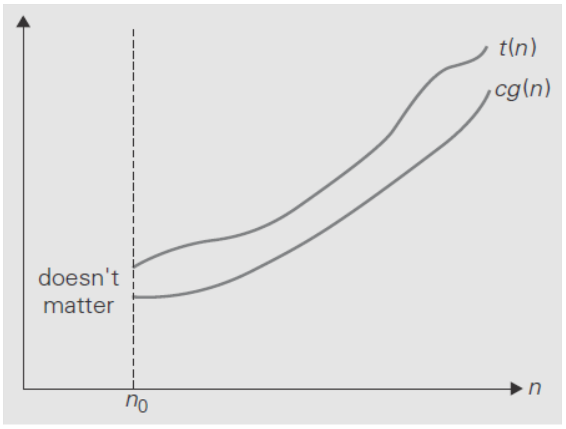
Omega Notation minimum time taken by an algorithm for execution time called omega Notation

A function *t(n)* is said to be in Ω*(g(n)),* denoted *t(n)* ∈ Ω*(g(n)),* if *t(n)* is bounded below by some positive constant multiple of *g(n)* for all large n, i.e., if there exist some positive constant c and some non negative integer n_0 such that

$$t(n) \geq cg(n) \text{ foralln } \geq n_0.$$

Where *t(n)*and *g(n)*are non negative functions defined on the set of natural numbers.

Ω=Asymptotic lower bound=Useful for best case analysis= Loose bound



Equation :$t(n) \geq cg(n)$ for all $n \geq n_0$.
Given $f(n)=10n^2+4n+3$ proved $f(n)= \Omega(n^2)$

Solution: let $f(n)=10n^2+4n+3$

Remove 4n+3

$f(n)= 10n^2$

$10n^{2+}4n+3=10n^2$

$f(n) \geq cg(n)$ for all $n \geq n_0$.

$C=10,g(n)=n^2,n_o=1$

$f(n)= \Omega(n^2)$

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
Maxval ←A[0]
    for i ←1 to n − 1 do
        if A[i]>maxval
                maxval←A[i]
return maxval
```

Algorithm analysis

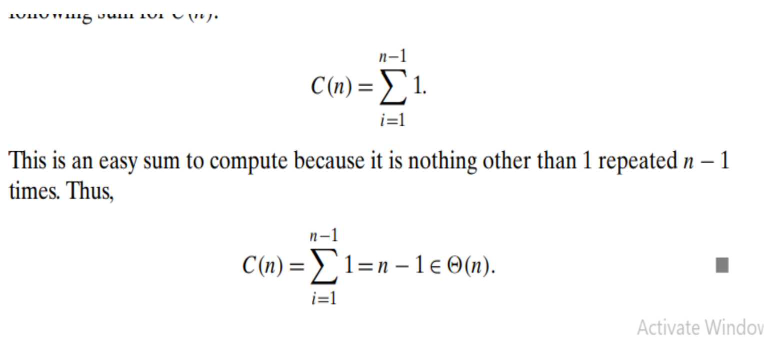
• The measure of an input's size here is the number of elements in the array, i.e., n.

• There are two operations in the for loop's body: o The comparison A[i]> maxval and o 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:**



following sum for $C(n)$.

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

This is an easy sum to compute because it is nothing other than 1 repeated $n - 1$ times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n). \quad ■$$

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 = (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 F(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
```

Algorithm analysis

• 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.
• The number of multiplications M(n) needed to compute it must satisfy the equality

$$M(n) = \underset{\substack{\uparrow \\ \text{To compute} \\ F(n-1)}}{M(n-1)} + \underset{\substack{\uparrow \\ \text{To multiply} \\ F(n-1) \text{ by } n}}{1} \quad \text{for } n > 0$$

M (n − 1*)* multiplications are spent to compute *F(n* − 1*)*, and one more multiplication is needed to multiply the result by *n*

Recurrence relations The last equation defines the sequence M(n) that we need to find. This equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely n − 1. Such equations are called recurrence relations or recurrences.

Solve the recurrence relation (n) = (n − 1) + 1, i.e., to find an explicit formula f o r M(n) in terms of n only.

To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if n = 0 return 1.

This tells us two things. First, since the calls stop when n = 0, the smallest value of n for which this algorithm is executed and hence M(n) defined is 0. Second, by inspecting the pseudo code'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

substitute $M(n - 1) = M(n - 2) + 1$

$M(n) = M(n - 1) + 1$

$= [M(n - 2) + 1] + 1$

$= M(n - 2) + 2$

$= [M(n - 3) + 1] + 2$

$= M(n - 3) + 3$

substitute $M(n - 2) = M(n - 3) + 1$

$= M(n - i) + i$

$= M(n - n) + n$

$=n$