Note:

UNIT-I

i. is Algorithm?

An algorithm is a sequence of unambiguous instructions for solving a problem ie for obtaining a required output for any legitimate input in a finite amount of time.

Points to remember about Algorithm

→ The nonambiguity requirement for each step of an algorithm cannot be compromised

→ The range of inputs for which an algorithm works has to be specified carefully

→ The same algorithm can be represented in several different ways

→ Several algorithms for solving the same problem may exist

→ Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds

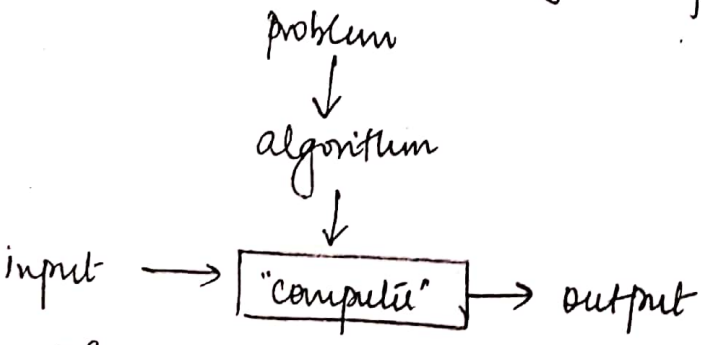


Fig Notion of algorithm

Example GCD of two non-negative, non-zero integers m and n → gcd(m,n)

It is defined as the largest integer that divides both m and n evenly ie with remainder of zero.

Euclid's algorithm

Algorithm Euclid(m,n)
// Computes gcd(m,n) by Euclid's algorithm
// Input: Two non-negative, not-both-zero integers m and n
// Output: GCD of m and n

```
while n≠0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

gcd(60,24) = gcd(24,12), gcd(12,0) = 12

2] Consecutive integers checking algorithm for computing gcd(m,n)

Step1 Assign the value of min{m,n} to t

Step2 Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise go to step 4

Step3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise proceed to Step 4

Step4 Decrease the value of t by 1 Goto Step 2

Note:- This algorithm does not work correctly when one of its input number is zero.
Thus we should specify the range of an algorithm's input explicitly and carefully.

Middle-school procedure for computing gcd(m, n)

Step 1: Find the prime factor of m
Step 2: Find the prime factor of n
Step 3: Identify all the common factor in the two prime expansions found in Step 1 and Step 2 (If p is a common factor occuring p_m and p_n times in m and n, respectively, it should be repeated $\min\{p_m, p_n\}$ times.)
Step 4: Compute the product of all the common factors and return it as the greatest common divisor of the numbers given

eg 60 & 24

$60 = 2 \cdot 2 \cdot 3 \cdot 5$

$24 = 2 \cdot 2 \cdot 2 \cdot 3$

$gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12$

Sieve of Eratosthenes

A simple algorithm for generating consecutive primes not exceeding any given integer n.

Eg n = 25

First list out all the numbers from 2 to 25

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17
18, 19, 20, 21, 22, 23, 24, 25

Remove all the values divisible by 2

2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25

Remove all the numbers divisible by 3

2, 3, 5, 7, 11, 13, 17, 19, 23, 25

by 5

2, 3, 5, 7, 11, 13, 7, 19, 23.

↳ Prime numbers.

Algorithm Sieve(n)

// Implements the sieve of Eratosthenes
// 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

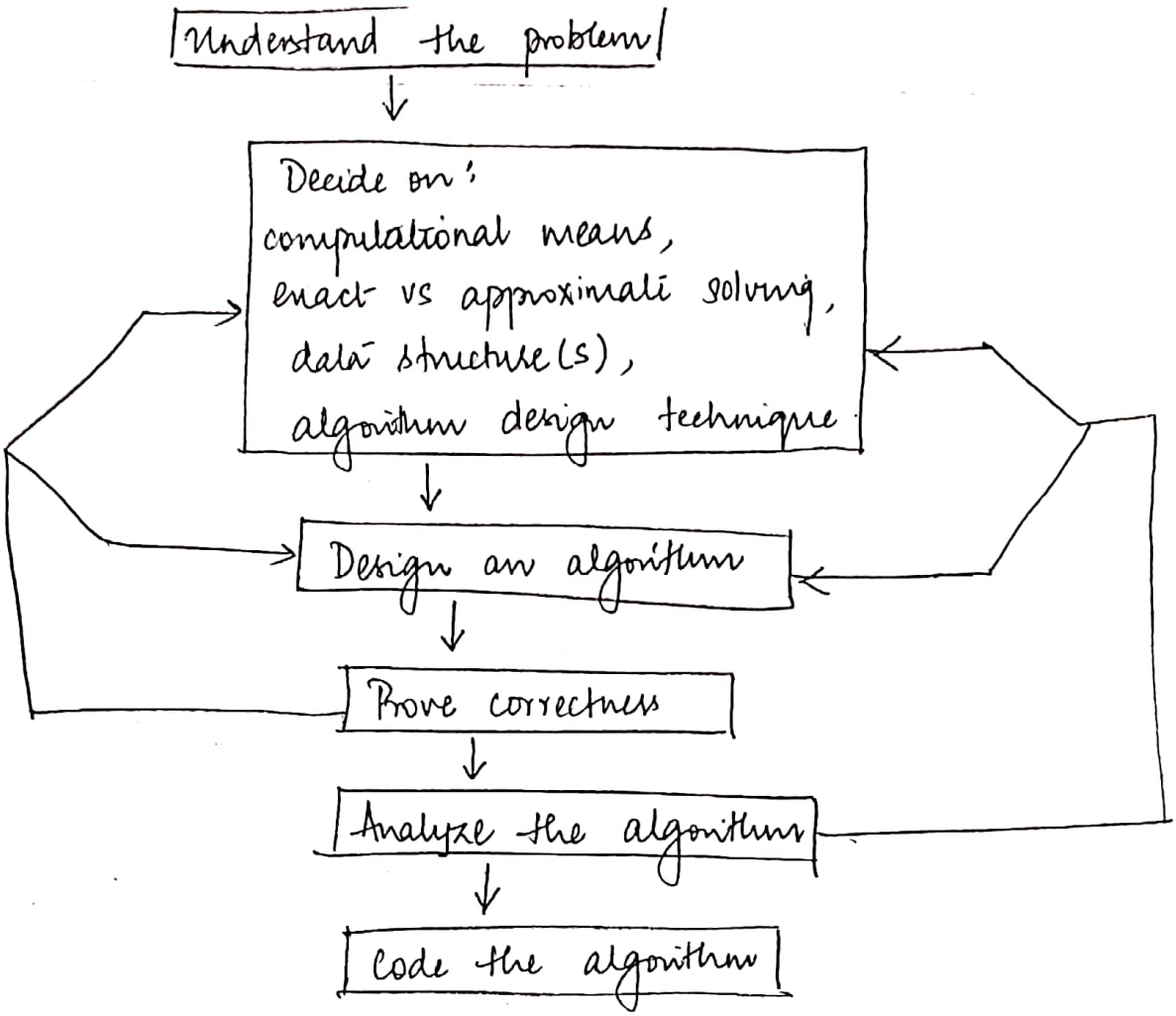
for p ← 2 to $\lfloor \sqrt{n} \rfloor$ do

```
        if A[p] ≠ 0    // p hasn't been eliminated on
            j ← p*p        previous passes.
            while j ≤ n do
                A[j] ← 0 // mark element as eliminated
                j ← j+p

// copy the remaining elements of A to array L
  of the primes
  i ← 0
  for p ← 2 to n do
      if A[p] ≠ 0
          L[i] ← A[p]
          i ← i+1
  return L
```

Fundamentals of Algorithmic Problem Solving

Algorithm design and analysis process



① Understanding the problem

Given a problem we have to understand the problem completely and clearly. If we have any doubt, we have to ask ques and clarify the doubt. we should should find what i/p is available and what i/p is missing, how is it useful to solve the problem. we should also know that what the i/p should be given and what should be the expected result/output. It is very essential to specify the exact range of i/ps given to the problem or algorithm.

The correct algorithm is not the one which works most of the time but one of that works correctly for all legitimate i/ps (in case range of i/p is not specified)

② Ascertain the capability of computational device

Based on the architecture of computational device we have to use 2 types of algorithms.
For the device based on Von Neumann architecture in which instructions are executed sequentially, we have to design a sequential algorithm.
If a device is capable of executing instructions in parallel, we may have to design parallel algorithms

③ Choosing between exact & approximate problem solving

The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an exact algorithm; in the latter case an algorithm is called approximation algorithm.
Why would one opt for an approximation algorithm? First, there are important problems that simply cannot be solved exactly, such as extracting square roots, solving nonlinear equations and evaluating definite integrals.
Second, available algorithms for solving a problem

may even be unacceptably slow because of the problem's intrinsic complexity. The most well known of them is the travelling salesman problem - finding the shortest tour through n cities.

Deciding on Appropriate Data Structures

Some algorithms do not demand any ingenuity in representing their inputs but others are, in fact, predicated on ingenious data structures. Select the appropriate data structures so that we can write efficient program

Algorithms + data structure = Programs

So we have to choose efficient algorithms & data structures.

- ~~Designing an algorithm~~

Algorithm Design Techniques

Now, with all the components of the algorithmic problem solving in place, how do you design an algorithm to solve a given problem?
An algorithm design technique (or strategy or paradigm) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

c) Methods of specifying an Algorithm

Once you have designed an algorithm, you need to specify it in some fashion

Natural language - using a natural language has an obvious appeal, however the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult.

Pseudocode

A pseudocode is a mixture of a natural language and programming language like constructs.

A pseudocode is usually more precise than a natural language, and its usage often yields more succinct algorithm descriptions

Flowchart :-

In earlier days of computing, the dominant vehicle for specifying algorithms was a flowchart, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

This representation technique has proved to be inconvenient for all but very simple algorithms, now a days it can be found only in old algorithm books.

The state of the art of computing has not yet reached a point where an algorithm's description - whether in a natural language or a pseudocode - can be fed into an electronic computer directly. Instead it needs to be converted into a computer program written in a particular computer language.

) Proving an Algorithm's Correctness

Once an algorithm has been specified, you have to prove its correctness. That is you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.

For example, correctness of euclid's algorithm for computing gcd stems from correctness of the equality gcd(m,n) = gcd(n, m mod n), the simple observation that the second number gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the number becomes 0.

e) Analyzing an Algorithm

Efficiency → time efficiency indicates how fast the algorithm runs

space efficiency indicates how much extra memory the algorithm needs.

Simplicity → Elaborate! Simpler algo are easier to understand & program & have fewer bugs
Generality → generality of problem + ip/p range
9) Coding an algorithm more efficient.

Once we have designed an algorithm, & verified its correctness, next step is to convert it into an equivalent programming code. We can write the program in many ways, the syntax may vary from language to language. The selection of programming language is also important. As a result, rule says that a good algorithm is the result of repeated effort & rework.

Important Problem Types

→ Sorting
→ Searching
→ String processing — String matching
→ graph problems — TSP,
→ Combinational problems.
→ Geometric problems
→ Numerical problems.

Fundamental Data Structures.

—Data Structure — linear DS.
linked lists
queue
graphs —
trees

Fundamentals of the analysis of algorithm efficiency

Analysis of algorithm means to investigate the algorithm's efficiency with respect to resources ie running time & memory space

Analysis framework

Time efficiency – indicates how fast an algorithm in question runs

Space efficiency - deals with the extra space the algorithm requires

1) Measuring i/p size

Depends on the situation (ie on the problem given)
Here we are investigating the algorithm efficiency as a function of some parameter 'n' indicating that algorithm i/p size.
eg - problem involving like sorting we can say that i/p size is n. But on the other side if you consider computing product of 2 matrices (m x n)
There are 2 measures for this problem. The 1st measure used is matrix order 'n'. But natural contendor is total no. of elements 'n' in a matrix being multiplied. So choice of appropriate size matrix also influenced by operation of algorithm.

Measuring the run time

We can simply use some standard unit of time measurement like second, millisecond and so to measure the running time But if we use that there are certain drawbacks like (1) it depends on the speed of particular computer (2) depends on quality of program (3) compiler used & so on So we have to measure algorithm run time that doesn't depend on external factor

So, basic thing to do is identify most important operation of an algorithm called basic operation

Basic operation - operation that contribute most towards the running time of the algorithm ie stalimat- that executes max. no. of time it a function is called basic operation Thus the established framework for analysis of an algorithms time efficiency suggests measuring it by counting no of times the algorithms basic operation is executed on i of size n.

Basic operation

Problem	Input size	Basic operation
1) Linear search	No. of lists of items ie n	key comparison
2) Multiplication of 2 matrices	dimension of matrix or total no. of elements	multiplication of 2 numbers
3) Graph problem	vertices & edges	visiting the vertex or traversing the edge.

Theoretical Analysis of Algorithm Efficiency

(Time efficiency)

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

C_{op} - execution time for basic operation

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

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

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

we can ignore n for small values

how much longer will the algorithm run if we double its input size?

ie $\frac{T(2n)}{T(n)} = \frac{C_{op} \frac{1}{2} (2n)^2}{C_{op} \frac{1}{2} n^2} = 4$

The efficiency analysis framework ignores multiplicative constant ($\frac{1}{2}$) and C_{op} (both are cancelled out) and concentrates on the counts ~~of~~ order of growth of $C(n)$.

Orders of Growth :-

We expect the algorithms to work faster for all values of n. Some algorithms execute faster for smaller values of n. But, as the value of n increases, they tend to be very slow.

So, the behavior of some algorithm changes with increase in value of n. This change in behavior of the algorithm and algorithms efficiency can be analyzed by considering the highest order of n. The order of growth is normally determined for larger values of n :-

1) The behavior of algorithm changes as the value of n increases

2) In real time applications we normally encounter large values of n.

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}		

The function growing the slowest among these is the logarithmic function.

2^n & n! → Factorial

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

lowest — highest

Worst-case, Best-case and average case efficiencies

Will the algorithm efficiency depends on the size of algorithm input alone? " — For many algorithms running time depends not only on an input size but also on the specific of particular i/p.

eg Linear search

Best case efficiency - The efficiency of an algorithm for the input of size n for which the algorithm takes least time during execution among all possible inputs of that size is called best case efficiency.

eg Item is present in the beginning

$C_{best}(n) = 1$

Worst case efficiency - The efficiency of an algorithm for the input of size n for which the algorithm takes longest time to execute among all possible inputs is called worst case efficiency

eg Item not present in the array is an example of worst case efficiency.

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

Indicate whether the first function of each of the following pairs has a smaller, same or larger order of growth than second function.

a) $n(n+1)$ and $2000n^2$

$n(n+1)$ and $2000n^2$

$\Downarrow$ $\Downarrow$

n^2+n $2000n^2$

$\Downarrow$ $\Downarrow$

Same order

b) $100n^2$ and $0.01n^3$

$\Downarrow$ $\Downarrow$

2 3

$100n^2$ has lower order of growth that $0.01n^3$

c) $\log_2^2 n$ and $\log_2 n^2$

$\Downarrow$ $\Downarrow$

$\log_2 n * \log_2 n$ $2\log_2 n$

$\log_2^2 n$ has higher order of growth than $\log_2 n^2$

d) 2^{n-1} and 2^n.

Same order

Average case efficiency

Assumptions:-

a) probability of successful search is p $(0 \le p \le 1)$

b) the probability of first match occurring in ith position of list is the same for every i.

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

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

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

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

$p=1$ - successful $\rightarrow$ on average $\frac{n+1}{2}$

$p=0$ - unsuccessful. $\rightarrow$ n comparisons

In case of successful search, the probability of first match occurring in ith position of list is p/n for every i & no. of comparisons is i

Asymptotic Notations and basic efficiency classes

Informal defn

O-notation

$O(g(n))$ is the set of all functions with a smaller or same order of growth as $g(n)$

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

$100n + 5 \in O(n^2)$

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

$n^3 \notin O(n^2)$

$0.001n^3 \notin O(n^2)$

$n^4 + n + 1 \notin O(n^2)$

Ω-notation

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

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

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

$100n + 5 \notin \Omega(n^2)$

$\Theta(g(n))$

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

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

Formal definitions

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 bound above by some constant multiple of $g(n)$ for all large n ie if there exist some positive constant c and some non-negative integer n_0 such that

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

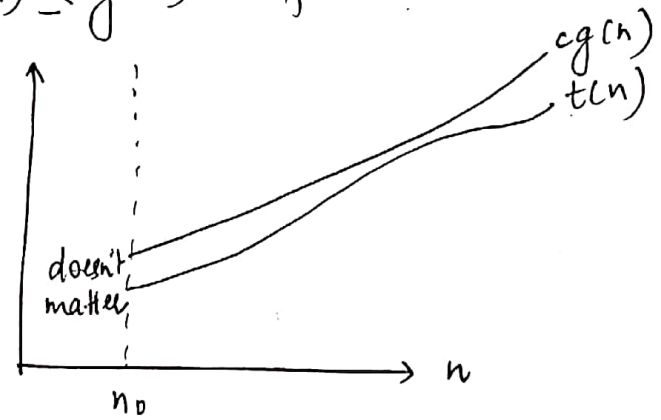


fig: $t(n) \in O(g(n))$

eg Let $f(n) = 100n + 5$, replacing 5 with n (so that next higher order term is obtained) we get, $100n + n$ and call it $c \cdot g(n)$

ie $c \cdot g(n) = 100n + n$ for $n = 5$

$= 101n$ for $n = 5$

Now $f(n) \leq c * g(n)$ for $n \geq n_0$

$100n + 5 \leq 101n$ for $n \geq 5$

$c = 101$, $g(n) = n$, & $n_0 = 5$.

-- notation

function $t(n)$ is said to be in $\Omega(g(n))$, symbol $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, ie if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0$$

eg $f(n) = 10n^3 + 5$. Express $f(n)$ using Big-omega

Soln

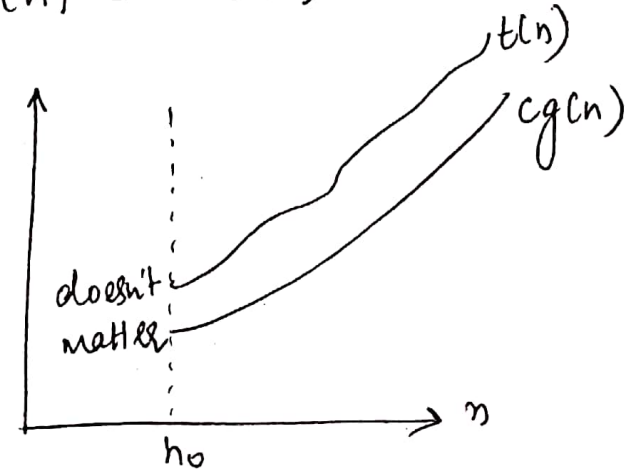
$f(n) \geq c * g(n)$ for $n \geq n_0$

$\downarrow$

$10n^3 + 5 \geq 10 * n^3$ for $n \geq 0$

$c = 10$, $g(n) = n^3$ & $n_0 = 0$.

$\therefore f(n) \in \Omega(n^3)$.

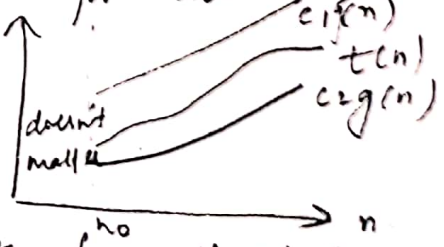


$t(n) \in \Omega g(n)$

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 multiples of $g(n)$ for all large n ie if there exist some positive constant c_1 and c_2 and some non negative integer n_0 such that

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



Let us prove $\frac{1}{2}n(n-1) \in \Theta(n^2)$,

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \text{for all } n \geq 0$$

Second we prove the left inequality (lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \quad (\text{for all } n \geq 2)$$

$$= \frac{1}{4}n^2.$$

Hence $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$ and $n_0 = 2$.

$\frac{1}{2}n = \frac{1}{2}n\frac{1}{2}n$
$\frac{1}{2}n^2 = \frac{1}{4}n^2$
$\frac{1}{2} = \frac{1}{4}n$, $n = 4/2$

Useful Property involving Asymptotic Notations

Theorem:- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$
then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$

Pf:- Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some non negative integers n_1 such that

$$t_1(n) \leq c_1 g_1(n) \quad \text{for all } n \geq n_1$$

Similarly since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \leq c_2 g_2(n) \quad \text{for all } n \geq n_2$$

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

Adding the two inequalities above yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) \\ &\leq c_3 [g_1(n) + g_2(n)] \\ &\leq c_3 2 * \max\{g_1(n), g_2(n)\} \end{aligned}$$

↓

(Four arbitary real numbers a_1, b_1, a_2 and b_2 if $a_1 \leq b_1$ and $a_2 \leq b_2$ then ~~$a_1 + a_2 \leq$~~ $a_1 + a_2 \leq 2 \max\{b_1, b_2\}$)

Hence $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$ with the constants c and n_0 required by O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $n_0 = \max\{n_1, n_2\}$

Hence the proof.

Mathematical Analysis of Nonrecursive Algorithms

1] Finding the value of the largest element in a list of n numbers.

Algorithm MaxElement (A[0...n-1])
// Determines the value of the 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
maxval ← A[0]
for i ← 1 to n-1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval.

General plan for Analyzing time efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size
2. Identify the algorithm's basic operation (As a rule it is located in its innermost loop)
3. Check whether the no. of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst case, average case & if necessary best case efficiencies have to be investigated separately
4. Set up a sum expressing the number of times the algorithm's basic operation is executed

using standard formulas and rules of sum manipulation either find a closed form formula for the count or at the very least establish its order of growth.

Rules

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

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

$$\sum_{i=l}^{n} 1 = u - l + 1 \text{ where } l \le u \text{ are some lower \& upper integer limits}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2)$$

Step 1 :- Input size = n

Step 2 :- There are two operations :-

1) A[i] > maxval

2) maxval ← A[i]

But we consider A[i] > maxval since comparison is executed on each repetition of the loop

Step 3 :- The basic operation is executed once each time the control enters the loop.

So the total no. of times the basic operation is executed is given by :

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

$$= n-1-1+1 \quad \longrightarrow \quad \sum_{i=l}^{u} 1 = u-l+1$$

$$= n-1 \in \Theta(n).$$

2] **Element uniqueness problem.**

Check whether all the elements in a given array are distinct.

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

// Determines 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 or "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
```

Step 1:- Input size = n

Step 2:- Basic operation is (if a[i] = a[j])

Step 3:- The no. of comparison not only depends on value of n but also on position of repeated element if present. The repeated elements if not present the no. of comparisons are more [illegible] case.

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

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

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

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

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

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

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

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

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

~~$n^2 - 2n + 1 - n^2 + n + 2n - 2$~~

$$\frac{2n^2 - 4n + 2 - n^2 + n + 2n - 2}{2}$$

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

$$\approx \frac{1}{2} n^2 \in \Theta(n^2)$$

3) Matrix Multiplication

Given two n-by-n matrices: A and B find the time efficiency of the definition based algorithm for computing their product C = AB.

Algorithm Matrix multiplication (A[0..n-1, 0...n-1], B[0..n-1, 0..
// Multiplies two n-by-n matrices by the definition base
// algorithm
// Input: Two n-by-n matrices A and B
// Output: Matrix C = AB

```
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        C[i,j] ← 0.0
        for k ← 0 to n-1 do
            c[i,j] ← C[i,j] + A[i,k] * B[k,j]
return C
```

The time complexity in the best case & worst case remains same.

Step 1: Input size parameter = n

Step 2: Basic operation – multiplication.

Step 3: The no. of multiplications depends on the value of n only and not on any other factors

Total no. of times the multiplication statement is executed can be obtained as follows:-

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

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

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

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

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

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

$$= n^3 .$$

Mathematical Analysis of Recursive Algorithms

) Decide on a parameter (or parameters) indicating an input's size

) Identify the algorithm's basic operation

) Check whether number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average case and best-case efficiencies must be investigated separately

4) Set up a recurrence relation with an appropriate initial condition for the no. of time the basic operation is executed

5) Solve the recurrence or atleast ascertain the order of growth of its solution

1) Factorial of a number

Algorithm F(n)

// Compute n! recursively
// Input: A non negative integer n
// Output: The value of n!

if n=0 return 1
else return F(n-1) * n

Analysis

Step 1 :- Input size parameter = n

Step 2 :- Basic operation - multiplication

Step 3 :- The total no. of multiplications can be obtained by :-

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

$M(n-1)$: to compute $F(n-1)$

1 : to multiply $F(n-1)$ by n

To determine a ~~cond~~ solution uniquely we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

```
if n=0 return 1
```

$\therefore M(0) = 0$ ← no multiplications when $n=0$

the calls stop when $n=0$

Hence the recurrence relation :-

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

We use method of backward substitution.

$M(n) = M(n-1) + 1$ Substitute $M(n-1) = M(n-2) + 1$

$= [M(n-2) + 1] + 1$

$= M(n-2) + 2$ Substitute $M(n-2) = M(n-3) + 1$

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

$= M(n-3) + 3$

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

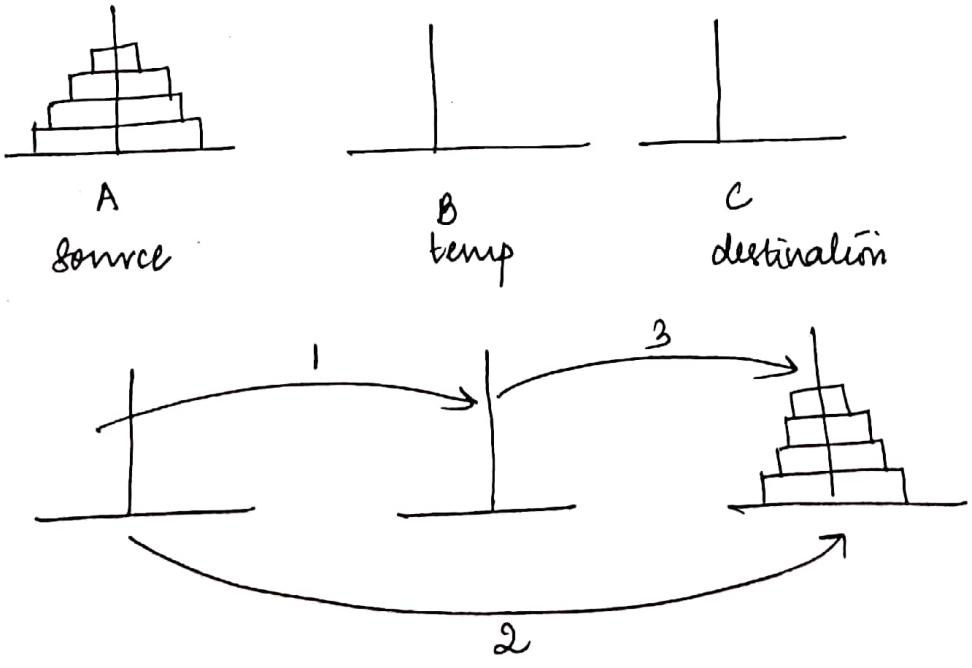
$= M(n-n) + n$ → To get initial condition $M(0) = 0$ let $i = n$

$= M(0) + n$

$= 0 + n$

$= n$

Towers of Hanoi



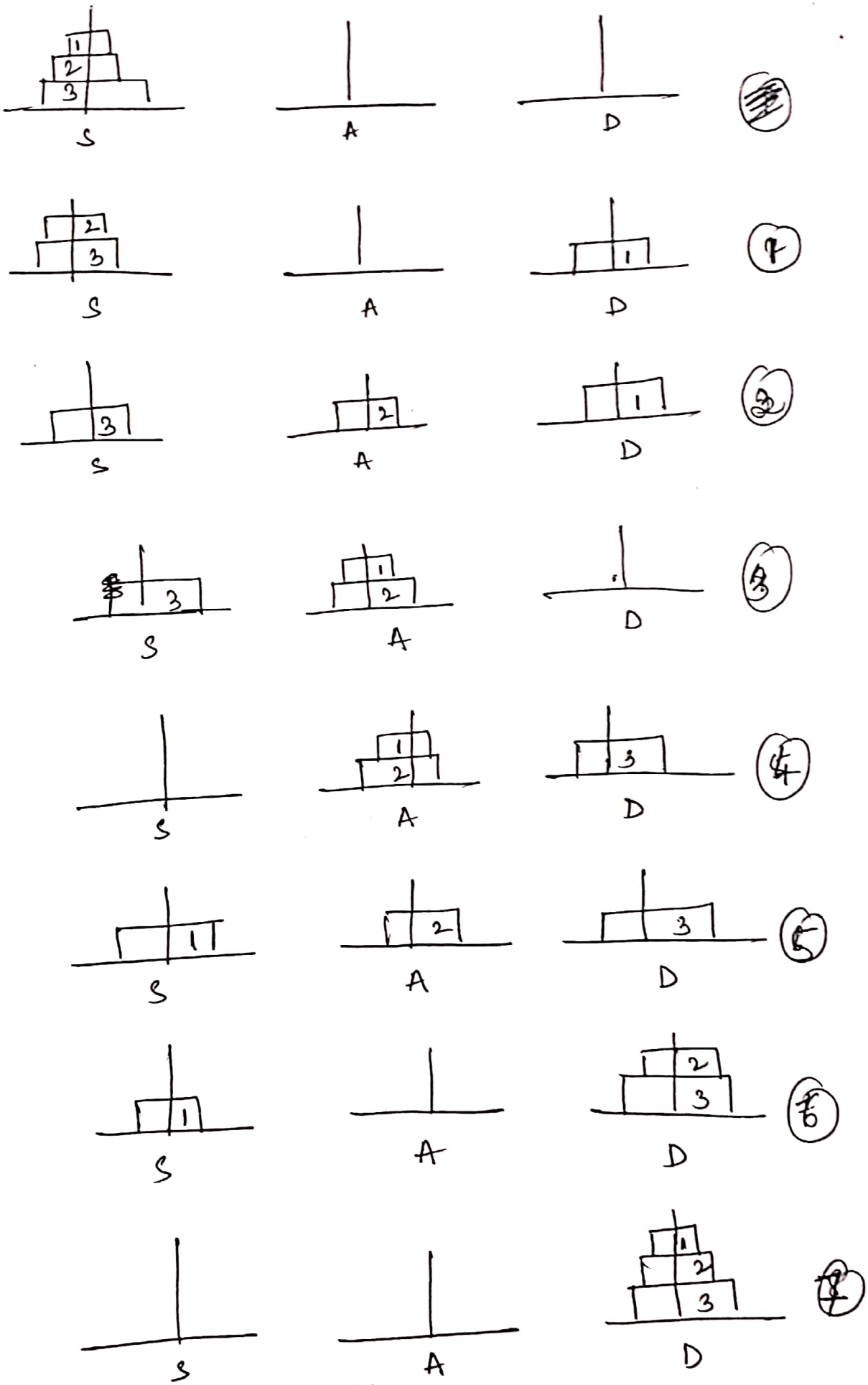
// Algorithm TowersOfHanoi (n, source, temp, destination)

// Purpose:- To move n discs from source to destination and see that only one disc is moved and always the smaller disc should be placed above the larger disc using one of the ~~needl~~ pe as temporary holder for the disc being moved

//Inputs:- n: total no. of disks to be moved

// Output: all n disks should be available on the destination ~~needle~~ peg

```
if (n=1)
    write ('Move disk 1 from source to destination')
    return
end if
//Move n-1 disks from A to B using C as tempora
// ~~needle~~ peg
TowersOfHanoi (n-1, source, destination, temp)
write ("Move disk", n, " from", source, "to", destination)
//Move n-1 disks from B to C using A as temporary
TowersOfHanoi (n-1, temp, source, destination)
```



Analysis

Step1 :- Input size parameter = n - ie no. of disks

Step 2 :- The basic operation is movement of disk

Step 3 :- No. of moves M(n) depends only on n

$\therefore M(n) = M(n-1) + 1 + M(n-1)$ for $n > 1$

Initial condition is $M(1) = 1$ ie when only one disk on source peg move it from source to destination

$\therefore$ The recurrence relation :-

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1$$
$$M(1) = 1$$

Solving the recurrence relation,

$M(n) = 2M(n-1) + 1$ Substitute $M(n-1) = 2M(n-2)$

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

↳ Substitute $M(n-2) = 2M(n-3)$

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

$= 2^3 M(n-3) + 2^2 + 2 + 1$

In general,

$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \dots + 2^3 + 2^2 + 1$

$= 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \dots + 2^1 + 2^0$

This geometric series can be solved using $S = \frac{a(r^n - 1)}{r - 1}$

where $a=1$, $r=2$ and $n=i$ (since no. of terms from 0 to $i-1 = i$)

$$\text{So, } S = \frac{1(2^i-1)}{2-1} = 2^i-1$$

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

To initial condition M(1), let $i=(n-1)$

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

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

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

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

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

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

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

The time complexity for Tower of Hanoi problem is $M(n) \in \Theta(2^n-1) \in \Theta(2^n)$

Digits in a binary number (given a decimal number) (Iterative)

Let us initial count to 1. As long as the given N is greater than 1 keep dividing it by 2 and increment corresponding count by 1.

Algorithm Binary(n)

// Purpose To count the no. of digits in a binary representation of a given positive decimal integer.

// Inputs :- n: a positive decimal integer

// Output :-

// count - no. of digits in a binary representation of a given decimal integer.

```
count ← 1
while (n > 1)
    count ← count + 1
    n ← ⌊n/2⌋
end while
return count
```

2 | 24
2 | 12 — 0
2 | 6 — 0
2 | 3 — 0
1 — 1

11000

count = 1
24 > 1 ✓
C = 2
n = 12
n > 1 ✓
C = 3
n = 6
n > 1 ✓
C = 4
n = 3
n > 1 ✓
C = 5
n = 1

Design (recursion) The time complexity of this algorithm can be found very easily if the recursive version of this algorithm is available.

$\therefore A(n) = \log_2 n \in \Theta(\log n)$

Fibonacci Numbers

~~Recursive~~ Algorithm:- Fibonacci(n)

// Purpose : find nth fibonacci number

// Inputs :

n: a positive integer

// Output :

// nth fibonacci number

if (n=0) return 0

if (n=1) return 1

return F(n-1) + F(n-2) for n>1

Explicit Formula for nth fibonacci Numbers

If we apply the method of backward substitutions to solve recurrence, we will fail to get an easily discernible pattern. Instead, let us take advantage of a theorem that describes solutions to a homogeneous second order linear recurrence with constant coefficients

$$ax(n) + bx(n-1) + cx(n-2) = 0. \quad \text{——} (1)$$

where a, b and c are some fixed real nos. ($a \neq 0$) called the coefficients of the recurrence

The characteristic equation for recurrence equation (1) is

$$ar^2 + br + c = 0 \quad \text{——— (2)}$$

Let us apply theorem to the case of Fibonacci nos.

$$\therefore F(n) - F(n-1) - F(n-2) = 0 \quad \text{——— (3)}$$

Comparing equation (1) & (3)

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

Substituting the values in (2), the characteristic equation is :-

$$r^2 - r - 1 = 0$$

The roots are,

$$r_{1,2} = \frac{1 \pm \sqrt{1 - 4(-1)}}{2} = \frac{1 \pm \sqrt{5}}{2}$$

$$\therefore r_1 = \frac{1 + \sqrt{5}}{2}, \quad r_2 = \frac{1 - \sqrt{5}}{2}$$

Substituting the formula, $F(n) = \alpha r_1^n + \beta r_2^n$

we have,

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

Now taking the initial condition,

$F(0)=1$ & $F(1)=1$

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

$$F(1) = \alpha\left(\frac{1+\sqrt{5}}{2}\right) + \beta\left(\frac{1-\sqrt{5}}{2}\right) = 1 \quad \text{---} (6)$$

from equation (5)

$$\alpha = -\beta \quad \text{---} (7)$$

Substituting this in 6, we have

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

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

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

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

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

$$\beta = \frac{-1}{\sqrt{5}} \quad \text{---} (8)$$

Substituting $\beta = \frac{-1}{\sqrt{5}}$ in (7)

$$\alpha = \frac{1}{\sqrt{5}}$$

Substituting α and β in eq (4) we have,

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

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

$$= \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right) \quad \text{——— (9)}.$$

$$\phi = \left(\frac{1+\sqrt{5}}{2}\right) \text{ and } \hat{\phi}\left(\frac{1-\sqrt{5}}{2}\right).$$

Using (9) we can find nth Fibonacci Number

Analysis

Basic operation – Addition. A(n).

– Let A(n) be the number additions needed for computing ~~f(n-1)~~ f(n).

– No. of Additions needed for computing f(n-1) and f(n-2) are A(n-1) & A(n-2). respectively and the algorithm needs one more addition to compute their sum.

∴ Recurrence relation,

$$A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1$$

$$A(0) = 0 \quad , A(1) = 0.$$

ie $A(n) - A(n-1) - A(n-2) = 1$ — [but here right hand side is not equal to zero].

Such recurrences are called inhomogeneous recurrences.

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

and Substituting $B(n) = A(n)+1$

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

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

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

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

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

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

Hence $A(n) \in \Theta(\phi^n)$