Unit-1

Algorithm

→ set of rules for carrying out calculation either by hand / on a machine.

→ Sequence of computational steps that transform the i/p into the o/p.

→ A finite set of instruction that specify a sequence of operations to be carried out in order to solve a specific pbm / class of pbms is called an algorithm.

An algorithm must have the following properties:

→ **finiteness** - Algm must complete after a finite no of instructions have been executed.

→ **Absence of ambiguity**: Each step must be clearly defined, having only one interpretation

→ **Definition of sequence**: Each step must have a unique defined preceding and succeeding step. The first & last step must be clearly noted.

→ **Input / output**: No & types of required i/p's & results must be specified.

→ Feasibility: It must be possible to perform

each instructions ②

Characteristics of an Algorithm

1. Input – zero/more quantities are externally supplied
2. Output – atleast one quantity is produced
3. Definiteness → each instruction is clear & unambiguous
4. Effectiveness → each step must be done by person using pencil in a finite amount of time.
5. Termination/finiteness – terminates after a finite no of operations.

→ Algorithms produces one/more outputs & have zero/more inputs that are externally supplied.

① Understanding – clarify doubts, commonly occurring existing algm strength & weakness instance, range of i/p's.

② Decision making

- Seq & parallel algm, exact & approximation algm.

data structure

Alg. design techn → Brute force, divide & conquer...

③ Specification — Algm → N.L
→ Pseudocode
→ flowchart

④ Alg Verification → Correct o/p in finite amount of time for a valid set of i/p.

⑤ Analyze the → Time, space, Simplicity, Generality
Range of i/p.

Algorithm

Some rules for writing the algorithm

Algorithm Heading

It consists of name of algorithm, problem description, input and output

Algorithm Body

It consists of logical body of the algorithm by making use of various programming constructs & assignment statement

Structure of Algorithm

1) Algorithm heading consists of keyword Algorithm and name of the algorithm & parameter list.

Syntax is

```
Algorithm name (P1, P2, ..... Pn)
```

- `Algorithm` — This keyword should be written first
- `name` — Name of the algorithm
- `(P1, P2, ..... Pn)` — write parameters (if any)

2) Then in the head section write the following

```
// Problem description:
// Input:
// Output:
```

3) Body of the algorithm is written after the head section using various programming constructs like if, for, while or some assignment statements may be written.

4) The compound statements should be enclosed within { and } brackets.

5) Single line comments are written using // as beginning of comment.

6) The identifier should begin by letter & not by digit. An identifier can be a combination of alphanumeric string. It is not necessary to write data types explicitly for identifiers.

7) For assignment use '←'

Variable ← expression

8) The inputting & outputting can be done using read and write.

eg) write(" Message");
read (val);

9) For conditional statements

if (condition) then statement
if (condition) then statement else statement

10) while statement

```
while (condition) do
{
    statement 1
    statement 2
      :
    statement n
}
```

11) For loop

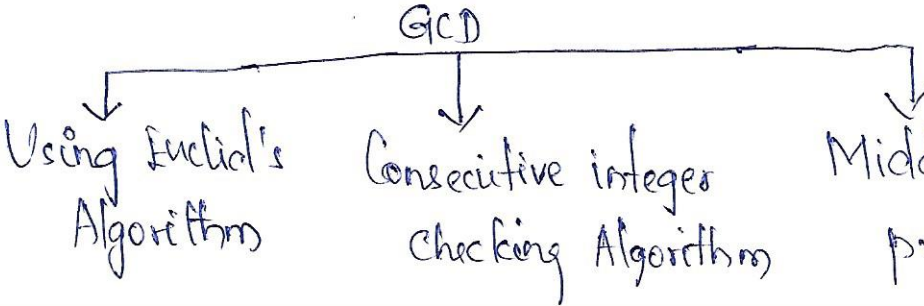
```
for variable ← value1 to value2 do
{
    statement1
    statement2
      :
    statementn
}
```

Example

Greatest Common Divisor (GCD)

*) The GCD of two non-zero numbers a and b is basically the largest integer that divides both a & b evenly ie) with a remainder of zero.

*) The algorithm for finding GCD is implemented by a famous mathematician Euclid Alexandria.



① Euclid's algorithm for computing gcd(m,n)

Step1: If n=0 then return gcd(m,n) = m

Step2: Divide m by n and assign the remainder to r.

Step3: Assign the value of n to m, r to n. Go to step 1.

ALGORITHM Euclid(m,n)

// Computes gcd(m,n) by Euclid's algorithm

// Input : Two non negative, not both zero integers m & n

// Output : Greatest Common divisor of m & n.

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

eg gcd(60,24) [gcd(m,n) = gcd(n, m mod n)]

Step1 : m=60 , n=24.

Step2 :

$$r = 12.$$

$$24 \overline{)60}\ 2,\ 48,\ 12$$

Step3 : m=24, n=12.

gcd(24,12).

Step1: m=24 n=12

Step2 : r=0.

$$12 \overline{)24}\ 2,\ 24,\ 0$$

Step 3: $m = 12$, $n = 0$.

$gcd(12, 0)$

Here $n = 0$, so 12 is gcd.

Algorithm 2:

Consecutive integer checking algorithm for computing $gcd(m, n)$

Step 1: Assign the value $\min\{m, n\}$ to t.

Step 2: Divide m by t, if remainder = 0, then divide n by t, If remainder = 0 then return 't' as answer and stop.

Step 3: Decrease the t value by 1 Repeat Step 2.

ex $gcd(60, 24)$

Step 1: $m = 60$ $n = 24$.

$\min\{60, 24\} = 24$

$\therefore t = 24$.

Step 2:

$\frac{60}{24} \neq 0$.

Now $t = 24 - 1 = 23$.

$\frac{60}{23} \neq 0$. $t = 22$.

$\frac{60}{22} \neq 0$. . .

:

$t = 12$

$\frac{60}{12} = 0$.

$\therefore$ Gcd (60, 24) = 12.

Algorithm 3 :

Middle - School procedure

Step 1: Find the prime factor of m

Step 2: " " " " " n

Step 3: Identify the common factor in the two prime expansion.

Step 4: Compute the product of all common factors & return it as the gcd.

gcd (60, 24).

Step 1: $m = 60$.

2	60
2	30
3	15
5	5
	1

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

Step 2: $n = 24$

2	24
2	12 - 0
2	6 - 0
3	3 - 0
	1.

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

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

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

$\Rightarrow 2 \times 2 \times 3 = 12$

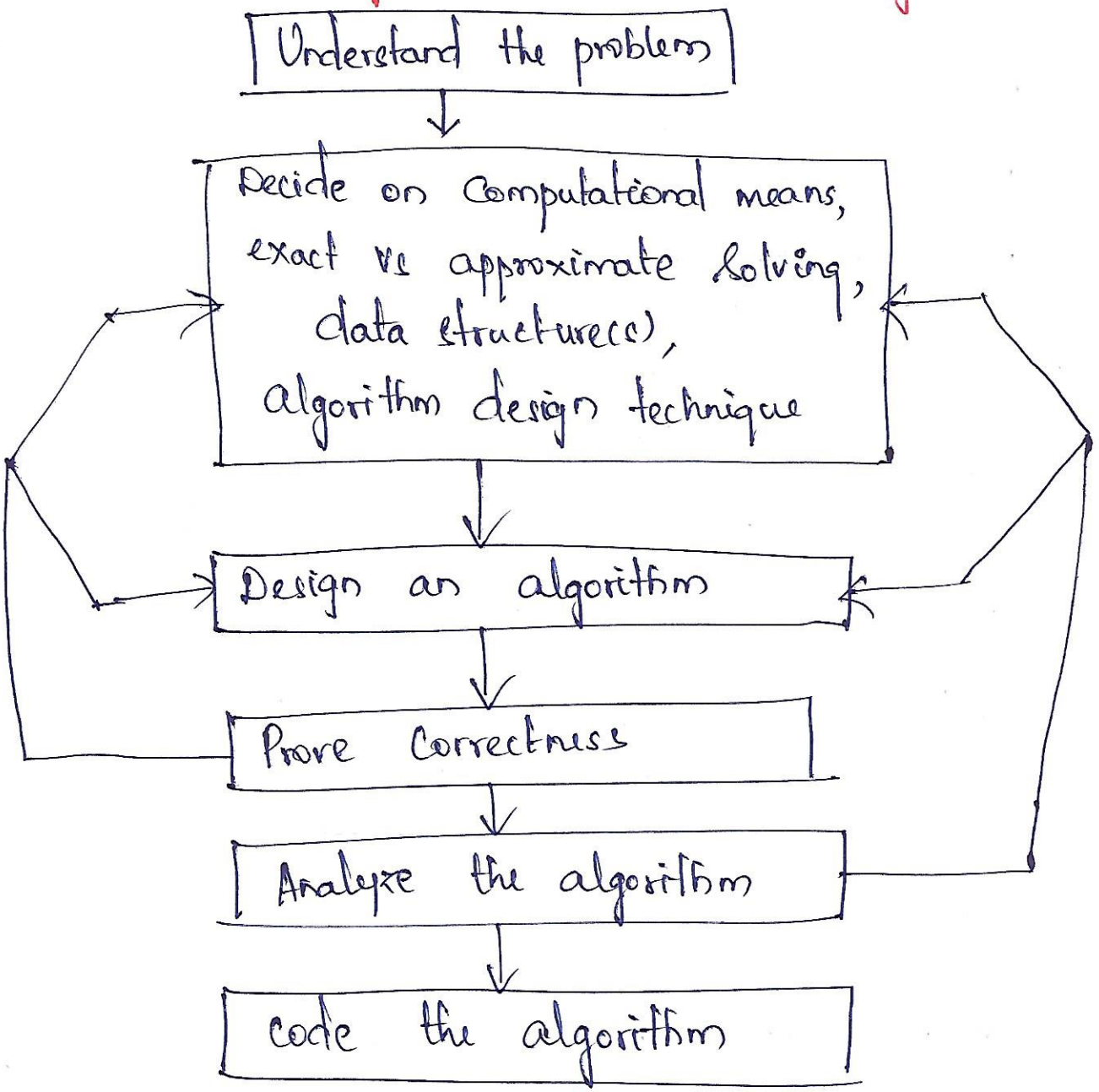


fig: Algorithm design and analysis process

1. Understanding the problem:

*) Understand the problem before designing an algorithm

*) Read the problem description carefully & ask questions if you have any doubts about the problem, solve few small examples by hand and think about special cases.

*) Select the known algorithm.

*) How the algorithm work with the particular problem. Find the strength and weakness of the particular algorithm.

*) If the algorithm is not available, then design your own algorithm.
→ After carefully understanding the problem statement find out what are the necessary I/P's for solving that problem.

*) An input to an algorithm specifies an instance of the problem the algorithm solves. Important to specify exactly the range of instances the algorithm needs to handle so that the boundary values of algorithm get fixed. The algorithm should work correctly for all valid inputs.

2. Decision making

a) Ascertaining the Capabilities of a Computational Device

*) Once you completely understand the problem, you need to ascertain the capabilities of the computational device the algorithm is intended for.

*) Based on computational device, classify the algm
→ Sequential Algorithm - Instruction are executed one after another. one operation at a time.
→ parallel Algorithm, Concurrently or parallel (RAM) execution

b) Choosing between exact and approximate problem solving:

*) Solving the problem exactly is called exact problem

Solving eg) Sorting.

*) Solving the problem approximately is called approximation algorithm. eg) Integral, [illegible], TSP.

Deciding on Appropriate Data Structures

*) To design a better program, use data structure

Concept. Data structure + algorithm work together & these are interdependent.

Algorithms + Data structures = Programs.

Algorithm Design Techniques

*) General approach to solve the problem algorithmically and that must be suitable for all types of inputs. It should produce desired output in a finite time

Techniques: Brute force, Divide & Conquer, Dynamic programming, Greedy technique, Backtracking

Methods of Specifying an Algorithm

*) Natural language - Can be understood by all types of users.

*) Pseudocode - Combination of natural & programming language

*) Flowchart.

↓

Pictorial representation of the program

Not suitable for lager program

eg) Addition of two nos.

Prove Correctness of an algorithm

*) A correct algorithm is not only works most of the time but one that work correctly for all types of inputs.

*) The algorithm yields required result for all possible input in a finite amount of time. For all input it should produce desired output.

*) Prove the correctness using mathematical induction.

Analyzing an algorithm

*) Time efficiency - How fast the algorithm works.

*) Space " - How much extra memory the algorithm needs.

*) Simplicity - Generating sequence of instructions which are easy to understand.

*) Generality - One algorithm must solve all kinds of problems.

Coding an algorithm

*) Convert algorithm to program using any one of high-level language.

Testing - While implementing algorithm as program, test the following characteristics.

→ Range of i/p - Whether giving correct input.

→ Optimizing the code - reduce the unnecessary code & make standardized.

→ Redesign the algorithm - modification can be done in any step

→ Analyse the o/p - Getting correct o/p or not for all possible combination of input.

Notion of Algorithm:

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

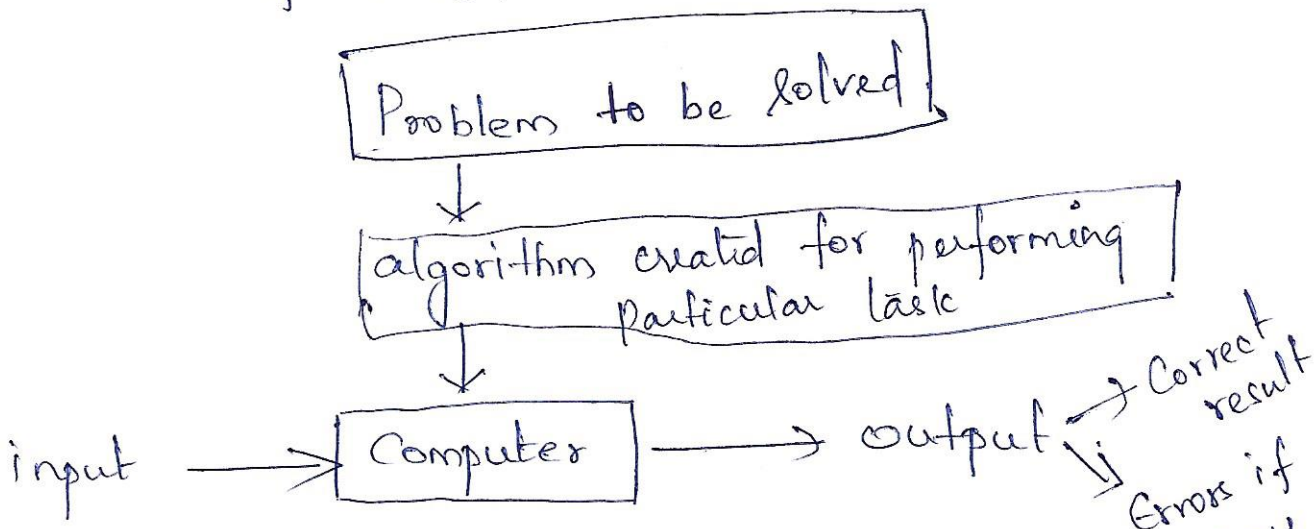
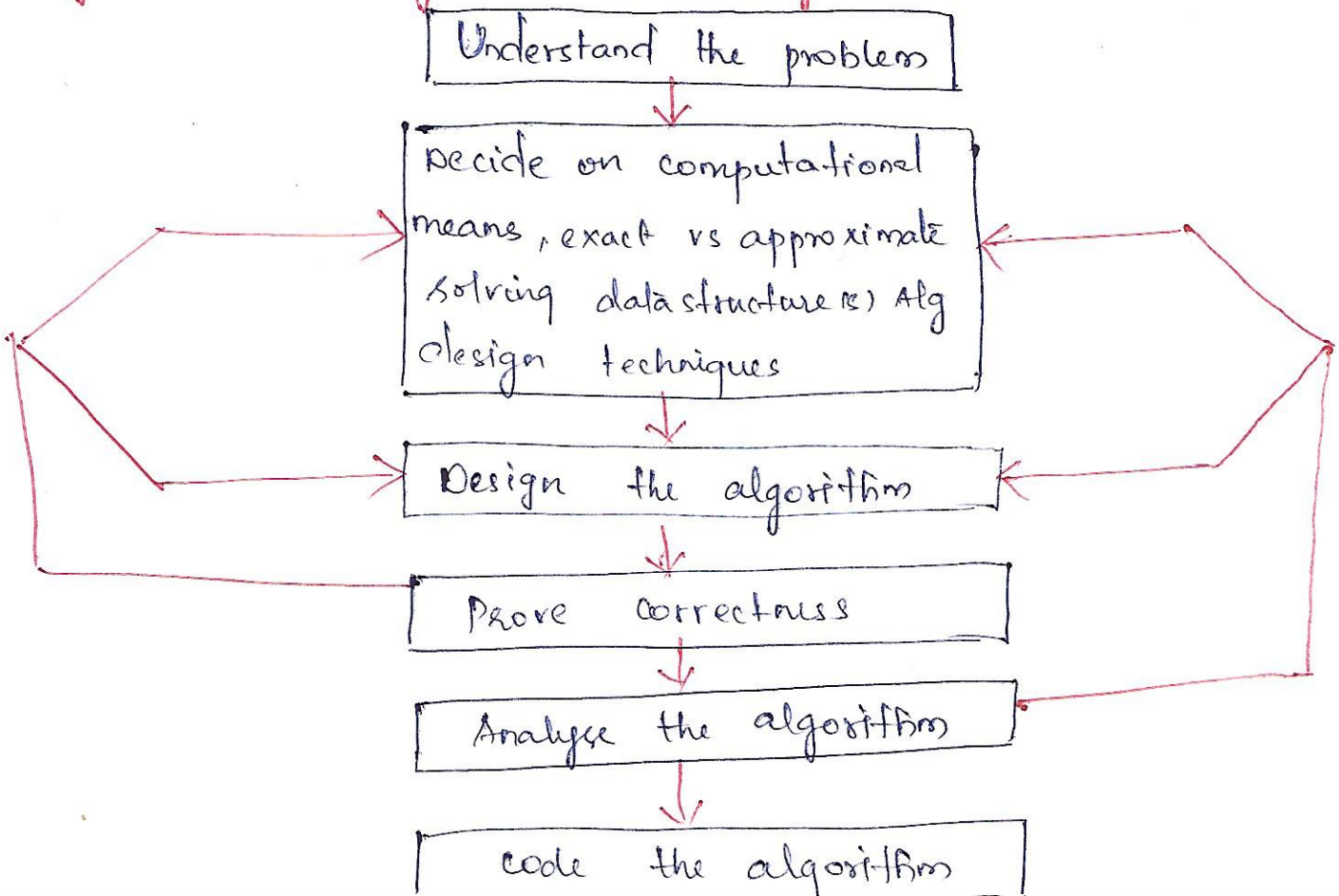


fig: Notion of algorithm

Algorithm Design and Analysis Process



Fundamentals of the analysis of algorithm efficiency

Analysis Framework:

There are two kinds of efficiency

→ Time efficiency - indicates how fast an algorithm in question runs.

→ Space efficiency - deals with the extra space the algorithm requires.

Measuring an input size

Pbm	Input size Measure
Searching	No of list's items (i.e. n)
Multiplication of 2 matrices	Matrix dimensions or total no of elements
Typical graph pbm	No of vertices and edges
GCD of 2 nos	Two nos

→ An algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.

→ In most cases, selecting such a parameter is quite straight forward.

→ For eg, it will be the size of the list for problems of sorting, searching, finding list's smallest element and most other problems dealing with lists.

→ for the problem of evaluating a polynomial $p(x) = a_n x^n + \dots + a_0$ of degree n, it will

be the polynomial's degree or the no of its coefficients, which is larger by one than its degree.

→ There are situations, ofcourse where the choice of a parameter indicating an input size does matter.

eg) Computing the product of two nxn matrices

→ there are two natural measures of size for this pbm.

*) The matrix order n

*) The total no of elements N in the matrices being multiplied.

→ Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of the two measures we use.

→ The choice of an appropriate size metric can be influenced by operations of the algm in question. For eg, how should we measure an i/p's size for a spell. checking algm? If the algm examines individual characters of its i/p, then we should measure the size by the no of characters, if it works by processing

words, we should count their number in the input.

→ We should make a special note about measuring size of i/p's for algms involving properties of numbers (eg. checking whether a gn integer n is prime).

→ for such algorithms, computer scientists prefer measuring size by the number b of bits in the n's binary representation

$$b = \lfloor \log_2 n \rfloor + 1$$

→ This metric usually gives a better idea about efficiency of algorithms in question.

Units for measuring runtime:

→ We can simply use some standard unit of time measurement - a second, a millisecond, & so on - to measure the running time of a program implementing the algm.

→ There are obvious drawbacks to such an approach. They are

A) Dependence on the speed of a particular computer.

*) Dependence on the quality of a program implementing the algorithm.

*) The compiler used in generating the machine code.

→ Since we are in need to measure algorithm efficiency we should have a metric that does not depend on these extraneous factors.

→ One possible approach is to count the no of times each of the algorithm's operations is executed. This approach is both difficult & unnecessary.

→ The main objective is to identify the most important operation of the algorithm called the basic operation, the operation contributing the most to the total running time and compute the no of times the basic operation is executed.

→ As a rule, it is not difficult to identify the basic operation of an algorithm.

Running time of basic operation $T(n) \simeq C_{op}\, C(n)$ ← No of times the operation needs to be executed
Execution time of basic operation

Worst, Best & Average cases Efficiences:

Example:
Sum(int a[], int
int i, res=0;
for(i=1;i<=
i++)
res=res+a[i];
}
Basic op: +addition
I/p size: n, no of
items in the
array
$T(n) = C(n) = n$

→ It is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algm's i/p.

→ But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular i/p.

→ eg) sequential search. This is a straight forward algm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

Algorithm sequential search (A[0...n-1], K)

// Searches for a given value in a given array
// by sequential search
// Input: An array A[0...n-1] and a search key K
// Output: Returns the index of the first element of
// A that matches K or -1 if there are no matching
// elements.

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

*) clearly, the running time of this algorithm can be quite different for the same list size n.

Worst case efficiency:

*) The worst-case efficiency of an algm is its efficiency for the worst-case i/p of size n, which is an i/p (or i/ps) of size n for which the algm runs the longest among all possible i/ps of that size.

*) In the worst case, when there are no matching elements / the first matching element happens to be the last one on the list, the algorithm makes the largest no of key comparisons among all possible inputs of size n

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

*) The way to determine is quite straightforward.

*) To analyze the algm to see what kind of i/p yield the largest value of the basic operation's count C(n) among all possible i/ps of size n & then compute this worst case value $C_{worst}(n)$.

*) The worst-case analysis provides very important information about an algm's efficiency by bounding

its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed Cworst(n) its running time on the worst-case i/ps.

Best case Efficiency

*) The best-case efficiency of an algorithm is its efficiency for the best-case i/p of size n, which is an i/p (or i/p's) of size n for which the algm runs the fastest among all possible i/p's of that size.

*) We can analyze the best case efficiency as follows.

→ First, determine the kind of i/p's for which the count C(n) will be the smallest among all possible i/p's of size n. (Note that the best case does not mean the smallest i/p, it means the i/p of size n for which the algm runs the fastest).

→ Then ascertain the value of C(n) on these most convenient inputs.

→ Eg for Sequential Search, best case i/p's

inputs will be lists of size n with their first elements equal to a search key, accordingly $C_{best}(n) = 1$

→ The analysis of the best case efficiency is not nearly as important as that of the worst-case efficiency.

→ But it is not completely useless. For eg, there is a sorting algm (insertion sort) for which the best-case i/p's are already sorted arrays on which the algm works very fast.

→ Thus, such an algm might well be the method of choice for application dealing with almost sorted arrays. And, of course, if the best case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis.

Average Case Efficiency

(algm - algorithm)

*) It yields the information about an algm about algm's behaviour on a "typical" & "random" input.

*) To analyze the algm's average-case efficiency, we must make some assumptions about possible i/p's of size n.

*) The investigation of the average case efficiency is considerably more difficult than investigation of the worst case & best case efficiency.

*) It involves dividing all instances of size n into several classes so that for each instance of the class the no of times the algorithm's basic operation is executed is the same.

*) Then a probability distribution of i/p's needs to be obtained / assumed so that the expected value of the basic operation's count can then be derived.

The average number of key comparisions $C_{avg}(n)$ can be computed as follows

*) let us consider sequential search. The std assumptions are,

*) In the case of successful search, the probability of the first match occurring in the i^{th} position of the list is p in for every i, & the no of comparisions made by the algm in such a situation is obviously i.

*) In the case of an unsuccessful search, the no of comparisons is n with the probability of such a search being (1-p).

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

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

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

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

eg) if $p=1$ (i.e the search must be successful), the average no of key comparisons made by sequential search is $(n+1)/2$.

if $p=0$ (i.e) the search must be unsuccessful), the average no of key comparisons will be n because the algorithm will inspect all n elements on all such i/ps.

Orders of growth

Common asymptotic functions

Function	Name	Description.
1	constant	Independent of i/p size.
$\log n$	Logarithmic	Get slightly slower as n grows
n	linear	is optimal if u need to process n i/ps.
$n \log n$	$n \log n$	scales to huge pbms.
n^2	Quadratic	use only on relatively small problems
n^3	Cubic	
2^n	Exponential	Not usually practical
$n!$	Factorial	

→ All polynomials of order k are $O(N^k)$

→ Each comparision based sorting algorithm is $\Omega(N \log N)$

Measuring runtime (Contd..)

① Addition of array no

$T(n) = n$.

$T(2n) = 2n$.

$\therefore \frac{T(2n)}{T(n)} = \frac{2n}{n} = 2$.

② Matrix multiplication

$T(n) = n^3$

$T(2n) = (2n)^3$

$\frac{T(2n)}{T(n)} = \frac{8n^3}{n^3} = 8$.

This algm taken 8 times longer time if i/p is doubled.

→ efficiency analysis framework ignores multiplicative constants & concentrates on the counts order of growth.

Time complexity of an algorithm is generally classified as 3 types.

They are 1) worst case 2) Average case 3) Best case.

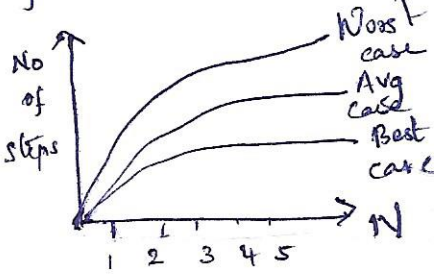
1. Worst case

→ It is the longest time that an algorithm will use over all instances of size n for a given problem to produce a desired results.

→ This can be represented by a function

$$f(n) = n^2 \text{ (or) } f(n) = n \log n$$

$$T(n) = O(f(n))$$



→ This algm takes more than f(n) operations.

2. Average Case

→ It is the average time that the algorithm will use over all instance of size n for a given problem.

→ It depends on the probability distribution of instances of the problem.

3. Best case

→ It is the shortest time that the algm will use over all instances of size n for a given problem

to produce a desired result.

$$T(n) = \Omega(f(n))$$

Space Complexity

→ Space complexity of a program is the amount of memory consumed by the algorithm until it completes its execution.

→ The way in which the amount of storage space required by an algorithm varies with the size of the problem to be solved

Instruction space: compiled version of the program

Data space: all constant + variables values

environment stack space: information to resume execution of partially completed fns.

Types of memory

Program uses 2 types of memory such as

1. Fixed amount of memory
2. Variable amount of memory

→ Total space needed by an algorithm can be simply divided into two parts from the 3 components of space complexity.

Fixed amount of memory

A fixed amount of memory occupied by the space for the program code and variables used in the program.

Variable amount of memory

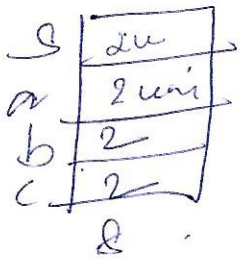
A variable amount of mly occupied by

by the component variable, whose size is dependent on the problem being solved. (15)

*) This space increase (or) decreases depending upon whether the program uses interactive / recursive procedures.

eg1 To find the addition of 3 integer numbers.

```
Void fun1()
{
    int a,b,c,s;
    s=a+b+c;
    Print "sum: ",s
}
```



The space required by this algorithm

a = 2 units
b = 2 units
c = 2 units
s = 2 units.

8 units

Algm requires 8 units of mly.

eg

```
int ary1[n];
int ary2[n];
for(int i=0; i<n; i++)
    ary2[i] = ary1[(n-1)-i];
```

space = 2n
Time = n.

```
int ary1[n];
for(int i=0; i<n/2; i++)
{
swap(&ary[i],
        &ary1[(n-1)-i]);
}
swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

space = n+1 (for temp) → array.
Time = 3·n/2.

eg2 To find the factorial of a given number

```
Void fact()
{
    n=5;
    f=1;
```

```
while (x <= n)
{
    F = x * F;
    x = x + 1;
}
}
```

The space required by this algorithm

x = 2 units
f = 2 "
n = 2 "
6 units.

This algm requires 6 units of mly.

→ Practically time and space complexity can be reduced only to certain levels, the reduction of time increases the space & vice versa. This is known as time space trade off.

eg

(1)

```
Sum1()
    Integer x, y, z; // declaration stmt
    Read x, y;
    z = x + y;
    Print "Sum of x & y", z
End sum1()
```

Executable stmt

Runtime for 3 executable stmt = 3 units

(2)

```
Sum2()
    Integer x, y, z
    Read x;
    Read y;
    z = x + y;
    Print "The sum of x and y is", z
End sum2()
```

4

(3)

```
Void display()
{ int i = 1; // decla.
  while (i <= n)      → n+1 times.
  { print i; i = i + 1; }   2^n time.
}
```

etc.

n+1+2n = 3n+1

(16)

Time Space tradeoff

*) Efficiency of an algorithm denotes the rate at which an algorithm solves a problem size n.

*) It is measured by the amount of resources it uses, the time and the space.

*) An algorithm's complexity is measured by calculating the time taken & space required for performing the algorithm.

*) The input size denoted by 'n' is one parameter used to characterize the instance of the problem.

*) The input size 'n' is the number of registers to hold the input.

[Refer Sivanakumar book for eg]

Time Complexity

*) Time complexity of an algorithm is the amount of time needed by a program to complete its task.

*) The way in which the number of steps required by an algorithm varies with the size of the problem.

*) The time taken for an algorithm is

Comprised of 2 times

1) Compilation time

2) Run time.

1) Compilation time

Compilation is the time taken to compile an algorithm.

→ while compiling, it checks for the Syntax and Semantic errors in the program

→ It takes constant time to compile the particular prog.

→ Compilation time is always dependent upon the compiler, since different compilers can take different times to compile the same program.

2) Run time

— It is the time to execute the compiled program.

→ It depends upon the no of instructions present in the algorithm.

→ Usually we consider one unit for executing one instruction.

→ It is calculated only for executable statements & not for declared statements.

Time Complexity — frequency count.

```
for (i=0; i<n; i++)
{
  for (j=0; j<n; j++)
  {
    c[i][j] = a[i][j] + b[i][j];
  }
}
```

many factors
- → Sly load
- → No of other pgms running
- → Instn set used
- → speed of hardw.

frequency count → denoting no of times of execution of stmt.

*) i=0 executes once. frequency count = 1

*) i<n executes for n+1 times

*) i++ executes for n times.

*) j=0 executes for

$n \times (1) = n$ times

↓ for outer loop; ↑ for initialization of j

*) j<n executes for

$n \times (n+1) = n^2 + n$ times

↓ execution of outer i loop; ↓ execution of inner j loop.

*) j++ executes for

$n \times n = n^2$ times

↓ execution of outer i loop; ↳ execution of inner j loop.

*) c[i][j] = a[i][j] + b[i][j] executes for $n \times n = n^2$ times

eg

```
for(i=0; i<n; i++)
{
  sum = sum + a[i];
}
```

i=0	1
i<n	n+1
i++	n.
sum=sum+a[i]	n.
Total.	3n+2.

O(n)

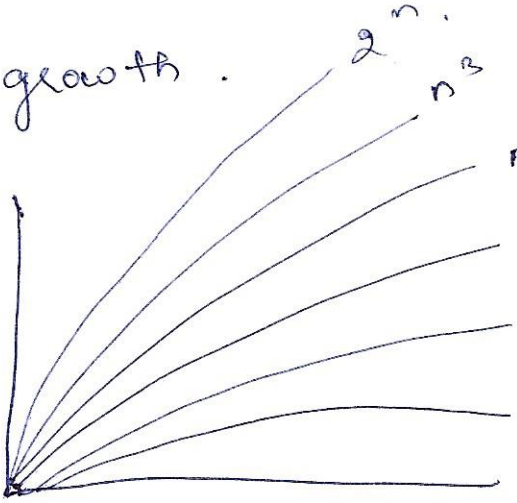
$1+n+1+n+n+n^2+n+n^2+n^2 = 3n^2+4n+2$

Hence frequency count = $3n^2+4n+2$

If the constants are neglected then the time complexity = $O(n^2)$

Order of growth.

→ Measuring the performance of an algm in relation with the i/p size n is called Order of growth.



logarithmic fn → slowest growing fn.

exponential fn → fastest & grows rapidly with varying i/p size n.

Algm with a time Complexity n is more efficient than one with a time Complexity of n^2

Two arrays

Time complexity – n times

Space – n + n. = 2n.

int arr1[n]; n + 1.

lc = n/2. n/2.

i = 0 i < n/2.

When n = 1

①	T	T	S	2
②	T	2	S	4
③		3	S	6

When n = 1.

n=1 S 2 T = 0.5

n=2 S 3. T = 1.5.

n=3 S 4 T = 2.

Basic Operation: the addition of an item in the array to sum.

I/p size : n, the no of items in the array.

Regardless of the values of the numbers in the array, there are n passes through the for loop.

∴ Basic operation is always done n times.

$$T(n) = n$$

Exchange Sort

In the case of an algorithm that works by comparing keys, we can consider the comparison instruction or the assignment instruction as the basic operation. We will analyze the no of comparisons here.

Basic Operation: The comparison of S[j] with S[i].

i/p size: n, the number of items to be sorted.

→ Determine how many passes there are through the for-j loop.

→ For a given 'n' there are always n-1 passes through the for-i loop.

→ In the first pass through the for-i loop, there are n-1 passes through the for-j loop in the second pass there are n-2 passes through

the for-j loop, in the third pass there are n-3 passes through the for-j loop, ... and in the last pass there is one pass through the for-j loop.

∴ Total no of passes through the for-j loop is given by

$$T(n) = (n-1)+(n-2)+(n-3)+\ldots+1 = \frac{n(n-1)}{2}$$

Matrix Multiplication

→ The only instruction in the innermost for loop is the one that does a multiplication & an addition.

→ It is not hard to see that the algorithm can be implemented in such a way that fewer additions are done than multiplication. Therefore, we will consider only the multiplication instruction to be the basic operation.

Basic Operation : Multiplication instruction in the innermost for loop.

Input size: n, the no of rows & columns.

There are always n passes through the for-i loop, in each pass there are always n passes through the for-j loop, & in each pass through the for-j loop there are

always n passes through the for-k loop. Because the basic operation is inside the for-k loop. (20)

$$T(n) = n \times n \times n = n^3.$$

Worst-case Time Complexity (Sequential Search)

Basic Operation: The comparison of an item in the array with x.

Input size: n, the number of items in the array.

The basic operation is done at most n times, which is the case if x is the last item in the array or if x is not in the array.

$$W(n) = n$$

Average-case Time Complexity (Sequential Search)

Basic operation: the comparison of an item in the array with x.

Input size: n, the number of items in the array.

$$A(n) = \sum_{k=1}^{n} (k \times 1/n) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

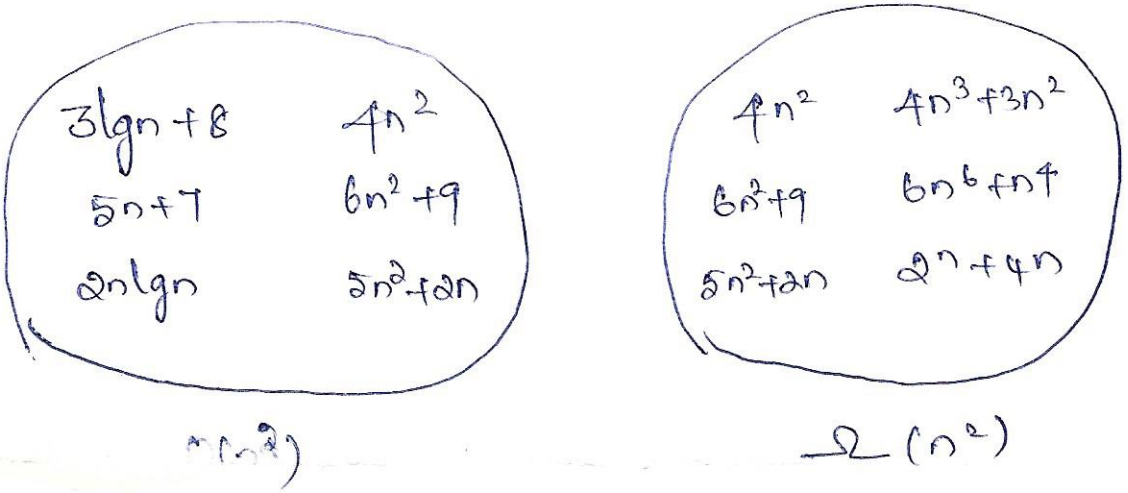
Best-case Time Complexity (Sequential Search)

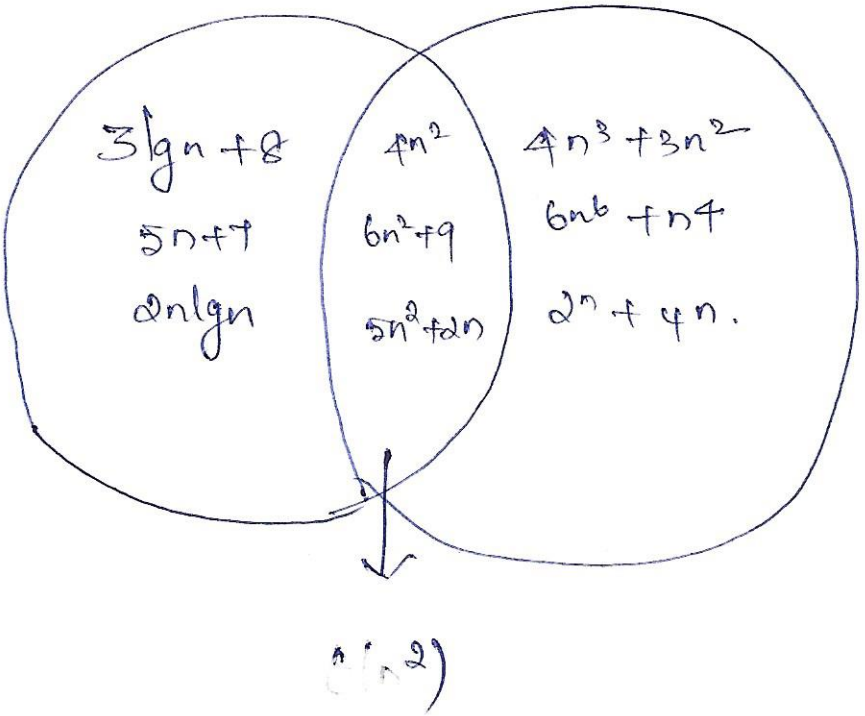
Basic Operation: the comparison of an item in the array with x.

Input size: n, the number of items in the array.

Because $n \geq 1$, there must be at least one pass through the loop, if $x = S[1]$ there will be one pass through the loop regardless of the size of n.

$\therefore$ $B(n) = 1$





$$\Theta(n^2) = O(n^2) \cap \Omega(n^2)$$

Mathematical Analysis of Recursive Algorithms

General plan for Analyzing Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the no of times the basic operation is executed
4. Set up a recurrence relation, with an appropriate initial condition, for the no of times the basic operation is executed.
5. Solve the recurrence or at least ascertain the order of growth of its solution.

eg

```
ALGORITHM F(n)
// computes n! recursively
// I/p - A nonnegative integer n.
// o/p - Value of n!
if n=0 return 1
else return F(n-1) * n.
```

I/p — n.

B.O — Multiplication.

$M(n) = M(n-1) + 1$ for $n > 0$.

to Compute $F(n-1)$ * to multiply $F(n-1)$ by n.

if $n=0$ return 1

$M(0) = 0$ ← no multiplications when $n=0$.

↑ Call stops when $n=0$

$\therefore M(n) = M(n-1) + 1$ for $n > 0$.

$M(0) = 0$.

Analysis

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

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

$= M(n-3) + 3$

$\vdots$

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

$n=0$.

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

$= M(n-n) + n$

$= n$

$M(n) = O(n)$

Examples of Recursive
Tower of Hanoi
Binary representation

Mathematical Analysis of Non-Recursive Algorithms

General plan

1) Decide the input size based on parameter n.
2) Identify algorithm's basic operation(s)
3) Check how many times the basic operation is executed. Then find whether the execution of basic operation depends upon the input size n. If the basic operation depends upon worst case, average case & best case then that has to be analyzed separately.
4) Set up a sum for the number of times the basic operation is executed.
5) Simply the sum using standard formula and rules

```
ALGORITHM MatrixMultiplication(A[0...n-1, 0...n-1],
                               B[0...n-1, 0...n-1])
// Multiplies two n-by-n matrices.
// 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
```

Mathematical Analysis

1) Input size is simply order of matrices i.e. n.

2) Basic Operation is in the innermost loop i.e. Multiplication & addition.

```
C[i,j] ← C[i,j] + A[i,k] * B[k,j]
```

3) Basic operation depends only upon input size. There are no best case, average case and worst case efficiencies.

4) Total no of multiplications M(n) executed by the algorithm

Set up a sum for the

5) The sum can be denoted by M(n)

M(n) = Outermost loop x Inner loop x Innermost loop (1 execution)

= for i loop x for j loop x for k loop.

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

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n. \qquad \left[\sum_{j=0}^{n-1} = n \right]$$

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

$M(n) = n^3$

∴ Simplified sum is n^3.

Time complexity of matrix multiplication $\Theta(n^3)$.

MaxElement

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

I/p → n. (no of elements).

B.O → Comparision.

B.O → depends only on i/p size n.

C(n) → no of times this comparision is executed & find the formula.

One comp on each execution

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

Examples

i) Element uniqueness ⇒ binary digit.

Sequence

A Sequence is an ordered list of nos.

eg 2, 4, 6, 8, 10

→ Sequence is denoted by a letter (x or a) with a subindex (such as n or i) written in curly brackets eg $\{x_n\}$ or $x(n)$.

$x(n)$ is the generic term of the sequence.

define a sequence

1) $x(n) = 2n$ for $n \geq 0$.

or

2) $x(n) = x(n-1) + n$ for $n > 0$. (B1)

$x(0) = 0$ (B2)

eqn B1 → Recurrence eqn / Recurrence relation (recurrence).

B2 → initial condition.

↓

given for a value of n other than 0..

Linear Second-order recurrences with constant coefficients

Recurrence Relation or Equation.

A homogeneous recurrence equation is written as

$$a_0 t_n + a_1 t_{n-1} + \dots + a_k t_{n-k} = 0.$$

How to solve the above equation.

Solution Technique

Step 1:

Set up a corresponding characteristic equation

$$a_0 x^n + a_1 x^{n-1} + \dots + a_k x^{n-k} = 0 \quad [\text{Replace } t_n \text{ as } x^n]$$

$\div x^{n-k}$.

$$a_0 x^k + a_1 x^{k-1} + \dots + a_k = 0 \quad [\text{for } x \neq 0]$$

Step 2:

Solve the characteristic equation as a polynomial equation. Say, the real roots are r_1, r_2, r_k.

Note there are k solutions for kth order polynomial equation.

Step 3

The general solution for the original recurrence equation is

$$t_n = \sum_{i=1}^{k} c_i r_i^n \quad \text{if roots are distinct}$$

r_1, r_2 are different real
$x(n) = \alpha r_1^n + \beta r_2^n$.
r_1, r_2 are equal
$x(n) = \alpha r^n + \beta n r^n$

$r_{1,2} = u \pm iv$
$x(n) = r^n [\alpha \cos n\theta + \beta \sin n\theta]$
$r = \sqrt{u^2 + v^2}$ $\theta = \arctan \frac{v}{u}$

$$T(n) = \sum_{i=1}^{p} c_i n^{i-1} r_1^n + \sum_{i=p+1}^{k} c_i r_i^n .$$

first p roots are equal to r_1.

Step 4:

Using initial conditions (if available) solve for the coefficients in above equation inorder to find the particular solution.

Problems

① $t_n - 3t_{n-1} - 4t_{n-2} = 0$ for $n \geq 2$.

{Initial condition $t_0 = 0$, $t_1 = 1$}.

Step 1:

characteristic equation

$$x^n - 3x^{n-1} - 4x^{n-2} = 0 \quad \text{—①}$$

Step 2

x^{n-2} in eqn ①.

$$x^2 - 3x - 4 = 0 .$$

$$(x+1)(x-4) = 0 .$$

$$x = -1 \;, \; x = 4 .$$

Roots are $r_1 = -1$ $r_2 = 4$.

Step 3:

General solution.

$$T(n) = c_1(-1)^n + c_2(4)^n \quad \text{--- (2)}$$

$\left[\text{i.e. } t_n = \sum_{i=1}^{k} c_i r_i^n \right]$

where $k = 2$ roots.

$$t_n = \sum_{i=1}^{2} c_i r_i^n.$$

$$t_n = c_1 r_1^n + c_2 r_2^n$$

Step 4:

Initial condition $t_0 = 0, t_1 = 1$.

sub $t_0 = 0$ in eqn (2).

Sol

$$T(0) = 0.$$

$$T(0) = c_1(-1)^0 + c_2(4)^0 = 0.$$

$$c_1 + c_2 = 0 \quad \text{--- (3)}.$$

Sol

$$T(1) = 1$$

$$T(1) = c_1(-1)^1 + c_2(4)^1 = 1.$$

$$-c_1 + 4c_2 = 1 \quad \text{--- (4)}.$$

solve (3) & (4).

$$c_1 + c_2 = 0$$

$$-c_1 + 4c_2 = 1$$

$$5c_2 = 1$$

$$c_2 = 1/5$$

$$c_1 = -1/5.$$

Step 5

Particular solution.

sub $c_1 = -1/5$ & $c_2 = 1/5$ in eqn (2)

$$T(n) = -1/5\,(-1)^n + 1/5\,(4)^n.$$

$$T(n) = \Theta(4^n).$$

Inhomogeneous recurrence equation

$$a_0 t_n + a_1 t_{n-1} + \dots + a_k t_{n-k} = b_1^n P_1(n) + b_2^n P_2(n) + \dots$$

where a_i's are constant.

each $P_i(n)$ is a polynomial of order n.

Soln Technique. Here, the characteristic eqn is

$$(a_0 x^k + a_1 x^{k-1} + \dots + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \dots = 0$$

$$\Rightarrow a_0 x^k + a_1 x^{k-1} + \dots a_k = 0$$

$$(x - b_1)^{d_1+1} = 0$$

$$a_n (x - b_2)^{d_2+1} = 0$$

Step 1: Homogenize the given equation to form equivalent homogeneous recurrence equation

Step 2: Though (3) or (4) are the same as in the case of solving homogeneous recurrence equation.

Problems

(1) $t_n - 2t_{n-1} = 3^n$:- or $T(n) - 2T(n-1) = 3^n$.

[Note: special case with $P(n) = 1$, polynomial of 0th order, there is no initial condition, so we get only the general soln]

Step 1

Given eqn.

$$t_n - 2t_{n-1} = 3^n \quad \text{---} (1).$$

Replace $n \to n+1$ in eqn (1).

$$t_{n+1} - 2t_{n+1-1} = 3^{n+1}$$

$$t_{n+1} - 2t_n = 3^{n+1} \quad \text{---} (2).$$

$3^n + 3^1$

3^{n+1}

xly eqn (1) by 3 on both sides.

$$3t_n - 6t_{n-1} = 3^n \cdot 3^1$$

$$3t_n - 6t_{n-1} = 3^{n+1} \quad \text{---} (3).$$

subtract (3) − (2).

$$3t_n - 6t_{n-1} - t_{n+1} + 2t_n = 3^{n+1} - 3^{n+1}$$

$$5t_n - 6t_{n-1} - t_{n+1} = 0.$$

$$\boxed{t_{n+1} - 5t_n + 6t_{n-1} = 0}$$

[This is a homogeneous recurrence equation which is equivalent to the given inhomogeneous equation].

Step 2:

Characteristic eqn.

$$x^{n+1} - 5x^n + 6x^{n-1} = 0.$$

$\div x^{n-1}$.

$x^2 - 5x + 6 = 0$.

$(x-2)(x-3) = 0$.

The roots are $r_1 = 2$, $r_2 = 3$.

Step 3:

General solution.

$$\boxed{t_n = c_1(2)^n + c_2(3)^n} = \Theta(3^n).$$

(2) $t_n - 2t_{n-1} = n$.

Step 1:

~~characteristic eqn~~ $t_n - 2t_{n-1} = n$ — (1).

~~homogeneous~~.

Replace $n \to n+1$ in eqn (1).

$t_{n+1} - 2t_{n+1-1} = n+1$.

$t_{n+1} - 2t_n = n+1$ — (2).

Sub (2) − (1)

$t_{n+1} - 2t_n - t_n + 2t_{n-1} = n+1-n$.

$t_{n+1} - 3t_n + 2t_{n-1} = 1$ — (3).

Still not a homogeneous equation

Second stage of homogeneous.

Replace $n \to n+1$ in eqn (3).

$$t_{n+1+1} - 3t_{n+1} + 2t_{n+1-1} = 1 \quad \text{---} (4).$$

$$t_{n+2} - 3t_{n+1} + 2t_n = 1 \quad \text{---} (4)$$

Sub (4) - (3)

$$t_{n+2} - 2t_{n+1} + 2t_n - t_{n+1} + 3t_n - 2t_{n-1} = 1 - 1$$

$$t_{n+2} - 4t_{n+1} + 5t_n - 2t_{n-1} = 0.$$

Now it is a homogeneous recurrence equation and one can solve it in the usual way.

Step 2:

Characteristic eqn.

$$x^{n+2} - 4x^{n+1} + 5x^n - 2x^{n-1} = 0.$$

Step 3 $\div x^{n-1}$

$$x^1 - 4x^2 + 5x - 2 = 0.$$

$$-4x^2 + 6x - 2 = 0.$$

$$4x^2 - 6x + 2 = 0.$$

$$(4x - 2)(x - 1) = 0.$$

$$x = 1/2. \quad x = 1.$$

Step 4:

General soln.

$$t_n = (1/2)^n c_1 + c_2 (1)^n.$$

For problem 1 in intomogeneous if the initial condition is given as $T(0)=0$.

$t_n - 2t_{n-1} = 3^n$ — ① $T(0)=0$.

General soln.

$$t_n = c_1(2)^n + c_2(3)^n \cdot \text{—— ②}$$

since $T(0)=0$,

Sub $n=0$ in eqn ①.

$$T(0) - 2T(0-1) = 3^0.$$

$$T(1) - 2T(0) = 3.$$

$$T(1) - 2\times 0 = 3.$$

$$\boxed{T(1) = 3}$$

Sub $T(0)=0$ in eqn ②.

$$T(0) = c_1 + c_2 = 0. \text{ —— ③}$$

Sub $T(1)=3$ in eqn ②.

$$T(1) = c_1 2 + c_2 3 = 3 \text{ —— ④}.$$

solve ③ & ④.

$$c_1 + c_2 = 0.$$

$$2c_1 + 3c_2 = 0.$$

$$c_1 = -3, \quad c_2 = 3.$$

Particular soln $T(1) = (-3)(2)^n + 3\cdot 3^n = \Theta(3^n)$

(3) $T(n) - 2T(n-1) = 1$ subject to $T(0) = 0$. (18)

Soln. Here $P(n) = 1$

So inhomogeneous equation.

Step 1.

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

Replace $n \to n+1$ in eqn (1).

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

$$T(n+1) - 2T(n) = 1 \quad \text{---} (2).$$

sub (2) − (1).

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

$$T(n+1) - 3T(n) + 2T(n-1) = 0 \quad \text{---} (3).$$

Step 2:

Characteristic Equation.

$$x^{n+1} - 3x^n + 2x^{n-1} = 0. \quad \text{---} (4)$$

Step 3.

$\div x^{n-1}$ in eqn (4).

$$x^2 - 3x + 2 = 0.$$

$$(x-2)(x-1) = 0.$$

$$x = 2 \qquad x = 1$$

The roots are $r_1 = 2$, $r_2 = 1$

Step 4:

General solution.

$$\boxed{T(n) = c_1(1)^n + c_2(2)^n.}$$

Step 5:

An $T(0) = 0$.

So $T(1)$ will be.

sub $n=1$ in gn eqn (1).

$$T(1) - 2T(0) = 1.$$

$$\boxed{T(1) = 1}$$

$$T(0) = c_1 + c_2 = 0.$$

$$T(1) = c_1 + 2c_2 = 1.$$

$$\therefore \boxed{c_1 = -1, c_2 = 1}$$

Step 6:

The particular solution is

$$T(n) = (-1)(1)^n + 1(2)^n. = \Theta(2^n)$$

Note:

If the roots are $(x-2)^3(x-1)^3 = 0$.

roots are $2, 2, 2, 1, 1, 1$

$$\boxed{\text{General solution} = c_1 2^n + c_2 n 2^n + c_3 n^2 2^n + c_4(1)^n + c_5 n(1)^n + c_6 n^2(1)^n.}$$

Solving Recurrence Equations.

1. Substitution method
2. Master Method.

(1) Substitution Method.

→ In this method a guess for the solution is made.

→ There are two types.

1) Backward substitution
2) forward substitution.

Forward substitution method

In this method, using initial condition in the initial term and value for the next term is generated. This process is continued until some formula is guessed. Thus in this kind of substitution method, we use recurrence equations to generate the few terms.

eg Consider a recurrence equation.

$T(n) = T(n-1) + n$ with initial condition $T(0)=0$

Ref. $T(n) = T(n-1) + n$. —— (1).

If $n=1$ then $T(1) = T(1-1) + 1$.

$T(1) = 0 + 1 = 1$.

If $n=2$. then $T(2) = T(1) + 2$

$= 1 + 2 = 3$.

If $n=3$ then $T(3) = T(2) + 3$

$= 3 + 3 = 6.$

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

$$\boxed{T(n) = O(n^2)}$$

Backward substitution Method

→ Backward values are substituted recursively in order to derive some formula.

eg. Consider, a recurrence relation.

$$T(n) = T(n-1) + n \quad \text{— (1)} \qquad \text{initial condition } T(0) = 0$$

$n \to n-1.$

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

$$T(n-1) = T(n-2) + n - 1. \quad \text{— (2)}$$

sub $T(n-1)$ in eqn (1).

$$\boxed{T(n) = T(n-2) + (n-1) + n.} \quad \text{— (3)}$$

Let

$$T(n-2) = T(n-3) + (n-2) \quad \text{— (4)}$$

Sub $T(n-2)$ value in (3)

$$T(n) = T(n-3) + (n-2) + (n-1) + n.$$

$$\vdots$$

$$= T(n-k) + (n-k+1) + (n-k+2) + \ldots + n.$$

If $k = n$ then.

$$T(n) = T(0) + 1 + 2 + \ldots + n \qquad \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = O(n^2).$$

Solving recurrence equations by master's method

We can solve recurrence relation using a formula denoted by Master's method.

$T(n) = aT(n/b) + F(n)$ where $n \geq d$ & d is some constant

If $F(n)$ is $\Theta(n^d)$ where $d \geq 0$ in the recurrence relation then,

1. $T(n) = \Theta(n^d)$ if $a < b^d$
2. $T(n) = \Theta(n^d \log n)$ if $a = b$
3. $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$.

(1) $T(n) = 4T(n/2) + n$.

Soln $T(n) = aT(n/b) + F(n)$ by master theorem

From gn eqn.

$a = 4 \quad b = 2 \quad f(n) = n$. i.e) n^1 so $d = 1$

$a > b^d$.

i.e) $4 > 2^1$

$T(n) = \Theta(n^{\log_b a})$.

$T(n) = \Theta(n^{\log_2 4})$.

$T(n) = \Theta(n^{\log_2 2^2})$.

$T(n) = \Theta(n^2 \log_2^2) = \Theta(n^2)$.

(2) $T(n) = 9T(n/3) + n$.

$a=9 \quad b=3 \quad f(n)=n^1 \quad d=1$

$9 > 2^1$

$T(n) = \theta(n^{\log_3 9}) = \theta(n^{\log_3 3^2})$.

$= \theta(n^{2\log_3 3})$.

$\boxed{T(n) = \theta(n^2)}$

(3) $T(n) = 8T(n/2) + n^2$.

$a=8 \quad b=2 \quad f(n)=n^2 \quad d=2$.

$8 > 2^2$.

$T(n) = \theta(n^{\log_2 8})$.

$\boxed{T(n) = \theta(n^3)}$

(4) $T(n) = 9T(n/3) + n^3$

$a=9, \quad b=3 \quad f(n)=n^3 \quad d=3$.

$9 < 27$.

$T(n) = \theta(n^d) = \theta(n^3)$.

① Relationship between O, Θ and Ω

→ Big-oh provides an Asymptotic upper bound on a function.

→ Ω-notation provides an Asymptotic lower bound on a function.

→ Θ - notation bounds a function from upper and lower bound.

② $n! = \omega(2^n)$.

Let as consider $f(n) = n!$ —— ①

$g(n) = 2^n$. —— ②

The ω- notation condition is given by

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \quad \text{for } f(n) = \omega(g(n))$$

From ① & ②.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n!}{2^n}$$

$$= \lim_{n\to\infty} \frac{n(n-1)(n-2)(n-3)\ldots\ldots 3\cdot 2\cdot 1}{2^n}$$

$$= \lim_{n\to\infty} \frac{n^2\left[(1-1/n)(1-2/n)(1-3/n)\ldots 3/n \cdot \frac{2}{n}\cdot\frac{1}{n}\right]}{2^n}$$

$$\therefore \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \quad \text{—— ③}$$

From ③ The condition for the ω-notation is satisfied

$$f(n) = \omega(g(n))$$

$$\Rightarrow n! = \omega(2^n)$$

③ prove that $n! = O(4^n)$.

Let us consider $f(n) = n!$ & $g(n) = 4^n$.

The O-notation condition is given by

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0. \quad \text{for } f(n) = O(g(n))$$

$$\therefore \lim_{n\to\infty} \frac{n!}{4^n} = \lim_{n\to\infty} \frac{n(n-1)\ldots 3\cdot 2\cdot 1}{n^n}$$

$$= \lim_{n\to\infty} \frac{n\left[n(1-1/n)\cdot n(1-2/n)\cdot n(1-3/n)\ldots 3\cdot 2\cdot 1\right]}{n^n}$$

$$= \lim_{n\to\infty}$$

Pbms (27)

Solve the following recurrence relation $T(n) = T(n-1)+1$ with $T(0)=0$ as initial condition.

Soln

Let $T(n) = T(n-1)+1$ — (1)

By Backward substitution,

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

$T(n-1) = T(n-2)+1$ — (2)

Sub eqn (2) in eqn (1).

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

$T(n) = T(n-2)+2$ — (3)

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

$T(n-2) = T(n-3)+1$ — (4)

Sub eqn (4) in (3).

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

$T(n) = T(n-3)+3$

⋮

$\boxed{T(n) = T(n-k)+k}$ — (5)

If $k=n$ then eqn (5) becomes

$n-k=0$

$k=n$

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

$T(n) = 0+n$

$\boxed{T(n) = n}$

Homogenous Equation.

(1) Find the solution of $a_n = 4a_{n-1} - 3a_{n-2}$ with $a_1 = 0$ & $a_2 = 12$.

The characteristic eqn is

$$x_r^n = 4x^{n-1} - 3ax^{n-2}.$$

$$x^n - 4x^{n-1} + 3ax^{n-2} = 0.$$

$$x^{n-2}\left[x^2 - 4x + 3\right] = 0.$$

$$x^2 - 4x + 3 = 0.$$

$$(x-1)(x-3) = 0.$$

$$r_1 = 1 \quad r_2 = 3.$$

3
1 | 3

General solution is

$$T(n) = c_1 r_1^n + c_2 r_2^n.$$

$$\boxed{T(n) = c_1 (1)^n + c_2 (3)^n.}$$

~~$T(0) \equiv c_1 + c_2 = 0$ ——— (1)~~

$$T(1) = c_1 + 3c_2 = 0. \quad \text{——— (1)}$$

$$T(2) = c_1 + 9c_2 = 12 \quad \text{——— (2)}.$$

$$c_1 + 3c_2 = 0.$$

$$c_1 + 9c_2 = 12.$$

$$\boxed{c_2 = 2}$$

$c_1 + 3c_2 = 0$.

$c_1 + 6 = 0$

$\boxed{c_1 = -6}$.

The particular solution is

$\boxed{F(n) = -6 + 2(3)^n}$.

(2) Find the solution of $a_n = a_{n-1} + a_{n-2}$ with $a_0 = 0$ and $a_1 = 1$

The characteristic eqn is

$x^n - x^{n-1} - x^{n-2} = 0$.

$x^{n-2}\left[x^2 - x - 1\right] = 0$.

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

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

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

General soln.

$\boxed{t(n) = \left(\frac{1+\sqrt{5}}{2}\right)^n c_1 + \left(\frac{1-\sqrt{5}}{2}\right)^n c_2}$.

$t(0) = 0$.

$t(0) = c_1 + c_2 = 0$. ——— (1)

$t(1) = \left(\frac{1+\sqrt{5}}{2}\right) c_1 + \left(\frac{1-\sqrt{5}}{2}\right) c_2 = 1$. ——— (2)

1

1 | 1

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

$\frac{1 \pm \sqrt{1+4\times1}}{2}$.

$\frac{1 \pm \sqrt{5}}{2}$.

Solving eqn (1) & (2) we get. (34)

$$c_1\left[\frac{1+\sqrt{5}}{2}\right] - c_2\left[\frac{1-\sqrt{5}}{2}\right] = 1. \quad [\because c_2 = -c_1]$$

$$c_1\left[\frac{1+\sqrt{5}-1+\sqrt{5}}{2}\right] = 1.$$

$$c_1\left[\frac{2\sqrt{5}}{2}\right] = 1.$$

$$c_1 = 1/\sqrt{5} \quad . \quad c_2 = -1/\sqrt{5}$$

Particular Soln.

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

(3) $a_n = 4a_{n-1}$ with $a_1 = 20$.

The characteristic eqn is

$$x^n - 4x^{n-1} = 0.$$

$$x^{n-1}[x-4] = 0.$$

$$x = 4.$$

Gen. soln $T(n) = c_1(4)^n$.

Given $a_1 = 20$.

$T(1) = 20$.

$T(1) = c_1(4) = 20$.

$4c_1 = 20$

$\boxed{c_1 = 5}$

Particular soln

$\boxed{T(n) = 5\,(4)^n}$

(4) $a_n = 5a_{n-1} - 6a_{n-2}$ with $a_1 = 1$ and $a_2 = 5$.

Charact eqn.

$x^n - 5x^{n-1} + 6x^{n-2} = 0$.

$x^2 - 5x + 6 = 0$.

$(x-3)(x-2) = 0$.

$r_1 = 3 \quad r_2 = 2$.

General soln:

$\boxed{t(n) = c_1(3)^n + c_2(2)^n}$.

$t(1) = c_1 3 + 2c_2 = 1$. ——— (1).

$t(2) = c_1 9 + 4c_2 = 5$ ——— (2).

(1) $\times 2 \Rightarrow 6c_1 + 4c_2 = 2$.

(2) $\Rightarrow 9c_1 + 4c_2 = 5$

$-3c_1 = -3$.

$\boxed{c_1 = 1}$

$3 + 2c_2 = 1$.

$2c_2 = 1 - 3$.

$t(n) = 1(3)^n + (-1)(2)^n$.

$\therefore \boxed{t(n) = 3^n - 2^n}$.

(5) $a_n = 6a_{n-1} - 5a_{n-2}$ with $a_1 = 1$ and $a_2 = 41$.

The characteristic eqn

$x^n - 6x^{n-1} + 5x^{n-2} = 0$.

$x^2 - 6x + 5 = 0$.

$(x-1)(x-5) = 0$.

5
1 | 5

General soln.

$\boxed{t(n) = c_1(1)^n + c_2(5)^n}$

$t(1) = c_1 + 5c_2 = 1$ ——— (1)

$t(2) = 2c_1 + 25c_2 = 41$ ——— (2)

Solving (1) & (2)

$c_1 + 5c_2 = 1$

$c_1 + 25c_2 = 41$

$-20c_2 = -40$.

$\boxed{c_2 = 2}$

$c_1 + 10 = 1$

$\boxed{c_1 = -9}$

Particular soln.

$\boxed{t(n) = -9 + 2(5)^n}$.

(6) $a_n = a_{n-1} + 2a_{n-2}$ with $a_1 = -3$ and $a_2 = 9$.

The characteristic eqn.

$$x^n - x^{n-1} - 2x^{n-2} = 0.$$

$$x^2 - x - 2 = 0.$$

$$(x-2)(x+1) = 0$$

$$r_1 = 2 \quad r_2 = -1.$$

	2
+1	-2

General soln.

$$\boxed{t(n) = c_1(2)^n - c_2(1)^n}$$

$$t(1) = 2c_1 - c_2 = -3 \quad \text{---} (1)$$

$$t(2) = 4c_1 - c_2 = 9. \quad \text{---} (2)$$

(1) $\Rightarrow$ $2c_1 - c_2 = -3$.

(2) $\Rightarrow$ $4c_1 - c_2 = 9$

$- \quad + \quad -$

$$-2c_1 = -12.$$

$$\boxed{c_1 = 6}$$

$$12 - c_2 = -3.$$

$$-c_2 = -3 - 12.$$

$$-c_2 = -15$$

$$\boxed{c_2 = 15}$$

Particular soln.

$$\boxed{t(n) = 6(2)^n - 15}$$

(4) $a_n = 6a_{n-1} - 9a_{n-2}$ with $a_0 = -2$ & $a_1 = 6$. (3b)

Characteristic eqn.

$$x^n - 6x^{n-1} + 9x^{n-2} = 0.$$

$$x^2 - 6x + 9 = 0$$

$$(x-3)(x-3) = 0.$$

9
-3 | -3

$$r_1 = 3 \quad r_2 = 3.$$

General Soln

$$t(n) = c_1(3)^n + c_2 n(3)^n.$$

$$t(0) = c_1 + 0 = -2.$$

$$\boxed{c_1 = -2.}$$

$$t(1) = 3c_1 + 3c_2 = 6.$$

$$-6 + 3c_2 = 6.$$

$$3c_2 = 6 + 6.$$

$$3c_2 = 12.$$

$$\boxed{c_2 = 4}$$

$$t(n) = (-2)(3)^n + 4n(3)^n.$$

$$\boxed{t(n) = (3)^n [-2 + 4n]}.$$

Mathematical Induction

1) Prove using mathematical induction that for all $n \geq 1$

$1+4+7+\ldots+(3n-2) = \frac{n(3n-1)}{2}$

$S(n) = 1+4+7+\ldots+(3n-2) = \frac{n(3n-1)}{2}$

$S(1) = \frac{1(3-1)}{2} = 1$

$S(2) = \frac{2(6-1)}{2} = 5$

$S(3) = \frac{3(9-1)}{2} = \frac{3\times 8}{2} = 12$.

$n = k$.

$S(k) = 1+4+7+\ldots+(3k-2) = \frac{k(3k-1)}{2}$.

For $k+1$

$S(k+1) = 1+4+7+\ldots+(3(k+1)-2) = \frac{k(3k-1)}{2} + (3(k+1)-2)$

$= \frac{3k^2-k}{2} + 3k+3-2$

$= \frac{3k^2-k+6k+6-4}{2}$.

$= \frac{3k^2+5k+2}{2}$.

$= 3k^2+3k+2k+2/2$.

$= 3k(k+1)+2(k+1)/2$.

$= (3k+2)(k+1)/2$.

i.e. $S(k+1)$ is valid.

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

Thm If $t_1(n) \in O(g_1(n))$ & $t_2(n) \in O(g_2(n))$ then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

Proof Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 & some nonnegative integer n_1 such that

$$t_1(n) \leq c_1 g_1(n) \quad \forall n \geq n_1$$

Also since $t_2(n) \in O(g_2(n))$

$$t_2(n) \leq c_2 g_2(n) \quad \forall n \geq n_2.$$

[a_1, b_1, a_2, b_2 be four arbitrary real nr if $a_1 \leq b_1$ & $a_2 \leq b_2$ then $a_1 + a_2 \leq 2\max\{b_1, b_2\}$]

$$t_1(n) + t_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$$

$$\leq c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)]$$

$$\leq c_3 \, 2\max\{g_1(n), g_2(n)\}.$$

[$c_3 = \max\{c_1, c_2\}$, $n \geq \max\{n_1, n_2\}$]

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