Algorithms

Algorithm is a sequence of unambiguous instructions for solving a problem for obtaining the desided O/P for any legitimate I/P in a finite amount of time.

- unambiguous
- desired O/P
- legitimate I/P
- finite time

problem

↓

algorithm

↓

I/P → $\boxed{\text{computer}}$ → O/P

Properties

- No ambiguity in the instructions.
- Range of I/P has to be specified carefully
- Same algorithm can be expressed in different ways.
- Several algorithms for solving same problem might exist.
- Algorithms for a problem can be based on different ideas and can solve the problem with dramatically different speeds.

Algorithm to find GCD of two numbers:

20, 30

$30 = 20 \times 1 + 10$

$20 = \boxed{10} \times 2 + 0$

Find GCD (123, 36)

$123 = 36 \times 3 + 15$

$36 = 15 \times 2 + 6$

$15 = 6 \times 2 + 3$

$6 = \boxed{3} \times 2 + 0$

Euclid's Algorithm for computing gcd (m, n)

Step 1: If n = = 0, return the value of m as answer and stop; otherwise proceed to step 2

Step 2: Divide m by n and assign the value of remainder to r.

Step 3: Assign the value of n to m and r to n. Go to step 1.

Pseudocode:

```
Euclid (m, n):
// Input: two non-negetive, not both zero integers
          m & n
// output: Greatest common divisor of m & n

while n ≠ 0
    r = m mod n
    m = n
    n = r
return m
```

Example:

$m = 30 \qquad n = 35$

i) $r = 30 \% 35 = 30$

$m = 35$

$n = 30$

ii) $r = 35 \% 30 = 5 \qquad [\because 30 != 0]$

$m = 30$

$n = 5$

iii) $r = 30 \% 5 = 0 \qquad [\because 5 != 0]$

$\boxed{m = 5}$

$n = 0$

Consecutive Integer Checking Algorithm

Step 1: Assign the value of min{m, n} to t

Step 2: Divide m by t. If the remainder of this division is zero, Go to step 3 otherwise go to step 4.

Step 3: Divide n by t. If the remainder of this division is zero, return the value of t as the answer and stop. Otherwise go to step 4.

Step 4: Decrement t by 1 and Go to step 2.

Find GCD(4, 6):

1) $t = \min(4, 6) = 4$

i) $\frac{4}{4} = 4 \% 4 = 0$

$6 \% 4 = 2 \; != 0$

$t = t - 1 = 3$

ii) $\frac{4}{3}$ $4 \% 3 = 1 \; != 0$

$t = t - 1 = 2$

iii) $4 \% 2 = 0$

$6 \% 2 = 0$

Hence GCD(4, 6) = 2.

Middle School Procedure:

Step 1: Find the prime factors of m.

Step 2: Find the prime factors of n.

Step 3: Identify all the common factors in the two prime expansions found in Step 1 and Step 2.

Step 4: Compute the product of all the common factors and return it as GCD of the numbers given.

Compute GCD (60, 24):

$$\begin{array}{r|l} 2 & 60 \\ 2 & 30 \\ 3 & 15 \\ 5 & 5 \\ & 1 \end{array}$$

$60 = 2 \times 2 \times 3 \times 5$

$24 = 2 \times 2 \times 2 \times 3$

$$\begin{array}{r|l} 2 & 24 \\ 2 & 12 \\ 2 & 6 \\ 3 & 3 \\ & 1 \end{array}$$

$GCD(60, 24) = 2 \times 2 \times 3$

$= \underline{\underline{12}}$

Sieve of Erathosthenes:

2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21
22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41
42	43	44	45	46	47	48	49	50	

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37

41, 43, 47

Algorithm Sieve(n):
// Input : An integer $n \geq 2$
// output : Array L of all prime numbers less than or equal to n.

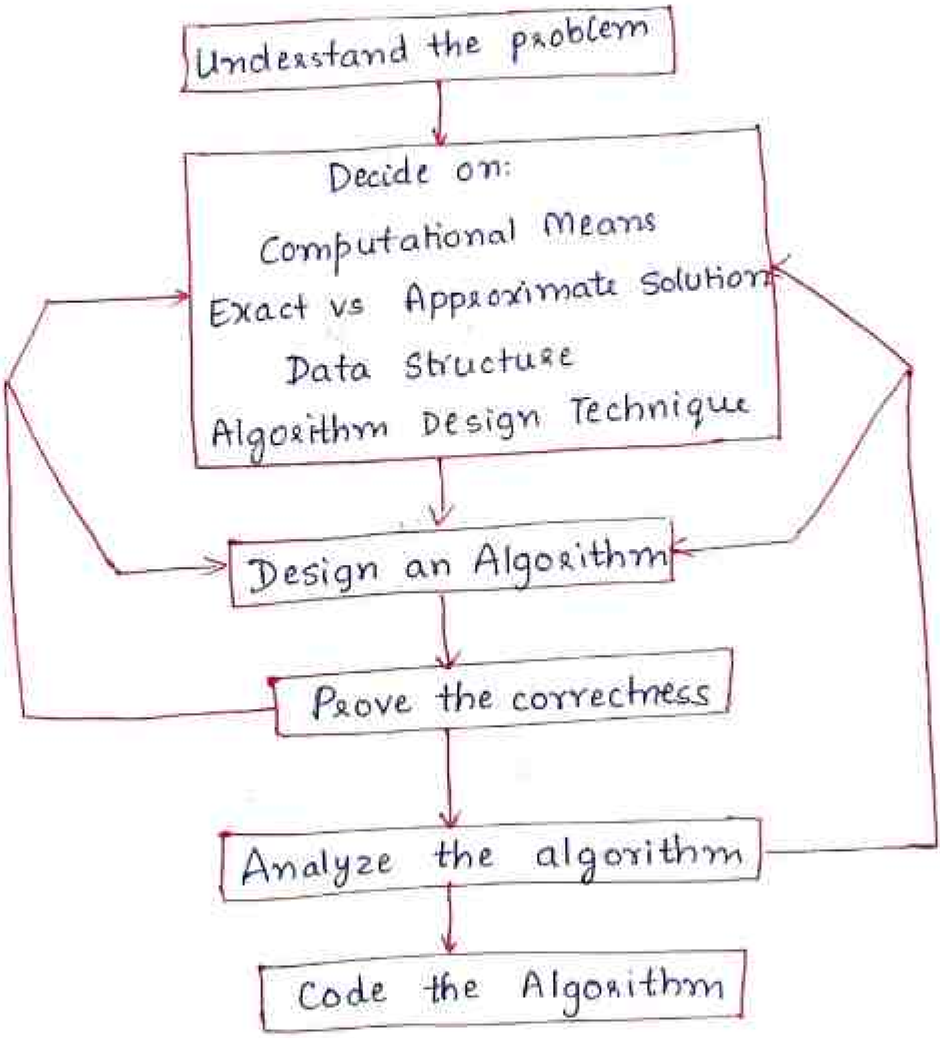
```
for p ← 2 to n do
    A[p] ← p
for p ← 2 to ⌊√n⌋ do
    if A[p] ≠ 0
        j ← p * p
        while j <= n do
```

```
        A[j] ← 0
        j ← j+p

i ← 0
for p ← 2 to n do
    if A[p] ! = 0
        L[i] ← A[p]
        i ← i+1

return L
```

Fundamentals of Algorithm Problem Solving:



<u>Understand the problem:</u>

- ask doubts to clarify
- clearly define range of I/P
- An algorithm must work correctly for all inputs and not just most of the I/Ps.

Decideon:

- Computational means
  - sequential algorithms
  - parallel algorithms

- Exact vs Approximate solutions
  - ↓ searching, sorting
  - ↳ square roots, integrals

- Data structure - Appropriate Data structure

  Algorithms + Data structure = Program

- Algorithm Design Technique
  - Brute Force
  - Divide and Conquer
  - Decrease and Conquer
  - Transform and conquer ...

Design an Algorithm:

- In terms of natural language statements, pseudocode or flowchart.

Prove the correctness:

Proving the algorithm's wrongness can be done just by using one invalid input but proving the correctness should be done to show that the algorithm works for all inputs in decided range. (Mathematical Induction)

Analyze the Algorithm:

↳ In terms of efficiency
  - Time
  - Space

Time - time taken for algorithm to complete execution

space - extra memory required for running algorithm.

Code the algorithm:

Develop the code for the algorithm using any specified programming language.

Important Problem Types:

- Sorting
  - Stable
  - in place
- Searching
- String Problems
- Graph Problems
- Combinatorial problems
- Geometrical problems
- Numerical Problems

Fundamental Data Structures:

- Linear Data Structure
  - Arrays
  - Linked Lists
  - Stack
  - Queue
- Graphs
  - Directed and undirected graph.
  - Paths and cycles
  - Connectivity
  - Representation
    - adjacency matrix
    - adjacency list
    - weighted matrix
- Trees
  - Rooted Trees
  - Free Trees
  - Forest
  - Ordered Trees - Binary Search Trees
- Sets & Dictionaries.
  - multiset or bag
  - key value pairs.

Abstract Data Type (ADT). - its an abstraction of data items with associated operations.

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

Analysis Framework:

Time Efficiency
- Speed of computer
- size of I/P
- choice of algorithm
- Type of compiler being used.

Space Efficiency
- Program space
- Data space
- Stack space

Measuring of I/P size:

Searching, Sorting $\longrightarrow$ 'n'

Polynomial Function $\longrightarrow$ 'highest order'

matrix $\rightarrow$ 'No. of elements'

String $\longrightarrow$ string length or word count.

Numbers $\rightarrow$ No. of bits or digits in number

Units of Measuring Running Time:

- Basic operation - most Important operation in the algorithm.

Searching algorithm - 'comparison'

Matrix multiplication - 'multiplication'

Let C_{op} - time for execution of basic operation on some computer.

$C(n)$ - no. of times basic operation is executed

$$T(n) \approx C_{op} \cdot C(n)$$

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

$$= \frac{1}{2}n^2 - \frac{1}{2}n$$

$$\approx \frac{1}{2}n^2$$

$$T(n) \approx c_{op} \cdot \frac{1}{2} n^2$$

$$\frac{T(2n)}{T(n)} \approx \frac{\cancel{c_{op}} \times \cancel{\frac{1}{2}} (2n)^2}{\cancel{c_{op}} \times \cancel{\frac{1}{2}} (n^2)}$$

$$\approx \frac{4\cancel{n^2}}{\cancel{n^2}}$$

$$\approx 4$$

Orders of Growth:

$$1 < \log_2 n < n < n\log_2 n < n^2 < n^3 < 2^n < n!$$

Sequential Search(A [0 .. n-1] , K)

```
i <- 0
while i < n and A[i] ≠ k do
      i <- i + 1
if i < n return i
else return -1
```

Best Case:

$$C_{best}(n) = 1$$

Worst case

$$C_{worst}(n) = n$$

Average case:

$$C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + n \cdot \frac{p}{n}\right] + n(1-p)$$

$$= \frac{p}{n} [1 + 2 + 3 + \cdots + n] + n(1-p)$$

$$= \frac{p}{\cancel{n}} \times \frac{\cancel{n}(n+1)}{2} + n(1-p)$$

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

If $p = 1$, then $C_{avg}(n) = \frac{n+1}{2}$

If $p = 0$, then $C_{avg}(n) = n$

Order of growth:

n (Input size)	n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	10	3.3	33	10^2	10^3	2^{10}	10!
10^2	10^2	6.6	660	10^4	10^6	2^{100}	100!
10^3	10^3	9.9	9900				

Slowest growing (above) - $\log_2 n$

Order:

$$1 < \log_2 n < n < n \log_2 n < n^2 < n^3 < 2^n < n!$$

constant (1), logarithmic ($\log_2 n$), linear (n), linearithmic ($n \log_2 n$), quadratic (n^2), cubic (n^3), Exponential (2^n), Factorial ($n!$)

Asymptotic Notations:

Informal:

$O(g(n))$ is set of all function with smaller or same order of growth as $g(n)$.

$$n \in O(n^2) \qquad n^3 \notin O(g(n))$$

$$\frac{n^2}{2} \in O(n^2)$$

$\Omega(g(n))$ is set of all functions that have larger or same order of growth as $g(n)$

$$n^3 \in \Omega(n^2) \qquad n \notin \Omega(n^2)$$

$$\frac{1}{2}n^2 - n \in \Omega(n^2)$$

$\Theta(g(n))$ is a set of all functions that have the same order of growth as $g(n)$

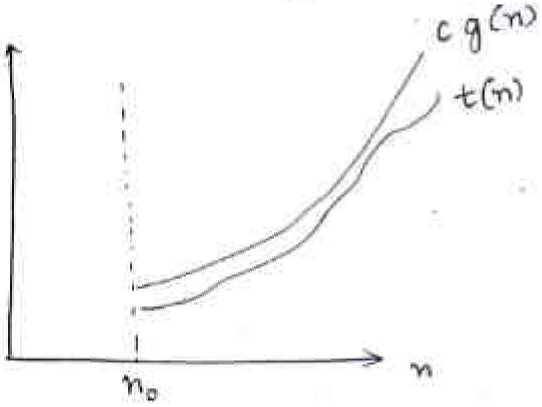
$$an^2 + bn + c \in \Theta(n^2)$$

$$n^2 + \log(n) \in \Theta(n^2)$$

O - Notation:

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

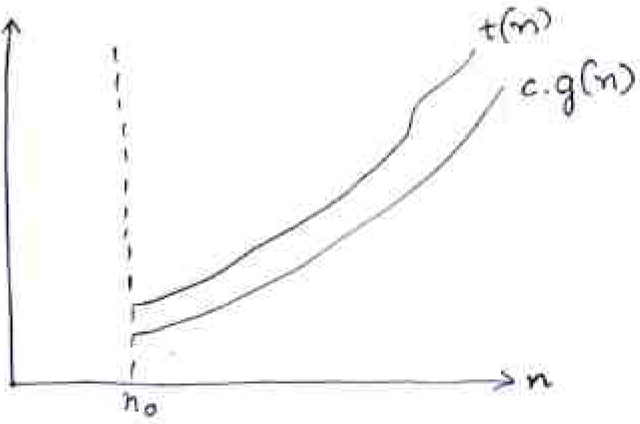
$$t(n) \leq c\,g(n) \quad \text{for all } n \geq n_0$$



Ω - Notation:

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

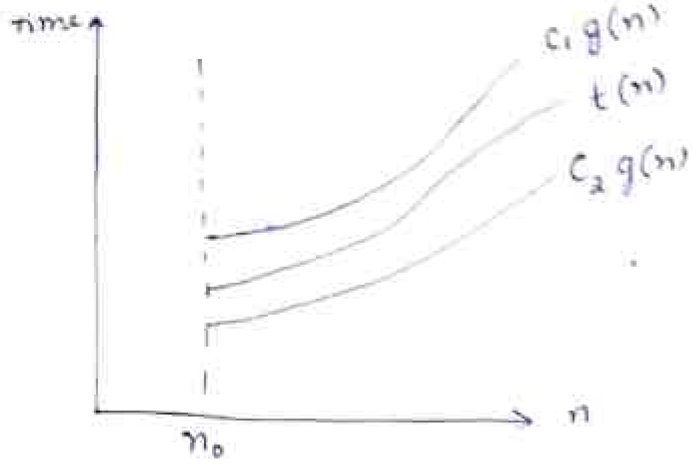
$$t(n) \geq c\,g(n) \quad \text{for all } n \geq n_0$$



Θ Notation:

A function $t(n)$ is said to be in $\Theta(g(n))$ denoted by $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiple of $g(n)$ for all large n, i.e if there exists same positive constant c_1 and c_2 and some non-negative integer n_0 such that

$$c_2\, g(n) \le t(n) \le c_1\, g(n) \quad \text{for all } n \ge n_0$$



Problems

1) Express $f(n) = 100n + 5$ in terms of Big Oh, Big Omega and Big Theta.

Big Oh:

$$f(n) = 100n + 5$$

$$f(n) \le c \cdot g(n), \quad \forall\, n \ge n_0$$

$$c \cdot g(n) = 100n + n$$

$$= 101n$$

$$f(n) \le c \cdot g(n)$$

$$100n + 5 \le 101n \qquad \forall\, n \ge 5$$

$c = 101$

$n_0 = 5$

$g(n) = n$

Big Omega

$$f(n) = 100n + 5$$

$$f(n) \ge c \cdot g(n)$$

$$100n + 5 \ge 100n \qquad n \ge 0$$

$c = 100$ $\quad n_0 = 0$

$g(n) = n$

Big Theta:

$$c_2\, g(n) \le f(n) \le c_1 \cdot g(n) \quad \forall n \ge n_0$$

$$100n \le 100n+5 \le 101n \quad , n \ge 5$$

$c_2 = 100$

$c_1 = 101$

$g(n) = n$

$n_0 = 5$

2) $f(n) = 6 \cdot 2^n + n^2$

<u>Big Oh:</u>

$$f(n) \le c \cdot g(n)$$

$$6 \cdot 2^n + n^2 \le 7 \cdot 2^n \quad , \quad n \ge 0$$

$n_0 = 0$, $c = 7$, $g(n) = 2^n$

<u>Big Omega:</u>

$$f(n) \ge c \cdot g(n)$$

$$6 \cdot 2^n + n^2 \ge 6 \cdot 2^n \quad n \ge 0$$

$n_0 = 0$, $g(n) = 2^n$, $c = 6$

<u>Big Theta:</u>

$$c_2\, g(n) \le f(n) \le c_1\, g(n)$$

$$6 \cdot 2^n \le 6 \cdot 2^n + n^2 \le 7 \cdot 2^n \quad , n \ge 0$$

$c_2 = 6$, $c_1 = 7$, $g(n) = 2^n$, $n_0 = 0$

<u>HW:</u>

$f(n) = 10n^3 + 5$

<u>Oh:</u> $c = 11$, $g(n) = n^3$, $n_0 = 2$

<u>omega</u>: $c = 10$, $g(n) = n^3$, $n_0 = 0$

Theta: $c_2 = 10$, $c_1 = 11$, $g(n) = n^3$. $n \ge 2$

$n_0 = 2$

* Useful property involving asymptotic notation:

Theorem:

If $f_1(n) \in O(g_1(n))$

and $f_2(n) \in O(g_2(n))$

Then $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

Proof:

Let a_1, b_1, a_2 and b_2 be four arbitrary real numbers such that $a_1 \leq b_1$ and $a_2 \leq b_2$

$$a_1 + a_2 \leq b_1 + b_2$$

$$a_1 + a_2 \leq 2 \max\{b_1, b_2\}$$

$$f_1(n) \leq c_1 g_1(n) \quad \text{since } f_1(n) \in O(g_1(n))$$

$$\forall n \geq n_1$$

Since $f_2(n) \in O(g_2(n))$,

$$f_2(n) \leq c_2 g_2(n) \quad \forall n \geq n_2$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$ so that we can compare both inequalities. Adding both inequalities,

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) \\ &\leq c_3 (g_1(n) + g_2(n)) \\ &\leq 2c_3 \max\{g_1(n), g_2(n)\} \end{aligned}$$

$$\Rightarrow f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

where $c = 2c_3 = 2\max\{c_1, c_2\}$

and $n \geq \max\{n_1, n_2\}$

⇒ Hence the proof.

Best Case - Ω Average case - Θ

Worst Case - O

Basic Efficiency classes:

1 - Constant , n - linear, $\log_2 n$ - logarithmic
$n\log_2 n$ - linearithmic, n^2 - Quadratic, n^3 - Cubic,
2^n - Exponential, $n!$ - Factorial

Mathematical Analysis of Non-Recursive Algorithms

ALGORITHM MaxElement(A[0 .. n-1])
// Input: Array of n elements
// Output: The value of largest element in the array.

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

→ Measure of I/P size - n
→ Basic operation - comparison
→ Best case, worst case and average case is same

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

Plan for analysing Time Efficiency of Non-Recursive Algorithms:

1. Decide on parameters indicating I/P size.
2. Identify the algorithm's basic operation
3. Check if the no. of times the basic operation is executed depends only on size of input. If it depends on additional property, best case, worst case and average case have to be investigated.
4. Set up the sum indicating the no. of times the basic operation is executed.

5. Use standard formulas of sum manipulation to established order of growth

Sum manipulation Formulae:

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i$$

$$\sum_{i=l}^{u} a_i \pm b_i = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$$

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u$$

$$\sum_{i=0}^{n} i = 1+2+3+\dots+n = \frac{n(n+1)}{2}$$

$$\approx \frac{n^2}{2} = \Theta(n^2)$$

Element Uniqueness Problem.

ALGORITHM UniqueElements (A[0 .. n-1])

// Input : An array of n elements

// Output : Returns true if all elements in A

// are distinct and false if all elements are not

// distinct

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

- Measure of I/P size : n
- Basic Operation : Comparison
- Worst case

when last two elements are only pair of equal elements

when no elements are equal

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

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

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

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

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

$$= n[n-2-0+1] - \frac{(n-2)(n-1)}{2} - (n-1)$$

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

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

$$= (n-1)\left[(n-1) - \frac{(n-2)}{2}\right]$$

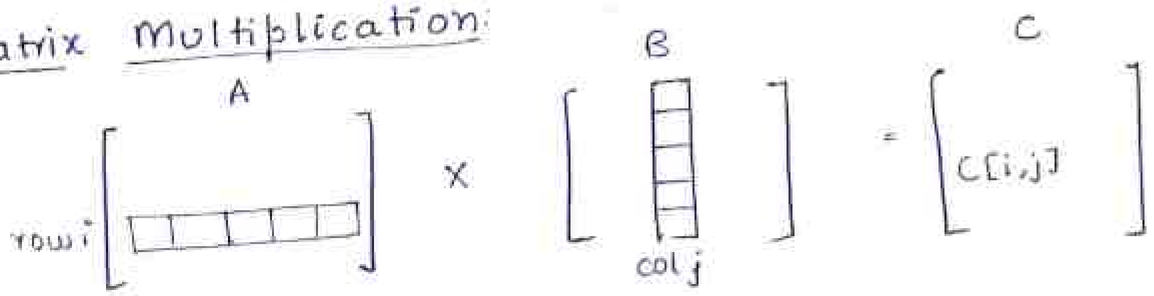
$$= (n-1)\left(\frac{2n-\cancel{2}-n+\cancel{2}}{2}\right)$$

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

$$\in O(n^2)$$

$$C_{best}(n) = 1 \in \Omega(1)$$

<u>Matrix Multiplication</u>:



$$c[i,j] = A[i,0] * B[0,j] + A[i,1] * B[1,j] + A[i,2] * B[2,j] + \ldots + A[i,n-1] * B[n-1,j]$$

Algorithm Matrix Multiplication (A[0..n-1], B[0..n-1]).

// Input two n by n matrices

// Matrix Product C = AB

for $i \leftarrow 0$ to $n-1$ do

for $j \leftarrow 0$ to $n-1$ do

$c[i,j] \leftarrow 0.0$

for $k \leftarrow 0$ to $n-1$ do

$C[i,j] \leftarrow A[i,k] * B[k,j] + C[i,j]$

return C

- Size of Input - $n \times n$
- Basic operation $C[i,j] \leftarrow C[i,j] + A[i,k] \times B[k,j]$

Hence multiplication is basic.

No. of multiplications.

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

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

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

$$= \sum_{i=0}^{n-1} \times n \sum_{j=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} \times n (n-\cancel{1}-0+\cancel{1})$$

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

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

$$= n^2 (n-\cancel{1}-0+\cancel{1})$$

$$= n^3$$

$f(n) \in \Theta(n^3)$

ALGORITHM Binary(n)

// Input: Positive decimal integer n

// output: The no. of binary digits in n's representation

// in binary

```
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

- Basic operation - while condition checking or no. of additions

$$f(n) \in \Theta(\log_2 n)$$

Mathematical Analysis of Recursive Algorithms:

```
Algorithm Factorial(n)
// computes n!
// I/P: non-negative integer n
// O/P: The value of n!
if n = 0 return 1
else return n * factorial(n-1)
```

$$F(n) = n * F(n-1), \quad n > 0$$

Let $M(n)$ be the no. of multiplications required to compute $n!$. Then

$$M(n) = \begin{cases} 0 & n = 0 \\ \underbrace{1}_{\text{To multiply } n * F(n-1)} + \underbrace{M(n-1)}_{\text{to compute } F(n-1)} & n > 0 \end{cases}$$

$$\left.\begin{aligned} 5! &= 5 \times 4! \\ 4! &= 4 \times 3! \\ 3! &= 3 \times 2! \\ 2! &= 2 \times 1! \\ 1! &= 1 \times 0! \end{aligned}\right] \quad 5 \text{ multiplications}$$

$$0! = 1$$

$$M(n) = 1 + M(n-1)$$
$$= 1 + (1 + M(n-2))$$
$$= 2 + M(n-2)$$
$$= 3 + M(n-3)$$
$$\vdots$$
$$= i + M(n-i)$$

Let $n - i = 0 \Rightarrow i = n$

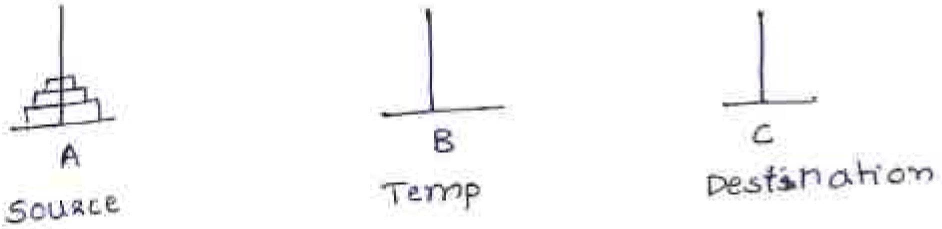
$$M(n) = n + M(n-n)$$
$$= n + M(0)$$
$$= n + 0$$
$$\boxed{M(n) = n}$$
$$M(n) \in \Theta(n)$$

General Plan for Analyzing Time Efficiency of Recursive Algorithms.

1) Decide on parameter indicating input size.
2) Identify the algorithm's basic operation.
3) Check whether the no. of times basic operation is executed varies based on different inputs of same size; if so, worst case, best case and average case efficiency must be investigated separately.
4) Set up a recurrence relation, with an appropriat initial condition, for no. of times basic operation is executed.
5) Solve the recurrence relation or atleast ascertain order of growth of its solution.

Tower of Hanoi:

- Problem of moving n disks of different sizes using 3 rods from one rod to another such that, at a time only one disk is moved, and only upper disk can be moved and no disk can be placed on smaller disk.



Solution:

1) Recursively move n-1 disks from source to Temp using Destination as auxiliary.
2) Move largest disk from source to destination rod.
3) Recursively move n-1 disks from temp rod to destination rod using source rod as auxiliary.

Algorithm Tower Of Hanoi(n, S, D, T)

```
if n = 0
    return
if n = 1
    print Move Disk 1 from S to D
    return
Tower Of Hanoi( n-1, S, T, D)
print Move Disk n from S to D
Tower Of Hanoi( n-1, T, D, S)
```

Recurrence Relation:

$$M(n) = \begin{cases} 1 & n = 1 \\ 2M(n-1) + 1 & n > 1 \end{cases}$$

Solution by backward substitution:

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

$$M(n) = 2^i M(n-i) + \underbrace{2^{i-1} + 2^{i-2} + \dots + 2^2 + 2 + 1}_{\text{In GP}}$$

GP n^{th} term sum $= \dfrac{a(r^n - 1)}{r-1}$

$$1 + 2 + 2^2 + \dots + 2^{i-1} = \frac{1(2^i - 1)}{2-1} = 2^i - 1$$

$$M(n) = 2^i M(n-i) + (2^i - 1)$$

Substitute $n - i = 1 \Rightarrow i = n-1$

$$M(n) = 2^{n-1} M(1) + 2^{n-1} - 1$$

$$= 2^{n-1}(1) + 2^{n-1} - 1$$

$$= 2^{n-1} \cdot 2 - 1$$

$$= 2^n - 1$$

$$\boxed{M(n) \in \Theta(2^n)}$$

Fibonacci Series:

0, 1, 1, 2, 3, 5, 8, ...

Simple recurrence Relation.

$$F(n) = F(n-1) + F(n-2), \quad n > 1$$

$$F(0) = 0, \quad F(1) = 1$$

```
Algorithm Fib (n)
// Input : non-negetive number n
// O/P : The nth Fibonacci number
    if n <= 1
        return n
    else return Fib(n-1) + Fib(n-2)
```

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & , n > 1 \end{cases}$$

$$F(n) = F(n-1) + F(n-2)$$

Backward substitution fails

using solution for linear recurrence relations of second order.

$$a(x(n)) + b\,x(n-1) + c\,x(n-2) = 0$$

$$\Downarrow$$

$$ar^2 + br + c = 0 \rightarrow \text{characteristic equation}$$

$$F(n) - F(n-1) - F(n-2) = 0$$

can be written as,

$$x^2 - x - 1 = 0$$

Roots of the above equation are:

$$x = \frac{+1 \pm \sqrt{(-1)^2 - 4 \times 1 \times -1}}{2(1)}$$

$$= \frac{1 \pm \sqrt{1+4}}{2} = \frac{1 \pm \sqrt{5}}{2}$$

$$x_1 = \frac{1+\sqrt{5}}{2}, \quad x_2 = \frac{1-\sqrt{5}}{2}$$

The solution can be represented in the form of,

$$F(n) = \alpha (x_1)^n + \beta (x_2)^n$$

$$F(n) = \alpha \left(\frac{1+\sqrt{5}}{2}\right)^n + \beta \left(\frac{1-\sqrt{5}}{2}\right)^n$$

To find value of α & β. we use base condition $F(0)$ & $F(1)$.

$$F(0) = 0$$

$$\alpha \left(\frac{1+\sqrt{5}}{2}\right)^0 + \beta \left(\frac{1-\sqrt{5}}{2}\right)^0 = 0$$

$$\alpha + \beta = 0 \Rightarrow \alpha = -\beta$$

$$F(1) = 1$$

$$\alpha \left(\frac{1+\sqrt{5}}{2}\right) + \beta \left(\frac{1-\sqrt{5}}{2}\right) = 1$$

$$\alpha \left(\frac{1+\sqrt{5}}{2}\right) - \alpha \left(\frac{1-\sqrt{5}}{2}\right) = 1$$

$$\frac{\cancel{2}\sqrt{5}}{\cancel{2}}\,\alpha = 1 \Rightarrow \alpha = \frac{1}{\sqrt{5}}$$

$\therefore \quad \beta = -\alpha = -\frac{1}{\sqrt{5}}$

$$F(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$$= \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$$

where $\phi = \frac{1+\sqrt{5}}{2}$, $\hat{\phi} = \frac{1-\sqrt{5}}{2}$

$$\therefore \boxed{F(n) \in \theta(\phi^n)}$$

Fibonacci Addition Reccurrence Relation:

$$A(n) = \begin{cases} 0 & n = 0 \\ 0 & n = 1 \\ A(n-1) + A(n-2) + 1 & n > 1 \end{cases}$$

$$A(n) = A(n-1) + A(n-2) + 1$$

$$A(n) - A(n-1) - A(n-2) = 1$$

$$\Downarrow$$

$$[A(n) + 1] - [A(n-1) + 1] - [A(n-2) + 1] = 0$$

Let $B(n) = A(n) + 1$

$B(n-1) = A(n-1) + 1$

$B(n-2) = A(n-2) + 1$

$$B(n) - B(n-1) - B(n-2) = 0$$

$B(0) = 1$

$B(1) = 1$

$B(n) = F(n+1)$

$\Rightarrow \quad A(n) = B(n) - 1$

$A(n) = F(n+1) - 1$

$= \frac{1}{\sqrt{5}}\left(\phi^{n+1} - \hat{\phi}^{n+1}\right) - 1$

$\Rightarrow A(n) \in \Theta(\phi^{n})$

Algorithm BinRec (n)

// I/P: Positive Decimal Integer n

// O/P: Number of digits in binary representation of n

```
if  n = 1
    return 1
else return 1 + BinRec(⌊n/2⌋)
```

Recurrence Relation:

$$F(n) = \begin{cases} 0 & n = 1 \\ 1 + F(n/2) & , n > 1 \end{cases}$$

$$\begin{aligned} F(n) &= 1 + F(n/2) \\ &= 1 + (1 + F(n/4)) \\ &= 2 + F(n/2^2) \\ &= 2 + (F(n/2^3) + 1) \\ &= 3 + F(n/2^3) \\ &\vdots \\ &= i + F(n/2^i) \end{aligned}$$

Let $\frac{n}{2^i} = 1 \Rightarrow n = 2^i$ or $i = \log_2 n$

$$F(n) = \log_2 n + F(1)$$

$$= \log_2 n + 0 = \log_2 n$$

$$\boxed{F(n) \in \Theta(\log_2 n)}$$

Fibonacci - Iterative Fib(n):

```
F[0] ← 0;  F[1] ← 1
for i ← 2 to n do
    F[i] ← F[i-1] + F[i-2]
return F[n]
```

Homework:

```
ALGORITHM Sum(n)
// Input: A non-negative integer n
S ← 0
for i ← 1 to n do
    S ← S + i
return S
```

a) What does this algorithm compute?

b) What is the basic operation?

c) How many times is the basic operation executed?

d) What is the efficiency class of this algorithm