FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

Analysis of algorithms means to investigate an algorithm's efficiency with respect to resources:

- **running time (time efficiency)** and
- **memory space (space efficiency)**

Time being more critical than space, we concentrate on Time efficiency of algorithms. The theory developed, holds good for space complexity also.

Experimental Studies: requires writing a program implementing the algorithm and running the program with inputs of varying size and composition. It uses a function, like the built-in clock() function, to get an accurate measure of the actual running time, then analysis is done by plotting the results.

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis: It uses a high-level description of the algorithm instead of an implementation. Analysis characterizes running time as a function of the input size, *n*, and takes into account all possible inputs. This allows us to evaluate the speed of an algorithm independent of the hardware/software environment. Therefore theoretical analysis can be used for analyzing any algorithm

Framework for Analysis

We use a hypothetical model with following assumptions

- Total time taken by the algorithm is given as a function on its input size
- Logical units are identified as one step
- Every step require ONE unit of time
- Total time taken = Total Num. of steps executed

Input's size: Time required by an algorithm is proportional to size of the problem instance. For e.g., more time is required to sort 20 elements than what is required to sort 10 elements.

Units for Measuring Running Time: Count the number of times an algorithm's **basic operation** is executed. (**Basic operation**: The most important operation of the algorithm, the operation contributing the most to the total running time.) For e.g., The basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

Consider the following example:

ALGORITHM sum_of_numbers (A[0… n-1])
// Functionality : Finds the Sum
// Input : Array of n numbers
// Output : Sum of 'n' numbers
i ← 0
sum ← 0
while i < n
    sum ← sum + A[i] ⟶ n
    i ← i + 1
return sum

Total number of steps for basic operation execution, C (n) = n

NOTE:
Constant of fastest growing term is insignificant: Complexity theory is an Approximation theory. We are not interested in exact time required by an algorithm to solve the problem. Rather we are interested in order of growth. i.e

- How much faster will algorithm run on computer that is twice as fast?
- How much longer does it take to solve problem of double input size?

We can crudely estimate running time by

T (n) ≈ Cop ∗ C (n)

Where,
T (n): running time as a function of n.
Cop : running time of a single operation.
C (n): number of basic operations as a function of n.

Order of Growth: For order of growth, consider only the leading term of a formula and ignore the constant coefficient. The following is the table of values of several functions important for analysis of algorithms.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Worst-case, Best-case, Average case efficiencies

Algorithm efficiency depends on the **input size n.** And for some algorithms efficiency depends on **type of input**. We have best, worst & average case efficiencies.

Worst-case efficiency: Efficiency (number of times the basic operation will be executed) **for the worst case input of size n.** *i.e.* The algorithm runs the longest among all possible inputs of size n.

Best-case efficiency: Efficiency (number of times the basic operation will be executed) **for the best case input of size n.** ***i.e.*** The algorithm runs the fastest among all possible inputs of size n.

Average-case efficiency: Average time taken (number of times the basic operation will be executed) **to solve all the possible instances (random) of the input.** NOTE: NOT the average of worst and best case

Asymptotic Notations

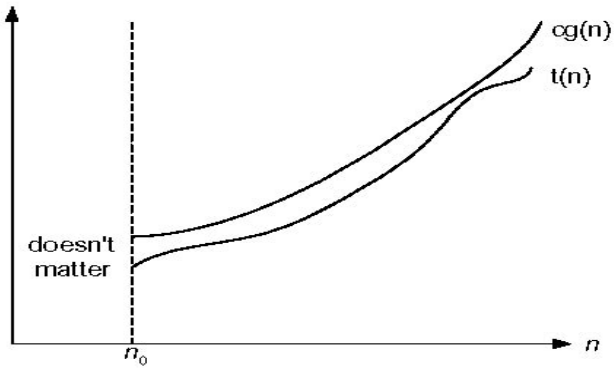
Asymptotic notation is a way of comparing functions that ignores constant factors and small input sizes. Three notations used to compare orders of growth of an algorithm's basic operation count are: **O, Ω, Θ notations**

Big Oh- O notation

Definition:

A function t(n) is said to be in O(g(n)), denoted **t(n) ∈ 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 n0 such that

t(n) ≤ cg(n) for all n ≥ n0



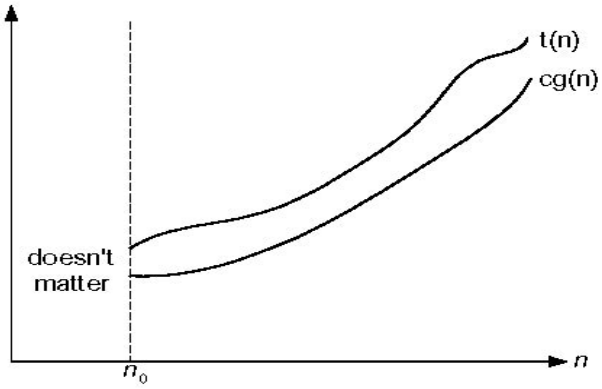
Big-oh notation: $t(n) \in O(g(n))$

Big Omega- Ω notation

Definition:

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 constant multiple of g (n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that

t(n) ≥ cg(n) for all n ≥ n0



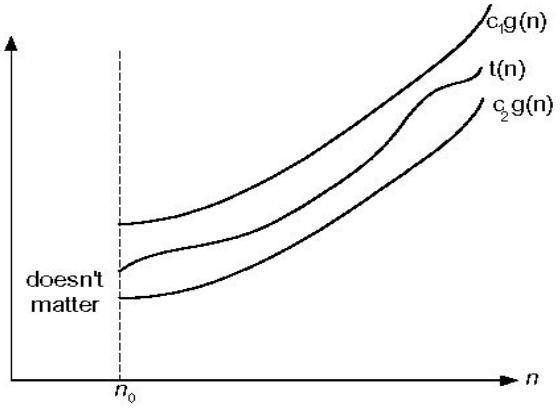
Big-omega notation: $t(n) \in \Omega(g(n))$

Big Theta- Θ notation

Definition:

A function t (n) is said to be in Θ(g (n)), denoted **t(n) ∈ Θ(g (n)),** if t (n) is bounded both above and below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that

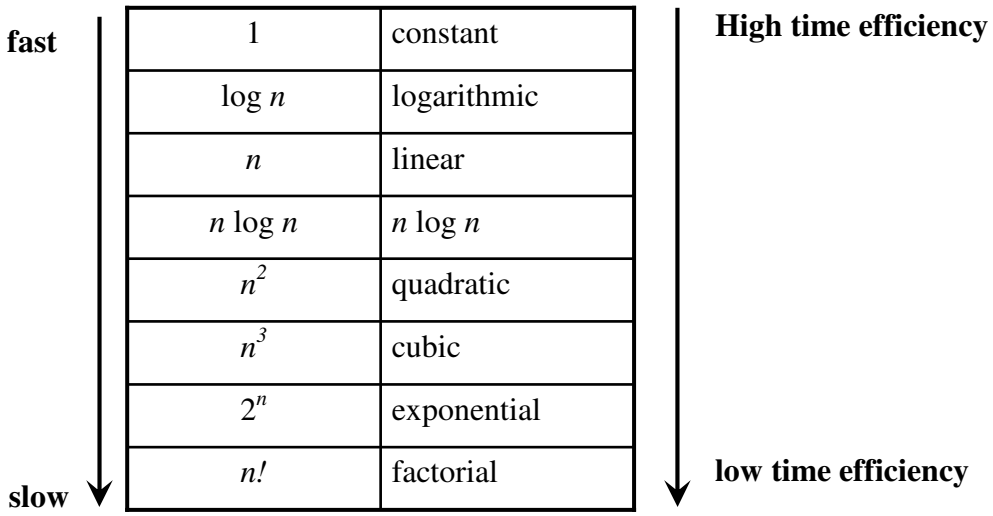
c2 g (n) ≤ t (n) ≤ c1 g (n) for all n ≥ n0



Big-theta notation: $t(n) \in \Theta(g(n))$

Basic Efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.



1	constant
log n	logarithmic
n	linear
n log n	n log n
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Mathematical analysis (Time Efficiency) of Non-recursive Algorithms

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on parameter n indicating **input size**
2. Identify algorithm's **basic operation**
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case efficiency** separately.
4. Set up **summation** for C(n) reflecting the number of times the algorithm's basic operation is executed.
5. Simplify summation using standard formulas

Example: Finding the largest element in a given array

ALOGORITHM *MaxElement*(A[*0..n*-1])

//Determines the value of 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

currentMax ← *A*[0]
for *i* ← 1 to *n* − 1 do
    if *A*[*i*] > *currentMax*
        currentMax ← *A*[*i*]
return *currentMax*

Analysis:

1. Input size: number of elements = n (size of the array)
2. Basic operation:
   a) Comparison
   b) Assignment
3. NO best, worst, average cases.
4. Let C (*n*) denotes number of comparisons: Algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bound between 1 and *n* – 1.

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

5. **Simplify summation** using standard formulas

$$\mathbf{C}(n) = \sum_{i=1}^{n-1} 1 \qquad \mathbf{1 + 1 + 1 + \ldots + 1 \text{ [(n-1) number of times]}}$$

$$\mathbf{C}(n) = \mathbf{n-1}$$

$$\mathbf{C}(n) \in \Theta(\mathbf{n})$$

Example: Element uniqueness problem

Algorithm *UniqueElements (A[0..n-1])*
//Checks 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
```

Analysis

1. Input size: number of elements = n (size of the array)
2. Basic operation: Comparison
3. Best, worst, average cases EXISTS.
   Worst case input is an array giving largest comparisons.
   - Array with no equal elements
   - Array with last two elements are the only pair of equal elements
4. Let C (*n*) denotes number of comparisons in worst case: Algorithm makes one comparison for each repetition of the innermost loop i.e., for each value of the loop's variable j between its limits *i + 1 and n – 1*; and this is repeated for each

value of the outer loop i.e, for each value of the loop's variable i between its limits *0 and n – 2*

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

5. **Simplify summation** using standard formulas

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

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

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

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

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

$$C(n) = (n-1)(n-1) - \frac{(n-2)(n-1)}{2}$$

$$C(n) = (n-1)((n-1) - \frac{(n-2)}{2})$$

$$C(n) = (n-1)\frac{(2n-2-n+2)}{2}$$

$$C(n) = (n-1)(n)/2$$
$$= (n^2 - n)/2$$
$$= (n^2)/2 - n/2$$
$$\mathbf{C}(\boldsymbol{n}) \in \Theta(\mathbf{n^2})$$

Mathematical analysis (Time Efficiency) of recursive Algorithms

General plan for analyzing efficiency of recursive algorithms:

1. Decide on parameter n indicating **input size**
2. Identify algorithm's **basic operation**
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case efficiency** separately.
4. Set up **recurrence relation**, with an appropriate initial condition, for the number of times the algorithm's basic operation is executed.
5. **Solve** the recurrence.

Example: Factorial function

ALGORITHM ***Factorial (n)***
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if $n = = 0$
    return 1
else
    return Factorial $(n - 1) * n$

Analysis:

1. Input size: given number = n
2. Basic operation: multiplication
3. NO best, worst, average cases.
4. Let M (n) denotes number of multiplications.

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0$$
$$M(0) = 0 \quad \textit{initial condition}$$

Where: M $(n - 1)$: to compute Factorial $(n - 1)$
1 :to multiply Factorial $(n - 1)$ by n

5. Solve the recurrence: Solving using *"Backward substitution method":*

$$\begin{aligned} M(n) &= M(n-1) + 1 \\ &= [\, M(n-2) + 1 \,] + 1 \\ &= M(n-2) + 2 \\ &= [\, M(n-3) + 1 \,] + 3 \\ &= M(n-3) + 3 \\ &\dots \end{aligned}$$

In the ith recursion, we have

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

When i = n, we have

$$= M(n-n) + n = M(0) + n$$

Since M (0) = 0

$$= n$$

$$\mathbf{M(n) \in \Theta(n)}$$

Example: Find the number of binary digits in the binary representation of a positive decimal integer

ALGORITHM *BinRec (n)*
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = = 1$
    return 1
else
    return *BinRec* ($\lfloor n/2 \rfloor$) + 1

Analysis:

1. Input size: given number = n
2. Basic operation: addition
3. NO best, worst, average cases.
4. Let A (*n*) denotes number of additions.

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1$$
$$A(1) = 0 \quad \textit{initial condition}$$

Where: A ($\lfloor n/2 \rfloor$) : to compute *BinRec* ($\lfloor n/2 \rfloor$)
1 : to increase the returned value by 1

5. Solve the recurrence:
   $A(n) = A(\lfloor n/2 \rfloor) + 1$ for $n > 1$
   Assume $n = 2^k$ (smoothness rule)
   $A(2^k) = A(2^{k-1}) + 1$ for $k > 0$; $A(20) = 0$
   Solving using "Backward substitution method":

$$A(2^k) = A(2^{k-1}) + 1$$
$$= [A(2^{k-2}) + 1] + 1$$
$$= A(2^{k-2}) + 2$$
$$= [A(2^{k-3}) + 1] + 2$$
$$= A(2^{k-3}) + 3$$
$$\dots$$

   In the ith recursion, we have

$$= A(2^{k-i}) + i$$

   When i = k, we have

$$= A(2^{k-k}) + k = A(2^0) + k$$

   Since $A(2^0) = 0$

$$\mathbf{A(2^k) = k}$$

   Since $n = 2^k$, HENCE $k = \log_2 n$

$$A(n) = \log_2 n$$
$$\mathbf{A(n) \in \Theta(\log n)}$$