## Computational Complexity:

The complexity of computational problems can be discussed by choosing a specific abstract machine as a model of computation and considering how much time and/or space machine of that type require for the solution of that problem.

- ♦ A given problem can be solved by using more than one computational model i.e. there may be more than one TM that solve the problem. It is thus necessary to measure the qualities of alternative model to solve the same computational problem.
- ♦ The quality of an computational model is measured usually in terms of the resources needed by the algorithm for its execution.
- ♦ The two important resources used for executing a given algorithm are (i) Time (ii) memory , required to execute that algorithm.
- ♦ When estimating execution time(Time complexity) we are interested in growth rate and not in absolute time.
- ♦ Similarly , we are interested in growth rate of memory need( space complexity) rather than the absolute value of space.
- ♦ So the boundary time and boundary space for executing an algorithm are usually expressed in terms of known mathematical functions.

## Notation for comparing growth rate.

**The Big-Oh notation:** Suppose ***f,g: N→ N*** are functions defined in a finite number of points, we write,

$f(n) = O(g(n))$ or simply ***f= O(g),*** if there are two positive constants c and $n_0$ such that for all $n>n_0$ , $f(n) \leq c\, g(n), n>n_0$
i.e. ***g*** is upper bound of ***f.*** Also **f** is "Big-Oh" of ***g.***

***Example:*** $f(n) = (n+1)^2 + n.$ $g(n) = n^2$ ***then*** $f(n) = O(g(n))$

Proof: We can choose c and $n_0$ to satisfy the requirement
$f(n) \leq c\, g(n)$ , $n> n_0$
i.e. $(n+1)^2 + n \leq c\, n^2$ , $n> n_0$
or $n^2 +3n +1 \leq cn^2$ , $n> n_0$
Choose $n_0 = 1$ and $c = 6$
Then $n^2 +3n +1 \leq 6n^2$ , $n >1$
Which is trivially true. Hence f(n) = O(g(n)).

## Big-Omega (Ω):

A function f(n) is said to be "big-Omega of g(n)" and we write,
$f(n) = \Omega(g(n))$ , if there exist two positive constants c and $n_0$ such that
$f(n) \geq c\, g(n)$ , $n>n_0$ i.e . ***g*** is lower bound of ***f.***

***Example:*** $g(n)= 10n^2+n$ , $f(n) = n^3$ ***then*** $f = \Omega(g)$

Proof: Choose c and $n_0$ to satisfy the relation $f(n) \geq c\, g(n)$ , $n> n_0$ as
$c = 1/20$ and $n_0 = 1$

i.e. $n^3 \geq \frac{1}{20}(10n^2 + n)$ , $n>1$

or $20 n^3 \geq 10 n^2 + n$ , $n>1$
Which is trivially true. Hence $f(n) = \Omega(g(n))$

**Big Theta(Θ):** Let f, g: N → N are functions defined over a finite number of points , we write , $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$. In other words the big-theta can be defined as $f= \Theta(g)$ if $f= O(g)$ and $f = \Omega(g)$

Example: $f(n) = n^3 + 10n^2$, $g(n) = n^3$ then $f = \Theta(g)$

Proof: $n^3 + 10n^2 \le 2*10n^3$ , $n > 1$ Choosing $n_0 = 1$ and $c = 2*10$.

Or $n^3 + 10n^2 \le 10n^3 + 10n^3$ $n > 1$ Which is trivially true

$\therefore$ *f(n) = O(g(n))…………………(1)*

Again, $n^3 \le 1.(n^3 + 10n^2)$ , $n>1$ choosing $c = 1$, $n_0 = 1$

Which is trivially true.

Hence *g(n) = O(f(n)) ……………………(2)*

From (1) and (2) it is concluded that

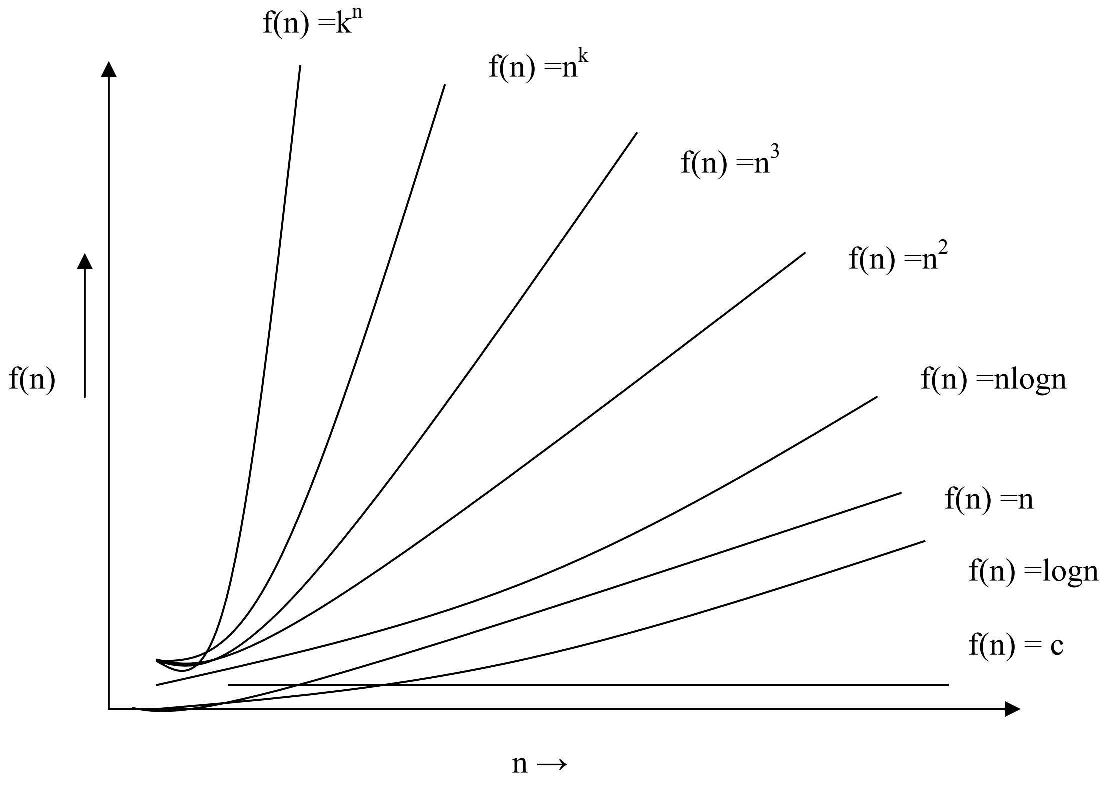
*f(n) = Θ(g(n))*

Some commonly used functions in complexity analysis and their order of complexity

| | | |
|---|---|---|
| $f(n) = c$ | constant | $O(1)$ |
| $f(n) = c\log n$ | logarithmic | $O(\log n)$ |
| $f(n) = cn$ | linear | $O(n)$ |
| $f(n) = cn\log n$ | linearithmic | $O(n\log n)$ |
| $f(n) = cn^2$ | quadratic | $O(n^2)$ |
| $f(n) = cn^3$ | cubic | $O(n^3)$ |
| $f(n) = cn^k$ | k polynomial in n | $O(n^k)$ |
| $f(n) = ck^n$ | exponential in k | $O(n^k)$ |

The order of complexity:

$f(n) = c < \log n < n < n\log n < n^2 < n^3 < \ldots\ldots\ldots\ldots < k^n$



## Time and Space Complexity of a Turing Machine:

The model of computation we have chosen is the TM when a Turing machine answers a specific instance of a decision problem, we can measure time ( the no of moves ) and the space ( no. of tape squares ) required by the computation. The most obvious measure of the size of any instance is the length of the input string. The worst case is considered as the maximum time or space that might be required by any string of that length.

The time and space complexity of a Turing machine ( deterministic ) can be defined as :

Let T be a Turing machine. The time complexity of T is the function $\tau_T$ defined on the natural number as follows. For n ∈ N , $\tau_T$ (n) is the maximum number of moves T can make on any input string of length n . If there is an input string |x| = n so that T loops forever on input x, $\tau_T$ (n) is undefined.

The space complexity function $s_T$ of T is defined as follows.

For some input of length n, causes T to loop forever $s_T$ (n) is undefined.

If no input string of length n causes T to use an infinite no of tape square, $s_T$ (n) is the maximum number of tape squares used by T for any input string of length n.

If T is multi-tape TM, “no of tape squares” means the maximum of the no of individual tapes.

For some input of length n, causes T to loop forever $s_T$ (n) undefined.

**Time and space complexity of NTM :**

Let T be a non-deterministic TM accepting a language time $L \subseteq \Sigma^*$ . For an input x, we define the computation time as follows.

- First $\tau_x$ is undefined if it is possible for T to loop forever on input x.
- If $x \in L$ , $\tau_x$ is the minimum number of move required for T to accept input x.
- If $x \notin L$ , $\tau_x$ is the minimum number of moves required for t to reject x.

The non-deterministic time complexity of T is the function $\tau_T$ (n) is the maximum value of $\tau_x$ over string x with |x| = n. Thus $\tau_T$ (n) is defined unless there is a string of length n on which T loop for ever.

Similarly space $s_x$ is undefined if T loop forever on input x. Otherwise it is the minimum no of tape squares required in accepting or rejecting x.

The non-deterministic space complexity of T is the function $s_T$ (n) is undefined if T can loop forever on some input of length n, and otherwise $s_x$ (n) is the maximum of the numbers $s_x$ for |x| = n.

## Reducibility

Reducibility is a way of converging one problem into another problem in such a way that a solution to the second problem can be used to solve first problem. Such "reducibilities" comes up in our every day life.

For example: Suppose we want to find a way around a new city. We know this would be easy if we have a city map. Thus we can reduce the problem of finding our way around the city to the problem of obtaining a map of the city.

Reduction doesn't mean to make smaller, but it means to transform or convert a problem X to another problem Y that is at least as hard as X. Usually Y is at least as hard as X and so we express a reduction from X to Y as X <= Y.

Reducibility is primary method for proving that a problem is computationally unsolvable. Reducibility says nothing about solving X orY alone only about the solvability of X in the presence of Y.

For example :

- ♦ problem of measuring the area of a rectangle to the problem of increasing its height and width .
- ♦ The problem of solving linear equation reduces to the problem of inverting matrices.

**Intractability:**

Intractability is technique for showing problems not to be solvable in polynomial time.
● The problems that can be solved by any computational model (TM) using no more time then some slowly growing function size of the input are called "tractable ", i.e. those problems solvable within reasonable time and space constraints ( polynomial time )

To introduce intractability theory; the classes $\mathcal{P}$ and $\mathcal{NP}$ of problems solvable in polynomial time by deterministic and non-deterministic TM's are essential. A solvable problem is one that can be solved by a particular algorithm i.e. there is a certain algorithm to solve this problem. But in practice algorithm may require a lot of space and time . When the space and time required for implementing the steps of the particular algorithm are reasonable , we can say that the problem is tractable, that is solvable in practice.

A decision problem is tractable if there is an algorithm to solve the given problem and time required is expressed as a polynomial P(n) , n being the length of input. Problems are intractable if the time required for any of the algorithm is at least f(n) , where f is an exponential function of n.

We know that if a turing machine of any kind , either multiple tape, multi-track etc. halts after polynomial number of steps , then there is an equivalent Turing machine of any other kind which also halts in polynomial number of steps but only polynomials may be different.

**Definition of the Class $\mathcal{P}$:** The class $\mathcal{P}$ is the set of problems that can be solved by a deterministic TM in polynomial time.

A Language L is in class $\mathcal{P}$ if there is some polynomial T(n) such that L=L (M) for some deterministic TM M of time complexity T(n)

Sorting , searching , shortest path problems are examples of problems in $\mathcal{P}$

**Example: Kruskal's Algorithm for MST**

**Idea:** It selects initially n nodes as n-trees i.e a forest with n trees

- It combines two trees by connecting them by a lowest cast edge that does not form cycle.

**Algorithm:**

```
1. T= n nodes.
2. while T contains fewer than n-1 edges and $E \neq \phi$ do
3. { - Chose an edge (v,w) from E of lowest cost
4.   – delete (v,w) from E
5.   – If adding edge (v,w) to T does not from cycle
```

```
        then add edge (v,w) to T
        else discard (v,w).
    }
```

Complexity of kruskal's algorithm is $O(|E| \log |E|)$

Complexity of Dijkstra's shertest path is $O(n^2)$

These problems are in class P

**<u>The class $\mathcal{NP}$:</u>** The class of decision problems for which there is a polynomially bounded non-deterministic algorithm is, called class $\mathcal{NP}$.

We can say, a Language L is in NP if there is a NTM M and a polynomial time complexity T(n) such that L = L (M) and M is given an input of length n .

- Since every deterministic TM is a non-deterministic TM having no choice of moves, $P \subseteq NP$ . But NP contains many problems not in $\mathcal{P}$.

Traveling Salesman problem is class

**<u>TSP:-</u>** Input- A graph G(v,e) having weight in each edge Question asked is, whether the graph has a :Hamilton circuit" of total weight as weight equal to MST of G.

Hamilton circuit is a set of edges that connect the nodes into a single cycle, with each node appearing exactly once.

No of edges on Hamilton circuit must be equal to the no of nodes in the graph.

Although the definition of P and NP are seems similar, but there is a vast difference between them. When L is in P, the number of moves to test whether any string of length n is less then or equal to P (n) where P(n) is a polynomial function of n.

**<u>The Big question</u> : Is $\mathcal{P} = \mathcal{NP}$ ?**

No body has suggested answer to this question. This is an open question for computational complexity research.

<u>$\mathcal{NP}$ – Completess:</u>

Let L be a problem in NP. We say that L is NP-complete if the following statements are true about L:

a. L is in NP
b. For every language $L_1$ in NP there exist a polynomial time reduction of $L_1$ to L:

Once we have some NP-complete problem, we can prove a new problem to be NP-complete by reducing some know NP-complete problem to it, using a polynomial-time reduction.

Now let us discuss some properties of NP-complete problems.

1. No polynomial time algorithm has been found fro any one of them.
2. It is not established that polynomial time algorithm fro these problems do not exist.
3. If a polynomial-time algorithm is found for one of them, there will be polynomial-time algorithm for all of them.
4. If it can be proved that no polynomial time algorithm exists for any one of them, then it will not exist for every one of them.

There are several example of NP-complete problems such as traveling salesman problem, zero-one programming problem, satisfiability problem and vertex-cover problem here we are not discussing detail of these example.

<u>$\mathcal{NP}$ – Complete Problem:</u>

A problem $\Pi$ is $\mathcal{NP}$ – complete if $\Pi \in \mathcal{NP}$ and every other problems in $\mathcal{NP}$ is polynomially reducible to $\Pi$ .

SAT is $\mathcal{NP}$ complete : [Cook's Theorem]

**CNF –Satisfiability :**

- A logical variable is a variable that can take values "true" or "false" [1 or 0]
- A literal is a logical variable or it's negation.
- A clause is a sequence of literals separated by Boolean OR( $\vee$ ) operators. e.g. $(a_1 \vee a_2 \vee a_3)$
- A Conjunctive Normal Form (CNF) is a sequences of clauses separated by AND( $\wedge$ ) operators. e.g. $(a_1 \vee a_2 \vee a_3) \wedge (a_1 \vee a_2$

## The CNF-SAT problem:

***Given : A logical expression E in CNF.***

***Question: Is there a truth assignment to the variables of E that make E true ?***

Example:

$$E_1 = (p \vee q \vee s) \wedge (\neg q \vee r) \wedge (\neg p \vee r) \wedge (\neg r \vee s) \wedge (\neg p \vee \neg s \vee \neg q)$$

if p=1, q=0, r=1,s=1 then

$E_1 = 1 \wedge 1 \wedge 1 \wedge 1 \wedge 1 = 1$ Which is satisfied.

$$E_2 = (a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$$

$E_2$ is not satisfied for any truth assignment of variables.

- No polynomial time algorithm is known to solve SAT problem.

***NP-Hard Problem:*** A problem $\Pi$ is called $\mathcal{NP}$-Hard if every problem in $\mathcal{NP}$ is polynomially reducible to $\Pi$.

Hamiltonian cycle, Graph coloring etc are $\mathcal{NP}$-Hard.