DESIGN AND ANALYSIS OF ALGORITHM.

Algorithm - Set of unambiguous instructions used for solving a problem to provide correct and expected outputs for all valid & legal input data.

Agent

valid input → [Algorithm] → o/p. Agent ⇒ human / computer.

Characteristics of Algorithm.

(i) Input - Algorithm takes 0 or more inputs.

(ii) Output - produce atleast 1 or more outputs.

(iii) Definiteness - instructions must be clear & unambiguous without any confusion

(iv) Uniqueness - well-defined and ordered procedure that consists of set of instructions in specific order

(v) Correctness - Algorithm should produce correct output for valid input.

(vi) Finiteness - Algorithm should have finite no. of steps and should terminate after executing finite set of instns.

(vii) Simplicity - Ease of implementation

(viii) Generality - Algorithm should be generic, independent of any programming language or OS.

Problem Solving:

* Evaluating an algorithm design uses 2 criteria namely algorithm correctness and efficiency. Designing an algorithm that is both correct and efficient is a challenging task.

* If the problem statement is clear & understandable, then the algorithm design can be done efficiently.

Algorithm Design:

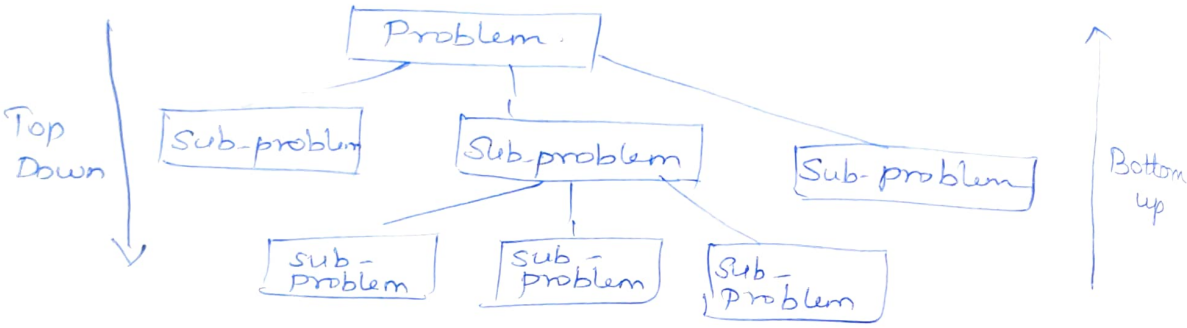
* Algorithm design can be done in 2 ways.

(i) Top down approach (Stepwise refinement)

(ii) Bottom up approach (Synthesis or compositional)

Top-Down approach:

* Starts with general outline of the problem. The problem is divided into set of subproblems or sub tasks which are further subdivided. Decomposition continues until subtask (solved as function or procedure) can be solved directly.



Advantages:

- Overall solution in terms of solutions to subproblems
- Support structured & object oriented programming.
- encourage abstraction

Disadvantage:

- more redundant work
- Subproblems interrelated and overlapping testing is difficult.

Bottom up approach:

* Results of smaller subproblem obtained and used to get the solution of larger problem at top.

eg. 3!

Top down:

$3! = 3 \times 2!$
$= 3 \times 2 \times 1!$
$= 3 \times 2 \times 1 \times 0!$
$= 3 \times 2 \times 1 \times 1$
$= 6$

Bottom up:

$3! \Rightarrow 0! = 1$
$1! = 1 \times 0! = 1 \times 1 = 1$
$2! = 2 \times 1! = 2 \times 1 = 2$
$3! = 3 \times 2! = 3 \times 2 = 6$

Advantage:

- Smaller problems solutions reused for solving bigger problem.
- Testing of smaller subproblem is easy.

Algorithm Analysis

* Algorithm is analysed using complexity analysis - finding optimal algorithms that use fewer computer resources.

* Complexity theory is a field of algorithms that ~~use few~~ deals with the analysis of a solution in terms of computational resources and their optimization.

* Complexity of an algorithm can be related to its input size. eg. An algorithm for sorting an array of 10 nos. is easy, but the same becomes difficult for 1 million inputs.

* Complexity analysis is done for 2 purposes.

1. To decide the efficiency of given algorithm.
2. To compare algorithms for deciding the effective solutions for a given problem.

* Two types of measures available for comparing algorithms

- Subjective measures - ease of implementation
  (non-measurable) L readability of algorithm.
- Objective measures - measurable & quantifiable.

  Time Complexity Space Complexity

Time Complexity:

* The time complexity of an alg. is estimated by counting the no of operations. It considers every step & every operation.
* Classified into 2 types: unit cost model & logarithmic cost model.
* Unit cost model: cost for performing an operation is assumed to be 1. The cost is not dependent on operand.
* Logarithm cost model: cost is dependent on the operand. If larger integer is used, then the cost is determined as the logarithm of the size of operand multified by a factor c (eg. c.log n).
* Time Complexity analysis of an alg. can be performed in 2 ways: – 1. Mathematical Analysis 2. Emprical analysis.

1. Mathematical analysis:
   - a priori analysis or theoretical analysis.
   - performed before the algorithm is translated into a program.
   - estimates time complexity in terms of step count or operation count.
   - independent of any particular machine, programming language, OS, compiler and so on.

2. Emprical analysis:
   - a posteriori analysis
   - analysis is performed after the implementation of an alg. in form of program.
   - a program is executed with real-time data sets and running time T(n) is determined by clocking the speed.
   - accurately find the values of machine-dependent algorithm parameters.

Analysis of Iterative Algorithms:

* Steps involved in mathematical analysis :

1. Measure the input size (ie) length of i/p data
2. Measure the running time (step count / operation count)
3. In case of operation count, find worst-, best- and average case efficiencies of algorithm.
4. Identify the rate of growth. This step determines the performance of an algorithm when the input size is scaled higher. Scaling means increasing input size and accessing the behavior of the algorithm.

1) Measuring input size :

* Alg analysis is dependent on input size.
* Input size (denoted as n or N) is defined formally as the no. of binary bits used to represent input data.

* Some inputs to the alg :

→ Integers - Common practice is to use the no. of bits required to store an input. eg. Assume N = 10. Then input size is $\log_2(N) = 4$ (binary repr. of 10 is 1010).

→ Arrays - The order of array used as input size.

eg. $I = (x_1, x_2 \ldots x_n)$, then the size is $\log_2 n$.

2) Measuring running time :

* Running time of alg. can be carried out - Step count or operation count.

Step count :

* Counts the no. of instructions or steps that are used by a given alg. to perform given task.
* Need to find steps per execution and frequency (no. of times) of statements.
* No. of steps per execution is 0 for non-executable stmts and 1 for executable stmts.

Ways of measuring step counts per execution.

1. Declarative stmts with no initializations have count 0
   With initializations, step count is 1.
2. Comments, brackets, begin/end/end if/end while/end for all having step count of 0.
3. Expressions have step count of 1.
4. Assignment stmt, fn invocation, return stmts & other stmts (break/continue/go) all have step count of 1.

* Total count can be obtained by multiplying frequency and steps per execution.

eg 1: Alg: Simple (a, b, c)

```
Begin
  A = B + 1
  C = A + 2
  D = A + B
End.
```

Find the step count of the alg.

Sl. No	Prg.	Steps/exec.	Freq	Total
1	Alg Simple (A, B, C)	0	–	–
2	Begin	0	–	–
3	A = B + 1	1	1	1
4	C = A + 2	1	1	1
5	D = A + B	1	1	1
6	End	0	–	=
				3

$\therefore T(n) = 3.$

eg 2

```
Alg Sum ()
Begin
  Sum = 0.0
  for i = 1 to n do
      Sum = Sum + i
  End for
  Return Sum
End.
```

Sl.	Pgm.	Step / exe	Freq	Total Count
1	Alg Sum ()	0	0	0
2	Begin	0	0	0
3	Sum = 0.0	1	1	1
4.	For i = 1 to n do	1	n+1	n+1
5	Sum = Sum + i	1	n	n
6.	End for	0	0	0
7	Return Sum	1	1	1
8	End	0	0	0
				2n+3.

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

Adv: Easy

Disadv: Not dependent on operands size.

eg. $a = a + 10$ & $a = a + 10,000,000,000$ have same step count.

Operation Count:

* The idea is to count the no. of operations. Operations can be divided into 2 categories: elementary (basic) and non-elementary (non-basic) operations.

* Elementary operations are basic one implemented with less effort.

eg. Assignment, Comparison, Arithmetic, Logical.

* Non-elementary operations are not atomic & involve many elementary operations.

eg. sorting, finding max in an array.

* Rules for finding Operation Count

(i) Sequence

Begin
S_1
S_2
End

* Let Stmt S_1 requires m operations & S_2 requires n operations
* Then, it requires m+n operations
* Addition principle

(ii) Selection

If (cond c) then Stmt P
else Stmt Q

* P requires m operations & Q requires n operations

Then if-then-else requires max (m, n) operations.

(iii) Repetition:

If loop executes a task n times and task involves m operations, then operation count is n×m.

Multiplication principle.

eg:

```
Alg swap (a, b)
Begin
  temp = a
  a = b
  b = a
End.
```

Step no	Pgm	Operations	Operation Cost	Freq	Total
1	Alg swap (a,b)	–	–	–	–
2	Begin	–	–	–	–
3	temp = a	Assignment	c_1	1	c_1
4	a = b	"	c_2	1	c_2
5	b = a	"	c_3	1	c_3
6	end	–	–	–	–

$$T(n) = c_1 + c_2 + c_3$$

This is constant time alg. Exact cost estimation and constants are machine dependent.

eg 2 for k = 1 to n do
    A = A × k
End for

Step	Alg.	Operation	Operation cost	Freq.	Total cost
1	Alg Sample ()	–	–	–	–
2	Begin	–	–	–	–
3	for k = 1 to n do	Assigning, testing & increment	c_1	$n+1$	$(n+1)c_1$
4	A = A * k	Multiplication	c_2	n	nc_2
5	End for	–	–	–	–
6	End	–	–	–	–

$$t(n) = (n+1) \times c_1 + n \times c_2$$
$$= (c_1 + c_2)n + c_1$$
$$= cn.$$

where c is machine dependent cost & n is input size.

Analysis as Dominant operations.

Counting all operations is difficult. Simple if the count of only dominant operation is taken into account.

eg. for i = 1 to n do
    for j = 1 to n do
        A = B × C
    end for
end for.

* Multiplication is the dominant operation & there are 2 loops for execution. Hence multiplication is done n × n times

$\therefore T(n) = n^2.$

eg.

```
void transpose(int a[][], int n)
{
    for (int i=0; i<n; i++)
1       for (int j=i+1; j<n; j++)
2           swap(a[i][j], a[j][i]);
3
}
```

Operation cost	Frequency	Total cost
c_1	$n+1$	$c_1(n+1)$
c_2	$n(n+1)/2$	$c_2 \cdot n(n+1)/2$
c_3	$n(n-1)/2$	$c_3 \cdot n(n-1)/2$

Outer for loop:

$i = 0$ to $n-1 \Rightarrow n$ times

false condition $\Rightarrow 1$ times

$\therefore n+1$ times.

Inner for loop $\Rightarrow$ consider outer loop also

if $i = 0$ $\quad j = 1$ to $n-1 \Rightarrow n-1$ times $\Rightarrow n-1+1 = n$ times

if $i = 1$ $\quad j = 2$ to $n-1 \Rightarrow n-2$ times $\Rightarrow n-2+1 = (n-1)$ times

$\vdots$

if $i = n-1$ $\quad j = n$ to $(n-1+1) \Rightarrow n$ to $n = 1$ time

$1 + 2 + \dots + (n-1) + n = n(n+1)/2$

Swap:

$i = 0$ $\quad j = 1$ to $n-1 \Rightarrow (n-1)$ times

$i = 1$ $\quad j = 2$ to $n-2 \Rightarrow (n-2)$ times

$\vdots$

$i = n-2$

$j = (n-2)+1 \Rightarrow n-1$ to $n-1$ times $= 1$ time

$i = n-1$

$j = (n-1+1)$ n to $n-1 = 0$ times

$0+1+2 \cdots + (n-2) + (n-1) = \frac{N(N+1)}{2}$ where $N = (n-1)$

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

$$T(n) = c_1(n+1) + c_2 \cdot \frac{n(n+1)}{2} + c_3 \cdot \frac{n(n-1)}{2}$$

$$= c_1 n + c_1 + c_2 \cdot \frac{n^2}{2} + c_2 \cdot \frac{n}{2} + c_3 \cdot \frac{n^2}{2} - c_3 \cdot \frac{n}{2}$$

$$= \left(\frac{c_2}{2} + \frac{c_3}{2}\right) n^2 + \left(c_2 \cancel{+} \frac{c_2}{2} - \frac{c_3}{2}\right) \cdot n + c_1$$

$O(n^2)$.

Analysis of Dominant Operations.

- → Basic operation ⇒ loops. → stmt within innermost loop.
  - most important operation that takes the maximum operation time (or) execution time.
- → Find out the execution time of basic operation ⇒ b
- → Frequency of basic operation ⇒ $c(n)$.
- → Total cost.

eg. 1:

```
int max (int a[], int n)
{
  max = a[0]
  for (i=1; i<n; i++)
  {
    if (a[i] > max)
      max = a[i]
  }
  return max;
}
```

In above ex. the comparison stmt is the basic operation

Execution time basic operation.

Constant amt of time = 1 unit (or) k units.

Freq.

$i = 1$ to $n-1$ ⇒ ~~$n-1+1 = n$~~

$= (n-1)$ times

$T(n) = k(n-1) = kn - k$ $\quad T(n) = O(n)$.

eg2 :

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

Basic operation
=) Addition & Assignment

Exec. time
$b = 1$ or k units

Freq. count
$C(n) = n^2$.

$$T(n) = b \times C(n)$$
$$= k \cdot n^2$$
$$= kn^2.$$
$$T(n) = O(n^2).$$

3. Best-, Worst, Average-Case Analysis :

Worst-case :
* Alg. takes maximum time for executions. The alg. needs to perform the max. no. of steps or operations to process the input for accomplishing given task.
* It gives an upper bound on the alg

Best-case :
* Alg takes minimum computation time.
* It represents lower bound on the alg.

Average Case :
* It is neither the best nor the worst case. It indicates the average performance of an alg.

eg. Linear Search problem.

```
for (i=0; i<n; i++)
  if (a[i] = key)
    return i;
  else
    return -1;
```

Worst case =) $T(n) = n$.
(Item not present)

Best case =) $T(n) = 1$.
(element is in pos 1)

Average case
(Element anywhere in the list).
$p = 1/n$.

$$\text{Comparisons} = 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + 3 \frac{1}{n} + \dots + n \cdot \frac{1}{n}$$
$$= \frac{1}{n}(1+2+\dots+n) = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2} \approx \frac{n}{2}$$

4. Rate of Growth:

+ How do 2 alg. differ if their run-times differ by only a few seconds or minutes?

+ If running time of one alg. is significantly larger than that of another, it is a matter of concern.

+ It is to decide which alg. is better based on how the alg. will behave when the input size N is scaled higher.

Thus, rate of growth of an alg is the rate at which running time increases as a fn. of input n.

Eg. Consider 3 alg. A, B & C. Their respective run-time complexities $T_A = 7n$, $T_B = 7\log_{10} n$, $T_C = n^2$. Compare the rates of growth when the input is scaled from $n = 100$ to $n = 10,000$

$$\frac{T_A(10000)}{T_A(100)} = \frac{7 \times 10,000}{7 \times 100} = 100.$$

$$\frac{T_B(10000)}{T_B(100)} = \frac{7 \cdot \log_{10}(10000)}{7 \log_{10}(100)} = \frac{7 \cdot \log_{10} 10^4}{7 \cdot \log_{10} 10^2}$$

$$= \frac{7 \times 4 \times \log_{10} 10}{7 \times 2 \times \log_{10} 10} = 2.$$

$$\frac{T_C(10000)}{T_C(100)} = \frac{10000^2}{100 \times 100} = 10,000.$$

$\therefore$ Second alg, T_B is better than T_A & T_C.

Comparison framework:

Rates of growth of functions.

Function name.	Name used for alg. analysis.
C	Constant alg.
Log N	Logarithmic fn
N^2	Quadratic
N^3	Cubic
N^k	Polynomial
a^n	Exponential

Eg1: Consider 4 algorithms with the following time complexities

$T_1(n) = n$, $T_2(n) = \log_2 n$, $T_3(n) = n^2$, $T_4(n) = 2^n$

Analysis of rates of growth.

Input size	n	$\log_2 n$	n^2	2^n
1	1	0	1	2
10	10	3.32	100	1024
100	100	6.64	10^4	2^{100}
1000	1000	9.96	10^6	2^{1000}.

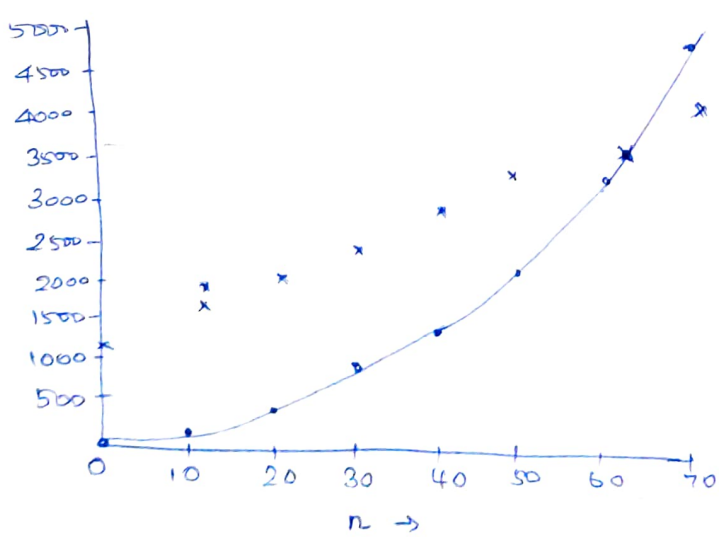
- Logarithmic functions provide optimal solutions.
- Quadratic functions gradually increase the running time
- Exponential function grows at a tremendous rate making computing more and more difficult as n grows larger.

Eg 2: Let A & B be 2 algorithms with the complexity function $t_A = n^2$ and $t_B = 40n + 1200$. Which alg. is better?

Soln: Find an instance of n where $T_A = T_B$. This instance can be obtained as follows.

$$n^2 = 40n + 1200 \iff n^2 - 40n - 1200 = 0$$

The roots of equation are $+60$ and -20. Therefore, when $n = 60$, both algorithms are equal.



A —— n^2

B * $40n+1200$

For $n > 60$, alg A can be seen to grow at a faster rate than alg B. Both alg A and B are suitable when $n = 60$. and alg A is more favourable than alg B when the input ranges from 1 to 59. For values of $n > 60$, alg B is better than Alg. A.

ASYMPTOTIC ANALYSIS

Asymptotic Notation.

* Asymptotic analysis is a kind of alg. analysis with larger values of input data.

Asymptotic Notations:

* We use notations to represent the time complexities which are given using functions.
* The notations are used for representing the simple form of a function
* Comparison of functions.

$$(1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \dots < 2^n < 3^n < \dots n^n)$$

↳ Time complexity of any alg (fn).

Notations:

1. O - big oh - upper bound of a function
2. Ω - big omega - lower bound of a function
3. θ - theta - Average bound of a function.

(i) Big Oh (O):

The function $f(n) = O(g(n))$ if and only if there exists a positive constant c and no such that $f(n) \leq c * g(n) \ \forall n \geq n_0$.

eg. $f(n) = 2n + 3$

$2n + 3 \leq 10n \Rightarrow$ This is true $\forall n \geq 1$

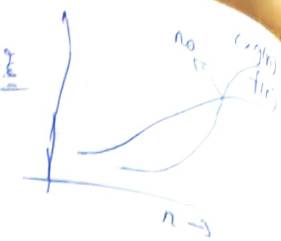
$2n + 3 \leq 7n \Rightarrow$ This is also true.

$f(n) \leq c \cdot g(n)$ where $c = 7$ & $g(n) = n$.

$f(n) = O(n)$

$2n+3 \le 5n^2 \ \forall n \ge 1$

$f(n) \le \cancel{5n^2}$ $c \cdot g(n)$ where $c = 5$ & $g(n) = n^2$



$\therefore f(n) = O(n^2)$

~~f(n) belongs~~

$f(n) = 2n+3$ belongs to class. So big-O can be any class to the right of class n.

$1 < \log n < \sqrt{n} < n < n.\log n < n^2 < n^3 < \dots < 2^n < 3^n < \dots n^n$

lower bound — $f(n)$ — Average bound — upper bound.

$f(n) = O(\log n) \Rightarrow$ This is not true.

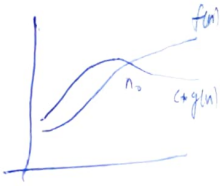
Big O should be the closest function. So, for $f(n)$ belonging to n class, the closest fn. is n itself.

2. Big-Omega (Ω) notation:

The function $f(n) = \Omega g(n)$ if and only if there exists constant +ve value for c and n_0 such that $f(n) \ge c * g(n) \ \forall n \ge n_0$.

eg. $f(n) = 2n+3$

$2n+3 \ge 1 \times n$ where $c = 1$ & $g(n) = n$ $\forall n \ge 1$



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

$2n+3 \ge 1.\log(n) \ \forall n \ge 1$.

↳ This is also true.

Big omega should be represented by closest function

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

3. Theta (Θ) notation:

The function $f(n) = \Theta(g(n))$ if and only if there exists +ve constants c_1, c_2 and n_0 such that

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

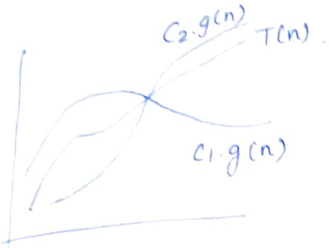
eg $f(n) = 2n+3$

$1 \cdot n \le 2n+3 \le 5 \cdot n$ where $c_1 = 1$

$c_1 \cdot g(n)$ $c_2 \cdot g(n)$ $c_2 = 5$

$g(n) = n$

$\therefore f(n) = \Theta(n)$.



4. Little-oh notation (o)

↳ used very rarely

↳ for small value of c like $c = 0.001$

$f(n) = o(g(n))$ iff +ve constants c & no. such that $f(n) < c \times g(n)$

5. Little-omega notation (ω)

$t(n) = \omega(g(n))$ iff there exists +ve constants c & no such that $f(n) > c * g(n)$.

Problems

1. Let $t(n) = 3n^3$ for an alg. Prove that $t(n)$ of the algorithm is in $O(n^3)$.

Soln: Definition of big-oh $t(n) \le c \cdot g(n)$

To prove $3n^3 \le c \cdot n^3$ [since $g(n) = n^3$].

↳ This holds when $c \ge 3$.

(ie) $c = 1$. & $n = 1$

$3 \cdot 1^3 \le 1 \cdot 1^3 \Rightarrow 3 \le 1$.

$3 \cdot 1^3 \le 3 \cdot 1^3 \Rightarrow 3 \le 3$.

Therefore $t(n) = g(n)$

$\therefore t(n) = O(n^3)$.

2. Let $t(n) = n^4 + 3n^3 + 2n + 1$. Let $g(n) = n^3 + 4$. Prove that $t(n)$ ~~of~~ alg. is $\Omega(n^3)$.

Defn. of big ~~the~~ omega. $t(n) \ge c * g(n)$.

To show, $n^4 + 3n^3 + 2n + 1 \ge c \cdot (n^3 + 4)$.

holds for +ve no. c.

$c = 1, n = 1$.

$n^4 + 3n^3 + 2n + 1 \geq 1.(n^3 + 4)$

$1 + 3 + 2 + 1 \geq 1.(1 + 4)$.

$7 \geq 5$.

Therefore, $t(n) = \Omega(n^3)$.

MATHEMATICAL ANALYSIS OF RECURSIVE ALGORITHMS.

Recurrence Equations:

* Analysing recursive algorithm is a difficult task. As recursive functions call themselves conditionally, it is difficult to obtain the count of basic operations in recursive programs. Hence, analysis of recursive algorithms requires recurrence equations

* To analyse recursive algorithm, 2 steps needed:

1. Formulating a recurrence equation
2. Solving the recurrence equation to understand the behavior of the program.

* Recurrence equation defines a sequence using the elements of that sequence.

* A sequence is a set of ordered elements or numbers. eg. 1, 3, 5, 7, 9 .... sequence of odd numbers. 0, 2, 4, 6, 8 .. sequence of even numbers.

* Therefore, a sequence is a finite or infinite list of numbers

* Time complexity function $T(n)$ can be represented as a sequence $T(0), T(1), T(2) \ldots T(n)$. Also denoted as $t_0, t_1, t_2 \ldots t_n$. $T(n)$ or t_n denotes the time complexity of an algorithm for an input n. This is called a sequence.

* A recurrence equation is a compact notation used to illustrate a sequence. A recurrence eqn. of a sequence $t_0, t_1, t_2, t_3 \cdots t_n$ is a compact equation that expresses a term of a sequence in relation to one or more preceding terms.

eg. The recursive relation of a factorial is given as,

$$n! = n \times (n-1)!$$

with a base condition $0! = 1$.

* The recurrence equation of a factorial is

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

+1 is given to represent one multiplication operation required.

* Recurrence Relation is of 2 types:

1.- Linear recurrences

2.- Non-linear recurrences.

1. Linear Recurrences:

* Formally, the terms of a linear equation are constants, a product of constants or the first power of a variable. Hence, a linear recurrence equation for a sequence $\{t_0, t_1, \cdots t_n\}$ expresses the final term t_n as a linear combination of its previous terms in a polynomial form.

* eg. The recurrence equation for a fibonacci series

$t_n = t_{n-1} + t_{n-2}$ => Every term in a Fibonacci sequence can be expressed as a sum of its previous 2 terms.

* In general, recurrence eqn. is of the following form:

$$a_0 t_n + a_1 t_{n-1} + \cdots a_k t_{n-k} = f(n).$$

where a_i and k are constants, n-order of recurrence eqn.

* Linear recurrences can be classified based on

→ Order

→ homogeneity.

Order of Recurrence Equations:

* The number of preceding terms used for computing the present term of a sequence is called the order of a recurrence equation. (ie) order is the difference between the highest and lowest subscripts of dependent variable in a recurrence equation.

eg. $t_n - t_{n-1} - 7t_{n-2} = 0$, order is $n-(n-2) = 2$.

* In alg analysis, mostly first and second-order recurrence equations are encountered.

* General form of first-order recurrence is

$$a_0 t_n + a_1 t_{n-1} = f(n)$$

If t_n is computed using only one previous term, then the recurrence eqn. is called first-order linear recurrence eqn. eg. Factorial, $t_n = t_{n-1} + 1$.

* General form of second order recurrence is

$$a_0 t_n + a_1 t_{n-1} + a_2 t_{n-2} = f(n)$$

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

* Higher order linear recurrences of order k is

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

Homogeneous Vs Non-homogeneous Equations

* In recurrence eqn. $a_0 t_n + a_1 t_{n-1} + \dots + a_k t_{n-k} = f(n)$, if $f(n) = 0$, it is called a homogeneous equation and if $f(n) \neq 0$, it is called non-homogeneous equation.

* To determine whether a recurrence equation is homogeneous (or) non-homogeneous, the homogeneity test is conducted. In the homogeneity test, all the order of t_n are substituted with zero and it is checked if the left-hand (LHS) is the same as right side (RHS).

eg, $t_n = t_{n-1} + t_{n-2}$. Apply $t_n, t_{n-1}, t_{n-2} = 0$.

$0 = 0 + 0$.

Therefore, $t_n = t_{n-1} + t_{n-2}$ is a homogeneous equation.

eg, $t_n = t_{n-1} + (n-3)$.

$0 = 0 + (n-3)$

$0 \neq (n-3)$.

Therefore, $t_n = t_{n-1} + (n-3)$ is non-homogeneous equation.

2. Non-linear recurrences:

* The non-linear recurrence equation of a sequence $\{t_0, t_1, \ldots t_n\}$ expresses t_n as a non-linear combination of its previous terms.

* A unique form of non-linear recurrence equations, called divide and conquer recurrences are often encountered. The divide and conquere recurrences are of the following form:

$$T(n) = a.T\left(\frac{n}{b}\right) + f(n).$$

here a is the number of subproblems, n is the size of the problem, n/b is size of subproblem, and $f(n)$ is the cost of work done for non-recursive calls, which accounts for division of problem into subproblems and the combination of the results of subproblem.

eg, The recurrence equation for merge sort is

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

Implies that no. of subproblems is 2 at every level, problem size reduced by factor of 2, n amount of work has to be performed to combine results.

FORMULATION OF RECURRENCE EQUATIONS.

* Formulation of a recurrence equation is more difficult than solving it. Given a sequence, we need to identify the pattern of the sequence, which is a challenging task.

eg) 1, 3, 5, 7, 9 . . . ⇒ odd no sequence

2, 4, 6, 8, 10 . . . ⇒ even no sequence

1, 7, 13, 19, 25 . . . ⇒ difference between 2 consecutive elements is 6. This is an arithmetic series.

3, 9, 27, 81, . . . ↳ represents a geometric series where the ratio of successive elements is 3.

eg) For the sequence 1, 4, 7, 10, . . . the recurrence equation is as follows.

$$T(n) = T(n-1) + 3.$$
$$T(0) = 1.$$

$T(0) = 1$ is base condition,

$T(1) = T(1-1) + 3 = T(0) + 3 = 1 + 3 = 4.$

$T(2) = T(2-1) + 3 = T(1) + 3 = 4 + 3 = 7$

and so on.

* The equation can be denoted as follows:

$$t_n = t_{n-1} + 3$$
$$t_0 = 1.$$

Example 1. What is the recurrence equation of the sequence 1000, 2000, 4000, 8000, . . .?

Solution:

$t_0 = 1000$

$t_1 = 2000 = 2 \times 1000 = 2 \cdot t_0$

$t_2 = 4000 = 2 \times 2000 = 2 \cdot t_1$

$\Rightarrow 4 \times 1000 = 2^2 \times 1000 = 2^2 \cdot t_0.$

$t_n = 2 \cdot t_{n-1}$ (or) $2^n \cdot t_0$.

Both forms the solution but the second one forms the original solution.

What would be the 100th element in the sequence?

$t_{100} = 2^{100} \times 1000$

Example 2: Find the recurrence equation and the initial condition of the sequence: $7, \frac{21}{4}, \frac{63}{16}, \frac{189}{64}, \ldots$

Solution: Let the initial condition be $t_0 = 7$.

Observe the pattern. Calculate the ratios of successive elements

$$\frac{t_1}{t_0} = \frac{21/4}{7} = \frac{21}{28} = \frac{3}{4}$$

$$\frac{t_2}{t_1} = \frac{63/16}{21/4} = \frac{63}{16} \times \frac{4}{21} = \frac{3}{4}$$

$$\frac{t_3}{t_2} = \frac{189/64}{63/16} = \frac{189}{64} \times \frac{16}{63} = 3/4.$$

Therefore, $\frac{t_n}{t_{n-1}} = \frac{3}{4} \Rightarrow t_n = t_{n-1} \times \frac{3}{4}$.

Example 3: (Staircase problem). Assume that a person can climb the stairs by taking one or atmost 2 steps at a time. Construct a recurrence relation for reaching the nth step.

Solution: Last step = nth step. A person could have climbed from t_{n-1} or t_{n-2}.

Step no.	No. of possibilities.
1	1
2	2 { 1+1 (step step) or 2 steps at a time
3	3 { 1+1+1, 2+1, 1+2
4	5 { 1+1+1+1, 1+2+1, 1+1+2, 2+1+1, 2+2

So, the sequence is 1, 2, 3, 5, 8 ..... The recurrence equation for the following sequence is:

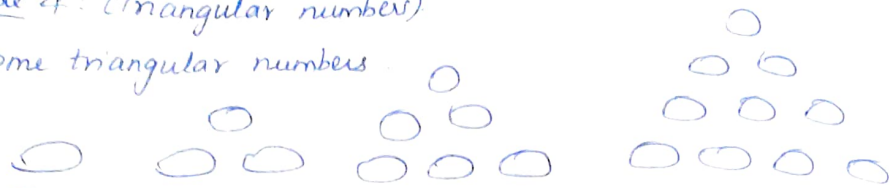
$$T(n) = T(n-1) + T(n-2)$$

(or)

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

Example 4: (Triangular numbers)

Some triangular numbers



$T = 1$ $T = 3$ $T = 6$ $T = 10$.

What is the recurrence equation for describing dots?

Solution:

The sequence of dots that represent the triangular numbers - 1, 3, 6, 10, 15, 21. . . .

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

$T(1) = 1$

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

$T(0) = 1$

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

$T(1) = 1$

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

$T(3) = T(2) + 3 = 3 + 3 = 6$.

$T(4) = T(3) + 4 = 6 + 4 = 10$.

TECHNIQUES FOR SOLVING RECURRENCE EQUATIONS:

* A solution for a recurrence equation is a non-recursive formula (also called closed formula) that satisfies the recurrence equation for all the range of values. Such a function that satisfies a given recurrence equation is called a general solution.

* A solution of a given recurrence equation is a formula that satisfies the given recurrence equations with respect to its initial conditions and this is called particular solution.

* Methods for solving recurrence equations:
  - Substitution method
  - Recurrence tree method.

Substitution Method:

* It is also called the iteration method.
* The solutions can be obtained by repeated substitution of RHS of recurrence equation till a pattern is obtained.

* If ~~pattern~~ solution is obtained by repeated substitution from base condition onwards, it is called forward substitution.

* If substitution starts from the last term and proceeds to the initial term, it is backward substitution or backtracking.

* Both the substitution involves 2 steps,

→ plug - substitute repeatedly.

→ chug - simplify the expressions.

Backward Substitution:

* It starts from the last term (t_n) of a sequence and moves towards the base (t_0) by repeatedly substituting RHS of a recurrence equation.

* The process is continued till the closed form is obtained, which can be confirmed by the observation of the common pattern that emerges from the process of repeated substitution.

Example 1: Solve the following recurrence equation:

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

$$t_1 = 4.$$

Solution:

* Substitute values of t_{n-1}

$= (t_{n-2} + 3) + 3$ (plug)

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

$= t_{n-2} + 2 \times 3$ (Chug).

* Repeat substitution for t_{n-2}

$= (t_{n-3} + 3) + (2 \times 3)$ (plug).

$= t_{n-3} + 9.$

$= t_{n-3} + (3 \times 3)$ (chug).

* Repeat substitution for t_{n-3}

$= (t_{n-4} + 3) + (3 \times 3)$ (plug)

$= t_{n-4} + 12$

$= t_{n-4} + (4 \times 3)$ (chug).

* By repeating the process, the eqn. in ith iteration:

$$t_n = t_{n-i} + (i \times 3)$$

when $i = n-1$,

$$t_n = t_{n-(n-1)} + (n-1) \times 3$$
$$= t_1 + 3(n-1)$$
$$= 4 + 3(n-1)$$
$$= 4 + 3n - 3$$
$$t_n = 3n + 1$$

Example 2: Solve the following recurrence equation.

$$t_n = n \cdot t_{n-1} \quad \text{for } n > 1$$
$$t_0 = 1.$$

Solution: Apply backward substitution

$$t_n = n \cdot t_{n-1}$$
$$= n[(n-1) \cdot t_{n-2}]$$
$$= n \cdot (n-1)[(n-2)\, t_{n-3}]$$
$$= n \cdot (n-1) \cdot (n-2)[(n-3) \cdot t_{n-4}]$$

$t_n = n \cdot t_{n-1}$

$t_{n-1} = (n-1) \cdot t_{(n-1)}$

$= (n-1) \cdot t_{n-2}$

$t_{n-2} = (n-2) \cdot t_{(n-2)}$

$= (n-2) \cdot t_{n-3}$

At ith step.

$$t_n = n(n-1)(n-2)(n-3) \ldots (n-i) \cdot t_{n-(i+1)}$$

when $i = n-1$

$$t_n = n(n-1)(n-2) \ldots (n-(n-1)) \cdot t_{n-(n-1+1)}$$
$$= n(n-1)(n-2) \ldots 1 \cdot t_0$$
$$= n(n-1)(n-2) \ldots 1$$
$$= n!$$

Forward Substitution:

* It is a method of solving the recurrence equation by repeated of its terms. The substitution starts from the base condition (t_0) and continues till the final term (t_n) is obtained.

eg. Solve the following recurrence equation using forward substitution method: $t_n = t_{n-1} + 3$ with initial condition $t_0 = 4$.

Solution:

$t_0 = 4$

$t_1 = t_0 + 3 = 4 + 3$ (plug).

$t_2 = t_1 + 3$

$= [4+3] + 3 = 4+3+3 = 4 + (2 \times 3)$ (chug)

$t_3 = t_2 + 3$ (plug)

$= [(4 + 2 \times 3)] + 3 = 4 + 3 + 3 + 3 = 4 + (3 \times 3)$ (chug)

$\vdots$

$= 4 + n \times 3$

$t_n = 4 + 3n$.

Mathematical Induction:

* It is a proof technique for proving the correctness of a program.

* Definition: To prove the statement "If p then q", mathematical induction is an argument that if p is true for base conditions, say 1, and for n and by induction (n+1) elements, then one can conclude that p is true for all n greater than or equal to base condition.

* It has 2 steps:

1. Base step - To prove $p(1)$ is true.
2. Inductive step - If $P(n)$ is true, then one has to prove that it works for $P(n+1)$ also (ie) $P(n) \rightarrow P(n+1)$ is true.

Eg 1: Use mathematical induction and P.T $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Basis Step: To prove for $n=1$.

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

$$\sum_{i=1} 1 = \frac{1(1+1)}{2} = 1.$$

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

$$\sum_{i=1}^{2} i + 3$$

$1+2+3$

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

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

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

Inductive step: To prove for $n+1$.

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

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

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

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

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

$$= \frac{n^2+n+2n+2}{2}$$

$$= \frac{n^2+3n+2}{2}$$

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

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

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

$$= \frac{n^2+2n+n+2}{2}$$

$$= \frac{n^2+3n+2}{2}$$

$LHS = RHS.$

Eg 2: Use M.I & P.T $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

Basis step: $n=1$

$$\sum_{i=1} 1^2 = \frac{1(1+1)(2+1)}{6} = \frac{2\times 3}{6} = 1.$$

Inductive step:

$$LHS = \sum_{i=1}^{n} i^2$$

$$= \sum_{i=1}^{n+1} i^2$$

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

$$= \frac{n(n+1)(2n+1)}{6} + (n+1)^2$$

$$= \frac{(n^2+n)(2n+1) + 6(n+1)^2}{6}$$

$$= \frac{(n^2+n)(2n+1) + 6(n^2+2n+1)}{6}$$

$$= \frac{2n^3+n^2+2n^2+n+6n^2+12n+6}{6}$$

$$= \frac{2n^3+9n^2+13n+6}{6}$$

$$RHS = \frac{n(n+1)(2n+1)}{6}$$

$$= \frac{(n+1)[(n+1)+1][2(n+1)+1]}{6}$$

$$= \frac{(n+1)(n+2)(2n+2+1)}{6}$$

$$= \frac{(n^2+2n+n+2)(2n+3)}{6}$$

$$= \frac{2n^3+3n^2+4n^2+6n+2n^2+3n+4n+6}{6}$$

$$= \frac{2n^3+9n^2+13n+6}{6}$$

LHS = RHS.

Guess and Verify:

- solution is obtained by mathematical induction.

Guess:

- Guess the solution by substituting different values of n to generate the sequence from recurrence equation.

Verify:

- verify the solution guessed in first phase. Verification is required because the solution is just guess. Verification or justification is done by mathematical induction.

eg. Solve the recurrence equation $t_n = t_{n-1} + 2$, $t_0 = 1$ using guess and verify method.

Guess: For making guess, use different values in recurrence eqn.

$t_0 = 1$

$t_1 = t_0 + 2 = 1 + 2 = 3$

$t_2 = t_1 + 2 = 3 + 2 = 5$

$t_3 = t_2 + 2 = 5 + 2 = 7$

The sequence obtained is $(1, 3, 5, 7 \ldots)$ indicates it is an odd-number sequence.

One can guess the solution to be $2n+1$, because it is a non-recursive formula.

Verify: Next step is to verify whether guess is right.

Use mathematical induction:

Basis step: verify for some initial terms.

Let $n = 0$ $\quad t_0 = 2(0) + 1 = 1$

$n = 1$ $\quad t_1 = 2(1) + 1 = 3$

$n = 2$ $\quad t_2 = 2(2) + 1 = 5$

So, for any n, eqn. $t_n = 2n+1$ holds.

Inductive step: To prove for $(n+1)$ as well.

Sub. $n = n+1$ in recurrence equation,

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

$$= t_n + 2$$

$$= 2n + 1 + 2$$

Rewrite

$$= 2n + 2 + 1$$

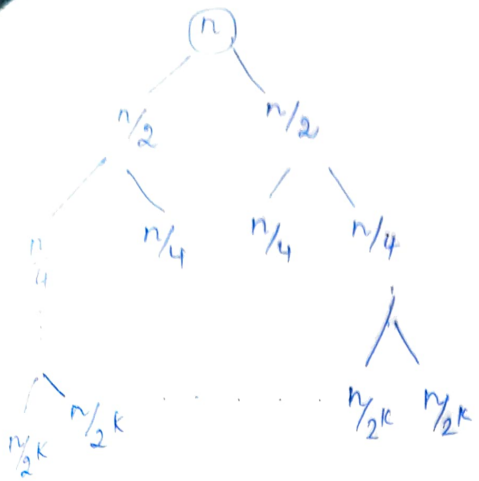
$$= 2(n+1) + 1.$$

RECURRENCE TREE:

* The recurrence tree method is another way of a recurrence equation by expanding its terms in a like manner.

Eg1: Solve recurrence equation $T(n) = 2T(n/2) + n$

The recurrence equation is interpreted as follows: divide the problem of instance n into 2 problems of size $n/2$ and combine the solutions of 2 subproblem using n amount of work to obtain final solution.



	cost
level 0	n
level 1	$n/2 + n/2 = n$
level 2	$n/4 + n/4 + n/4 + n/4 = n$
⋮	⋮
Level k	n

* The recurrence eqn can be expanded as a tree. At every level, the amount of work done is computed.
* In this case, the no. of problems grows as a power of 2 and size of the problem decreases in following pattern : $n + n/2 + n/4 + \dots + 2 + 1$.

(ie) $n/2^k = 1$

$n = 2^k \Rightarrow k = \log_2 n$

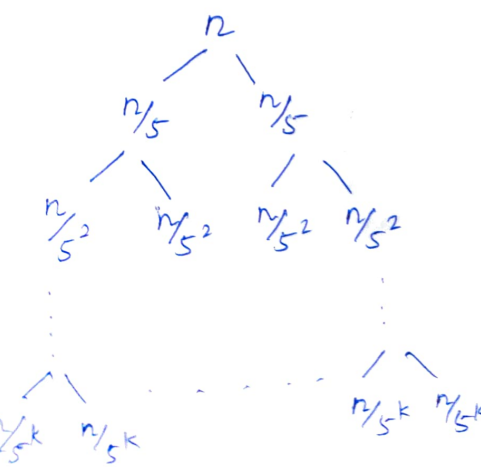
$x^y = z$.

$y = \log_z x$

* The amount of work done at each level is n. The total cost or work done at k levels is kn.

$$\text{Total cost} = k.n$$
$$= \log_2 n . n$$
$$= n \log_2 n$$
$$= O(n . \log_2 n)$$

Eg 2: Solve the recurrence eqn. $T(n) = 2.T(n/5) + n$.



level 0	n
level 1	$n/5 + n/5 = \frac{2}{5}n$
level 2	$n/5^2 + n/5^2 + n/5^2 + n/5^2 = \left(\frac{2}{5}\right)^2 n$
⋮	⋮
level k	$n/5^k + \dots + n/5^k = \left(\frac{2}{5}\right)^k . n$

$$\frac{n}{5^k} = 1 \Rightarrow n = 5^k \Rightarrow k = \log_5 n$$

$$\text{Total cost} = k\left[n + \left(\frac{2}{5}\right)n + \left(\frac{2}{5}\right)^2 n + \left(\frac{2}{5}\right)^3 n + \cdots + \left(\frac{2}{5}\right)^k n\right]$$

$$= n\left[1 + \frac{2}{5} + \frac{2^2}{5^2} + \frac{2^3}{5^3} + \cdots \frac{2^k}{5^k}\right]$$

$$1 + d + d^2 + \cdots = \frac{1}{1-d}$$

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

$$= n\left[\frac{1}{\left(\frac{5-2}{5}\right)}\right] = n\left[\frac{1}{[3/5]}\right]$$

$$= n \cdot \frac{5}{3}$$

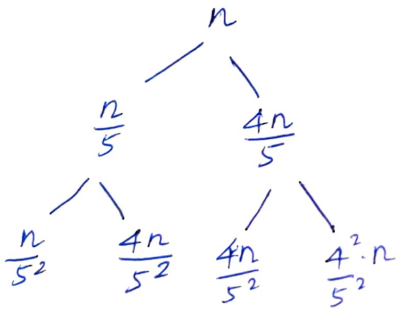
$$= k\left[n \cdot \frac{5}{3}\right]$$

$$= \log_5 n \cdot n\, 5/3$$

$$= \frac{5n}{3} \cdot \log_5 n$$

$$= O(n \log_5 n).$$

Eg 3. $T(n) = T(n/5) + T(4n/5) + n.$



$n = n/5$

$\frac{n/5}{5} = n/5^2$

$\frac{4n/5}{5} = 4n/5^2$

$\frac{\frac{4n}{5}}{5}$

$\frac{4\left(\frac{4n}{5}\right)}{5}$

$\frac{4^2 n}{5^2}$