Algorithms

An **algorithm** or effective procedure is any well-defined computational procedure that takes some values, as **input** and produces some values, as **output**.

Notations : $\lfloor x \rfloor$ is Floor $\lceil x \rceil$ is Ceiling lg n stands for $\log_2 n$

Pseudo code is a standard notation used to write an algorithm. It has the following conventions:

1. Indentation (**tab**s) is for structuring blocks.
2. We can use loop constructs (**for, while**, …)
3. '►' is for single line comment.
4. Assignment as a back arrow $j \leftarrow e$.
5. Variables are of local scopes.
6. Arrays have index in brackets e.g. A[i].
7. Attributes are accessed using square brackets. For an attribute 'attrib' of object 'x', we write *attrib[x]*.

```
INSERTION-SORT(A)
1.  for j ← 2 to length[A]
2.      do key ← A[j]
3.      ► Insert A[j] into A[1 .. j-1]
4.          i ← j – 1
5.          while i > 0 and A[i] > key
6.                  do A[i+1] ← A[i]
7.                      i ← i – 1
8.          A[i+1] ← key
```

```
% LATEX
\usepackage{algorithm}
\usepackage{algpseudocode}
\begin{algorithm}
\caption{Compute sum of integers in array}
\label{array-sum}
\begin{algorithmic}[1]
\Procedure{ArraySum}{$A$}
    \State $sum = 0$
    \For {each integer $i$ in $A$}
        \State $sum = sum + i$
    \EndFor
      \State Return $sum$
\EndProcedure
\end{algorithmic}
\end{algorithm}
```

Algorithm 1 Compute sum of integers in array

```
1: procedure ARRAYSUM(A)
2:     sum = 0
3:     for each integer i in A do
4:         sum = sum + i
5:     end for
6:     Return sum
7: end procedure
```

Algorithms are compared and based on five criteria
Completeness, **Correctness**, **Optimality**, **Time Complexity**, **Space Complexity**

Time Complexity : ***running time***, T(n), is affected by: Speed of the machine, Programming Language for implementation, Efficiency of the compiler , Input Organization, **Input size**. For simplification, we shall consider the **running time** of an algorithm as the number of "steps" executed.

Growth rates are used to predict the relationship between **input size** and **running time**.

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

Plots are used to compare growth rates. To compares n log n and n^2 using **MATLAB**.

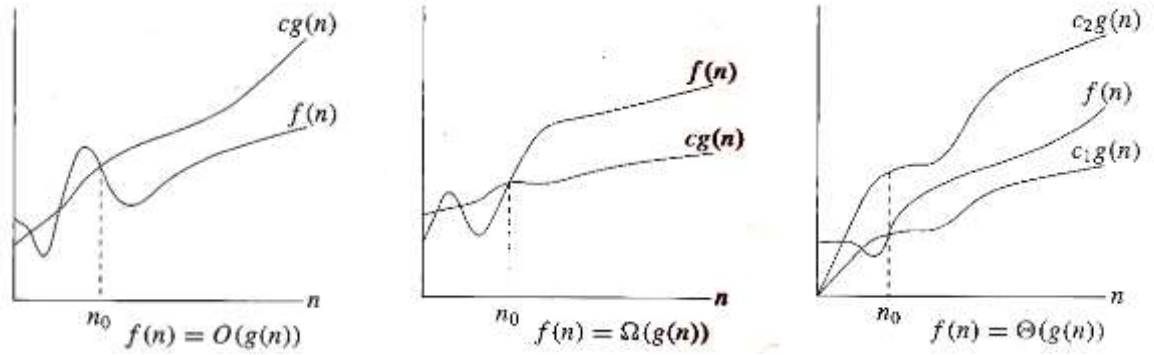
AVERAGE(n)	**Statement**	no. of times executed
1. sum ← 0	1	1
2. i ← 0	2	1
3. **while** i < n	3	n+1
4. number ← input_number()	4	n
5. sum ← sum + number	5	n
6. i ← i + 1	6	n
7. mean ← sum / n	7	1
8. **return** mean	8	1

T(n) = 4n + 5, ***growth rate*** of n.

We use, such as big-Oh. So,T(n) = O(n), i.e., it is of O(n) **complexity**.

asymptotic notation

- **O-notation (Big-O)** $O(g(n))$ = {f(n): there exist positive constants c and n_0 such that $0 \le$ **f(n) ≤ cg(n)** for all $n \ge n_0$}.
- **Ω-notation (Big-Omega)** $\Omega(g(n))$ = {f(n) : there exist positive constants c and n0 such that $0 \le$ **cg(n) ≤ f(n)** for all $n \ge n_0$}.
- **Θ-notation (Big-Theta)** $\Theta(g(n))$ = {f(n): there exist positive constants c_1, c_2, and n_0 such that $0 \le$ **c_1g (n) ≤ f (n) ≤ c_2g** (n) for all $n \ge n_0$}.



Example $3n^2 + 2n + 5 \in O(n^2)$, $7n \in O(n^2)$, $\sin n \in O(n^2)$ Tighter : $7n \in O(n)$, $\sin n \in O(1)$

Example Assume an algorithm called X-sort	X-sort is $O(n^2)$.
for j ---------- n while i ---------- n ---------- ----------	X-sort is Ω (n), neglecting the inner loop, as array is sorted.

Finding worst case rather than **average case**, for the following reasons:

- We are in safe side for allocating enough resources, as the worst case is an upper bound.
- In real-world applications, worst-case happens more frequent as in searching a database.
- Usually, the average-case is as bad as worst case, as for insertion sort both are quadratic.
- In some problems, it is not easy to define what is the average.

Recursive algorithms, are structured according to **divide-and-conquer**:

X-SORT(A, p, r) 1 if $p < r$ 2 then $q \leftarrow \lfloor (p + r)/2 \rfloor$ 3 **X-SORT**(A, p, q) 4 **X-SORT**(A, $q+1$, r) 5 Fun(A, p, q, r)	**Divide:** get the middle of the subarray, D(n) = Θ(1). **Conquer:** 2 sub-problems of size n/2, so 2T(n/2) **Combine:** assume that Fun is Θ(n), so C(n) = Θ(n).

The recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases} \qquad T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

The master theorem

THEOREM: For recurrence $T(n) = aT(n/b) + f(n)$, calling $n^{\log_b a}$ a **magic formula** or **M.F.**

Then T(n) can be bounded asymptotically as follows:

1. *If* $f(n) < O(M.F.)$, then T(n) = Θ($M.F.$).
2. *If* $f(n) = \Theta(M.F.)$, then $T(n) = \Theta(M.F. \lg n)$.
3. *If* $f(n) > \Omega(M.F.)$, then T(n) = Θ(f(n))

In case 1, **M.F.** is the larger, then T(n) = $\Theta(n^{\log_b a})$.
In case 3, f(n) is the larger, then T(n) = Θ(f(n)).
In case 2, T(n) = Θ($n^{\log_b a} \lg n$) = Θ(f(n) lg n).
The master theorem can not be applied if the function f(n) falls into the gap between:

- cases 1 and 2 when f(n) is smaller than **M.F.** but not polynomially smaller.
- cases 2 and 3 when f(n) is larger than **M.F.** but not polynomially larger.

Example $T(n) = 9T(n/3) + n.$	**Example** $T(n) = T(2n/3) + 1,$
a = 9, b = 3, f(n) = n, M.F.= $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) < O(M.F.)$, we can apply **case 1** T(n) = Θ(n²).	a = 1, b = 3/2, f(n) = 1, M.F.= $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$ case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, $T(n) = \Theta(\lg n)$.

Example $T(n) = 2T(n/2) + n \lg n,$

a = 2,
b = 2,
f(n) = n lg n,
$n^{\log_b a} = n$.

It seems that case 3 can apply,

f(n) = n lg n is asymptotically larger than $n^{\log_b a} = n$. But it is not polynomially larger.

Ratio f(n)/ $n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^{ϵ} for any positive constant ϵ.
So, it falls into the gap between case 2 and case 3.
The master method does not apply !