A. M. PRASAD
Assistant Professor
Department of Computer Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

CHAPTER II

FUNDAMENTALS OF ANALYSIS OF ALGORITHM EFFICIENCY

Topics:

* Analysis of frame work
  1. Measuring the Input size
  2. Units of measuring running time
  3. Orders of Growth
  4. Best, Worst, Average case Efficiencies
* Asymptotic Notations & Basic Efficiency classes
  1. Ω-notation
  2. O-notation
  3. Θ-notation.
* Mathematical analysis of nonrecursive Algorithm
* Mathematical Analysis of recursive Algorithm

I. Analysis of framework

The Efficiency of any algorithm mainly depends upon two factors: (1) Space efficiency and (2) Time Efficiency

The space efficiency of an algorithm is the ammount of memory space required to execute the progr-am.

The time efficiency of an algorithm is how fast a given algorithm is executed. The time efficiency dependsupon choice of algorithm, Number of i/ps etc

Measuring Input size:

In general, the parameter 'n' indicates number of inputs or size of input. With obvious obser-vation that almost all algorithms run longer on larger inputs. for example, it takes longer to sort larger arrays and long time to search larger arrays. & multiply larger matrices. So the time efficiency

of an algorithm depends upon size of input n. ing

Therefore time efficiency is always expressed in terms of 'n'.

A. M. PRASAD
Assistant Professor
Department of Computer Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

UNITS FOR MEASURING RUNNING TIME

To find time efficiency of an algorithm is to identify the most important operation of the algorithm, called basic operation. The basic operation contributes most to the total running time. To find time efficiency, it is required to compute the number of times the basic operation is executed. The time efficiency can be calculated as given.

Let c be time of execution of basic operation

Let C(n) be total number of times basic operation is executed. The running time T(n) is given by $T(n) = c \cdot C(n)$

ORDER OF GROWTH

Some algorithms works faster for small values of n. But as the value of n increases they work very slow. So the behavior of the algorithm changes as n increases. This change in behavior is called "Order of Growth". The order of growth of the common computing time functions given below

N	$\log N$	N	$N \log N$	N^2	N^3	2^N	N!
1	0	1	0	1	1	2	1
2	1	2	2	4	8	4	2
4	2	4	8	16	64	16	24
8	3	8	24	64	512	256	40320
16	4	16	64	256	4096	65536	high
32	5	32	160	1024	32768	4294967 -296	very high

Worst-Case, Best-Case, Average-Case Efficiencies

The worst-case, Best-case and Average-case efficiencies can be explained easily by any Search algorithm.

* if the item to be searched found in first comparision then it is best case
* if the item to be searched present in last or 'nth' location, then it is worst case
* if the item is present in anywhere in the list, then it is required to take average number of cases, so it is average case.

A. M. PRASAD
Assistant Professor
Department of Computer Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

The worst case efficiency can be defined as

" for input of size n for which the algorithm takes longest time to execute among all possible inputs "

The best case efficiency can be defined as

" for input of size n for which the algorithm

takes shortest or least time to execute among all possible Inputs".

II. ASYMPTOTIC NOTATIONS

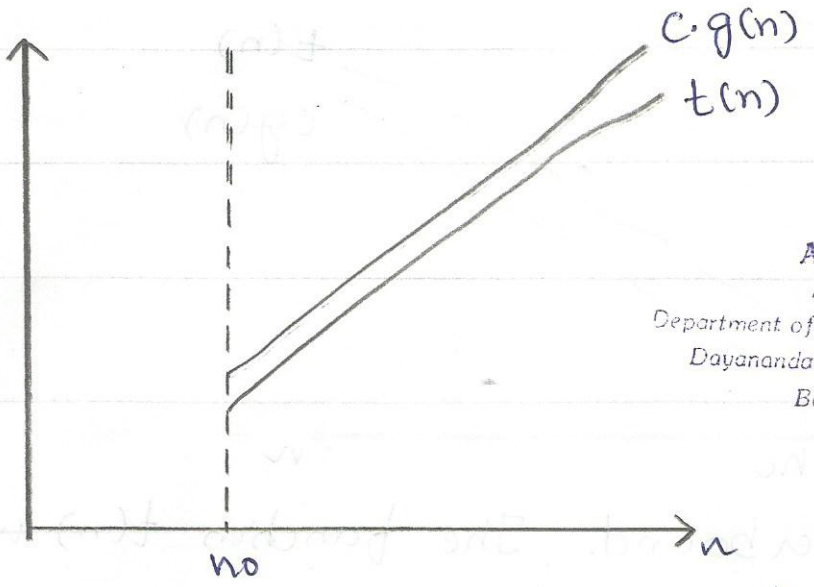
The time Efficiency of an algorithm can be represented using Asymptotic notations. So Asymptotic notations are used to represent time efficiency of algorithm. The different notations are

(1) O - notation
(2) Ω - notation
(3) Θ - notation.

1. O - notation: Big O-notation is used to represent upperbound of an algorithm. It is measure of longest ammount of time it could possibly take for algorithm to complete.

Definition: Let $t(n)$ be the time efficiency of an algorithm, A function $t(n)$ is said to be in $O(g(n))$, if $t(n)$ is bounded above some constant multiple of $g(n)$ for all large n such that

$t(n) \leq c \cdot g(n)$ where c, n_0 are constants



A. N. PRASAD
Assistant Professor
Department of Computer Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

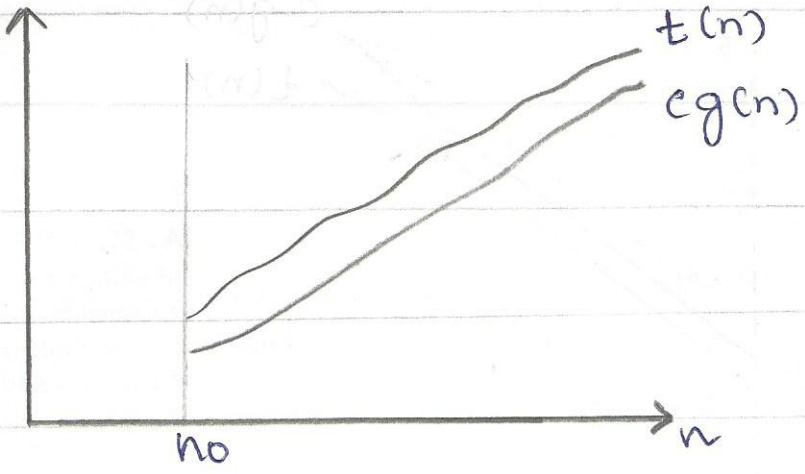
C·g(n) is upper bound. The function t(n) will not consumes more than the specified time C·g(n).

2. Ω - Notation

Ω - notation is used to represent lower bound of an algorithm. It is measure of least ammount of time it could possibly take for algorithm to complete

Definition: Let t(n) be the time efficiency of an algorithm. A function t(n) is said to be in Ω(g(n)) if t(n) is below some constant multiples of g(n) for all large n such that

$$t(n) \geq C \cdot g(n) \quad \text{for all } n \geq n_0$$

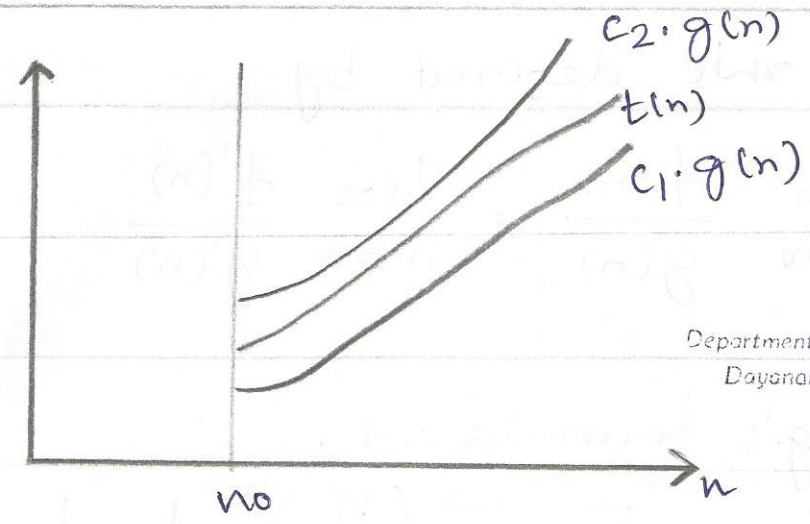


c·g(n) is lowerbound. The function t(n) will consume at least specified time c·g(n)

3. θ-NOTATION: θ-notation is used to denote both lower bound and upperbound on a function t(n). The upperbound on t(n) indicates that function t(n) will not consume more than specified time ie $c_2 * g(n)$. The lower bound on t(n) indicates that function t(n) in best case will consume atleast specified time ie $c_1 * g(n)$

Definition: A function t(n) is said to be in θ(g(n)) denoted by $t(n) \in \theta(g(n))$, if t(n) is bounded above and below by some constant multiples of g(n) for all large n such that

$$c_1 \cdot g(n) \leq t(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0$$



A. M. PRASAD
Assistant Professor
Department of Computer Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

$c_2 * g(n)$ and $c_1 * g(n)$ are upper and lower bound.

ORDER OF GROWTH USING LIMITS

Order of growth of an algorithm can be found or derived using <1> Asymptotic notations. and

<2> By computing limits of two functions

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \begin{cases} 0 ; & f(n) \text{ is smaller order of growth compared to } g(n) \\ c ; & f(n) \text{ --"-- same --"-- --"-- --"--} \\ \infty ; & f(n) \text{ is larger --"-- --"-- --"-- --"--} \end{cases}$$

The first two O and c are like O-notation $f(n) \leq g(n)$

The second c is like Θ-notation, $c_1 g(n) \leq f(n) \leq c_2 g(n)$

The last two c, ∞ are like Ω-notation. $f(n) \geq c \cdot g(n)$

To compute order of growth, the limit based approach L. Hospital rule and sterling's formula are used.

L. Hospital rule defined by

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

and Sterling's formula is

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \text{ for large values of } n$$

Eg1: Compare the order of growth $\frac{1}{2}n(n-1)$ and n^2

Soln: Here $f(n) = \frac{1}{2}n(n-1)$ and $g(n) = n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{1/2\, n(n-1)}{n^2}$$

$$= \frac{1}{2} \lim_{n\to\infty} \left[\frac{n^2 - n}{n^2}\right]$$

$$= \frac{1}{2} \lim_{n\to\infty} \left[1 - \frac{1}{n}\right]$$

$$= \frac{1}{2}\left[1 - \frac{1}{\infty}\right]$$

$$= \frac{1}{2}$$

Since limit has a positive constant both functions $f(n)$ and $g(n)$ have same order of growth $1/2\, n(n-1) \in n^2$

2. Compare the order of growth of $\log_2 n$ and $\sqrt{n}$

Soln: Here $f(n) = \log_2 n$

and $g(n) = \sqrt{n}$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}}$$

$$= \lim_{n\to\infty} \frac{\left[\frac{\log_e n}{\log_e 2}\right]}{\sqrt{n}}$$

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

$$= \lim_{n\to\infty} \frac{\frac{1}{\log_e 2}(\log_e n)}{\sqrt{n}}$$

$$= \lim_{n\to\infty} \frac{\log_2 e (\log_e n)}{\sqrt{n}}$$

$$= \log_2 e \left[\lim_{n\to\infty} \frac{\log_e n}{\sqrt{n}}\right]$$

According L. Hospital rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \log_2 e \left[\lim_{n\to\infty} \frac{(\log_e n)'}{(\sqrt{n})'}\right] \quad \text{--- (1)}$$

$(\log_e n)'$ = derivation of $(\log_e n) = 1/n$ --- (2)

$(\sqrt{n})'$ = derivation of $(\sqrt{n}) = n^{1/2}$

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

$$= \frac{1}{2 \cdot n^{1/2}} = \frac{1}{2\sqrt{n}} \quad \text{—③}$$

By substituting ② and ③ in ①

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \log_2 e \left[\lim_{n\to\infty} \frac{1/n}{1/2\sqrt{n}} \right]$$

$$= \log_2 e \left[\lim_{n\to\infty} \frac{2\sqrt{n}}{n} \right]$$

$$= 2 \log_2 e \left[\lim_{n\to\infty} \frac{\sqrt{n}}{n} \right]$$

$$= 0$$

Since the limit is equal to 0, the first condition holds good and $f(n) = \log_2 n$ has smaller order growth than $\sqrt{n}$.

MATHEMATICAL ANALYSIS OF NON RECURSIVE ALGORITHMS

The general plan or procedure for analyzing non recursive algorithms is

1. Based on input size, decide the number of parameters to be considered
2. Identify the algorithms basic operation.
3. Check whether the number of times the basic operation is executed depends on the size of input.
4. Setup sum expressing the number of times the algorithm's basic operation is executed
5. Simplify using standard formulas, obtain order of growth.

A. M. PRASAD
Assistant Professor
Department of Computer Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

Example 1: Element uniqueness problem.

The algorithm to check the elements in the array are distinct.

The algorithm is given

```
Algorithm UniqueElements(A[1...n])
    // Input: An array A[1...n]
    // Output: returns "true" if all are
    // distinct and "false" otherwise
    for i ← 1 to n-1 do
    begin
        for j ← i+1 to n do
        begin
            if (a[i]=a[j]) return false
        Endfor
    Endfor
    return true
```

Illustration with numbers.

Consider the numbers 9 2 7 6 1. The logic is to select first element is compared with elements in [2...n] positions. If it matches with any element then algorithm returns ~~false~~. Otherwise second element is selected and it is Com-pared with elements in [3..n] positions. If it matches with any element then algorithm returns false. Other-wise third element is selected and process is repeated till the $(n-1)^{th}$ element.

TIME EFFICIENCY: The basic operation is if(a[i]= a[j]). The number of times the basic operation to be executed depends upon two for loops. Let T(n) be time taken by algorithm to check uniqueness.

for i← 1 to n-1 do

for j← i+1 to n do

if (a[i]=a[j]) return false

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

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

$$= \sum_{i=1}^{n-1} n-i-1+1$$

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

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

By applying values of i in expression n-i

$$T(n) = (n-1)+(n-2)+(n-3)+\cdots+3+2+1$$

Mathematically

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

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

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

Note: while finding the time efficiency, constants cannot be considered

$$T(n) = n^2 - n$$

Note: Higher order terms should be considered.

So $T(n) = n^2$.

Since it is time taken in average or worst case

$$T(n) = \theta(n^2)$$

Example 2: Multiplication of two matrices

Two matrices of size $(n \times n)$ to be multiplied.

The algorithm for multiplication of two matrices is

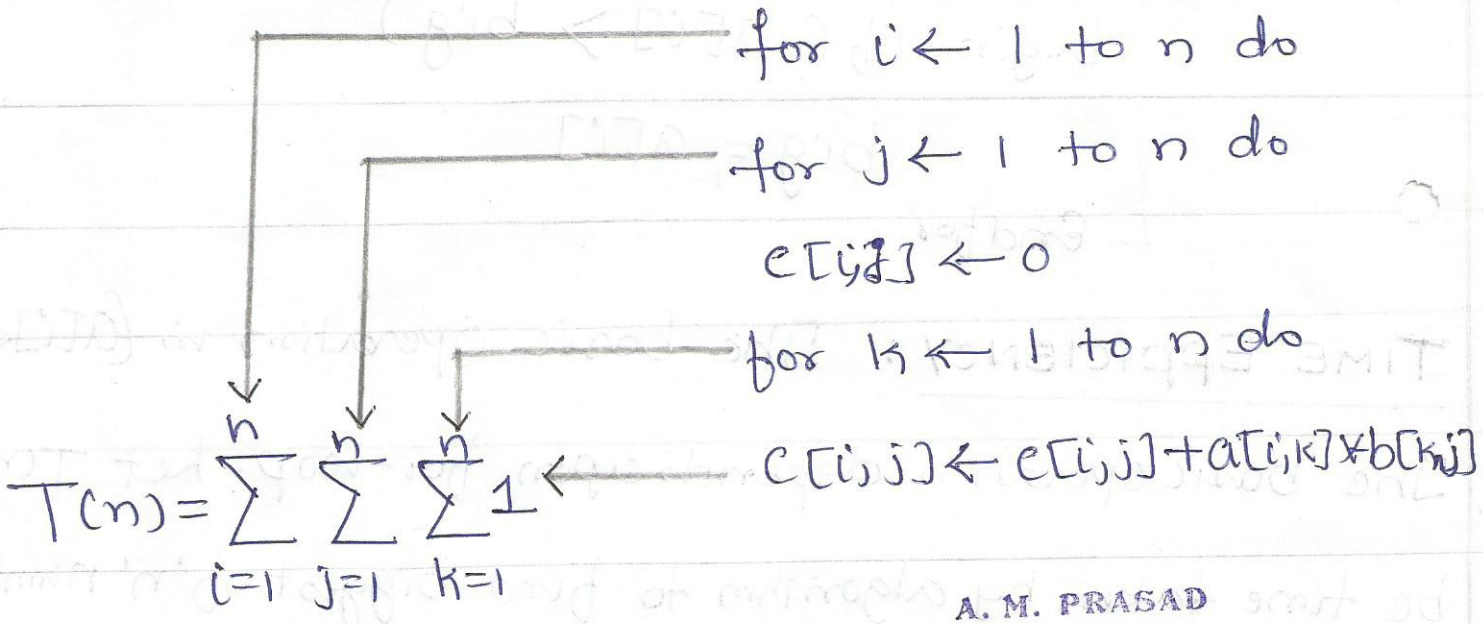
```
Algorithm MatrixMultiplication(a, b, c)
// Input: two n-by-n matrices a and b
// output: matrix c = a.b
for i ← 1 to n do begin
for j ← 1 to n do begin
c[i,j] ← 0
for k ← 1 to n do begin
c[i,j] ← c[i,j] + a[i,k] * b[k,j]
```

TIME EFFICIENCY: The basic operation is

$$c[i,j] \leftarrow c[i,j] + a[i,k] * b[k,j]$$

The execution of basic operation depends upon for-loops

Let $T(n)$ be time taken to find product of two matrices



A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

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

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} n$$

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

$$T(n) = \sum_{i=1}^{n} n^2$$

$$T(n) = \sum_{i=1}^{}$$

Since n^3 is the time taken in best or worst case

$$T(n) = \Theta(n^3)$$

Example 3: Algorithm to find biggest of 'n' numbers

```
Algorithm biggest (a[1....n])
  // i/p: array a with n elements
  // o/p: returns biggest of 'n' numbers
big = a[1]
for i ← 2 to n do
begin if ( a[i] > big )
        big = a[i]
End for
```

TIME EFFICIENCY: The basic operation is (a[i] < big)

The basic operation depends upon for-loop. Let T(n) be time taken by algorithm to find biggest of 'n' numbers.

$$T(n) = \sum_{i=2}^{n} 1$$

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

$$T(n) = n-1$$ Note: constants cannot be considered

So $T(n) = n$

Since time taken is same in all the cases

$$T(n) = \theta(n)$$

Mathematical Analysis of Recursive Algorithms

The general plan for analysis of recursive algorithms is given below

1. Based on the input size, decide the parameters
2. Identify the algorithms basic operation
3. Check the number of times the basic operation executed on different input size
4. Set up recurrance relation, for the number of times the basic operation executed
5. Solve the recurrance and obtain order of growth.

Example 1: Algorithm to find factorial of number

The recursive definition to compute n! is

$$fact(n) = \begin{cases} 1 & \text{if } n=0 \\ n * fact(n-1) & \text{if } n>0 \end{cases}$$

A. M. PRASAD
Assistant Professor
Department of Computer Science & Engineering
Ramanda Sagar College of Engineering
Bangalore - 560 078

Algorithm:

```
Algorithm fact(n)
// I/P: Integer n  O/P: factorial of n
if (n=0)
    return 1
Else
    return (n * fact(n-1))
```

TIME EFFICIENCY: The basic operation is n * fact(n-1), the total number of multiplications can be obtained using recurrance relation

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

The above recurrance relation can be solved using repeated substitution

$$T(n) = 1 + T(n-1) \text{ ; substitute for } T(n-1)$$
$$= 1 + [1 + T(n-2)]$$
$$= 2 + T(n-2) \text{ ; substitute for } T(n-2)$$
$$= 3 + T(n-3)$$

finally when n^{th} value is reached

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

Time required to find factorial of 0 is 0

So $T(0) = 0$ So $T(n) = n$

Since all different time efficiencies are same

$$T(n) = \theta(n)$$

Example 2: Tower of Hanoi problem

: The goal of this problem is to move three different sized plates from source to destination pole using temporary pole. During transfer of plate, the smaller plate cannot be placed on bigger one. This can be done recursively using following steps.

* Move n-1 disks recursively from source to temp
* Move n^{th} from source to destination
* Move n-1 disks from temp to destination.

A. M. PRASAD
Assistant Professor
Department of Computer Science & Engineering
Vivekananda Sagar College of Engineering
Bangalore - 560 078

The algorithm is

```
Algorithm TowerofHanoi (n, source, dest, temp)
    // n: total number of disks to be moved
    // o/p: all n disk should be in destination pole

    if (n != 0)
begin   TowerofHanoi(n, source, temp, dest)
        write source 'to' dest
        TowerofHanoi(n, temp, dest, source)
End
```

The recurrance relation is

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1)+1+T(n-1) & \end{cases}$$

No of disk movements from source to temp

disk movement from source to destination

No of disk movements from temp to destination

if $T(n)$ is the time taken move 'n' discs, then

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

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

Now to solve recurance relation, we reverse substitution.

$$T(n) = 2\left[2T(n-2)+1\right] + 1$$

$$= 2^2 T(n-2) + 2 + 1$$

Now substitute for $T(n-2)$, the equation becomes

$$= 2^3 T(n-2) + 2^2 + 2^1 + 1 \quad \text{and so on.}$$

In general

$$= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2^2 + 2^1 + 2^0$$

the geometric series $2^0 + 2^1 + 2^2 + \cdots + 2^{i-2} + 2^{i-1}$

can be written as $\dfrac{a(r^n-1)}{r-1}$

where $a=1, r=2, n=i$

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

$$= 2^i - 1$$

$$= 2^i T(n-i) + 2^i - 1, \text{ let } i=n-1$$

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

$T(n) = 1$

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

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

$$= 2 * 2^{n-1} - 1$$

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

$$= 2^n - 1$$

So $T(n) = 2^n - 1$

The number of disk movements is same in best, worst case. So time efficiency is given by

$$T(n) = \theta(2^n - 1) \in \theta(2^n)$$

Example 3: Fibonacci numbers

The fibonacci series is an infinite sequence of integers 0, 1, 1, 2, 3, 5, 18, 13, 21 .....

The recurrance relation is

$$fib(n) \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ fib(n-1)+fib(n-2) & \text{Otherwise} \end{cases}$$

The algorithm for fibonacci numbers

```
Algorithm fib(n)
// n: the number of terms to be generated
//                    nth fibonacci numbers

if (n=0) return 0
if (n=1) return 1
return fib(n-1) + fib(n-2)
```

TIME EFFICIENCY: To generate n^{th} fibonacci number

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

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

which is of form

$$a * fib(n) + b * fib(n-1) + c * fib(n-2) = 0$$

which is like Quadratic Equation $ax^2+bx+c=0$

from equation $fib(n) - fib(n-1) - fib(n-2) = 0$

$a=1, \quad b=-1, \quad c=-1.$

The roots of Quadratic equation are

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

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

this can be solved using formula

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

$$= \alpha \left[\frac{1+\sqrt{5}}{2}\right]^n + \beta \left[\frac{1-\sqrt{5}}{2}\right]^n$$

if $n=0$

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

$$= \alpha+\beta \quad \text{or} \quad \alpha+\beta=0$$

if $n=1$

$$X(1) = \alpha \left[\frac{1+\sqrt{5}}{2}\right]^1 + \beta \left[\frac{1-\sqrt{5}}{2}\right]^1 = 1 \quad \text{—(A)}$$

$X(0) = \alpha + \beta$ or $\alpha + \beta = 0$ or $\alpha = -\beta$

Substituting this in Equation (A)

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

$$-\beta\left[\frac{1-\sqrt{5}}{2}\right] + \beta\left[\frac{1-\sqrt{5}}{2}\right] = 1$$

$$\beta\left[\left[\frac{1-\sqrt{5}}{2}\right] - \left[\frac{1+\sqrt{5}}{2}\right]\right] = 1$$

$$\beta\left[\frac{1}{2} - \frac{\sqrt{5}}{2} - \frac{1}{2} - \frac{\sqrt{5}}{2}\right] = 1$$

$$\beta[-\sqrt{5}] = 1$$

$$\beta = -\frac{1}{\sqrt{5}}$$

But we know that $\alpha = -\beta$ when $n=0$

So $\alpha = \frac{1}{\sqrt{5}}$ substituting this in Eqn (A)

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

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

where $\phi = \left[\frac{1+\sqrt{5}}{2}\right]$ and $\hat{\phi} = \left[\frac{1-\sqrt{5}}{2}\right]$