Module-1

Introduction to Algorithm

Defination of Algorithm: The algorithm is defined as a collection of unambiguous instructions occurring in some specific sequence and such algorithm should produce output for given set of input in finite amount of time.

This defination of algorithm is represented Fig

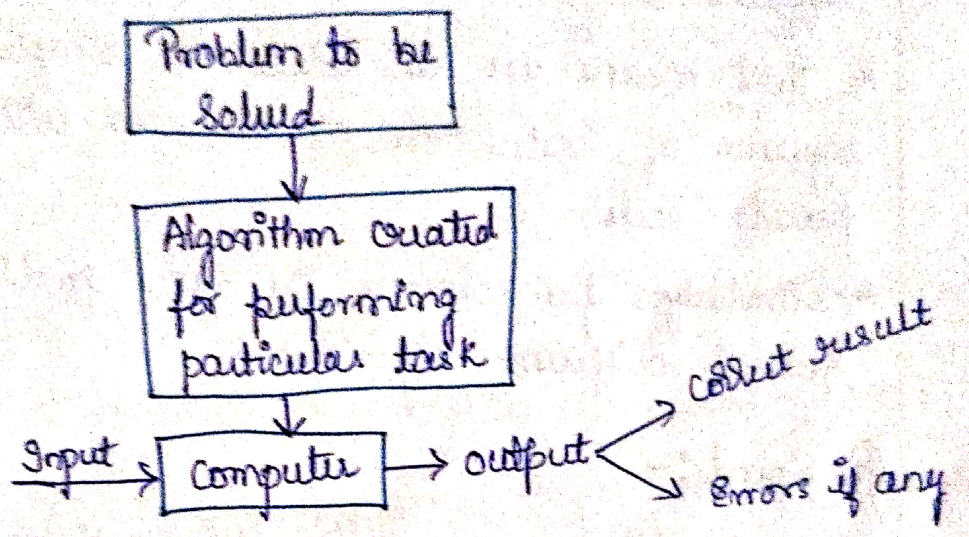


Fig: Notation of Algorithm.

Properties of Algorithm.

1. Non-ambiguity

* Each step in an algorithm should be non-ambiguous
* That means each instruction should be clear and precise
* The instruction in an algorithm should not be denote any conflicting meaning.
* This property also indicate the effectiveness of algorithm

② Range of input :-

* The range of input should be specified
* This is because normally the algorithm is input driven and if the range of the input is not been specified then algorithm can go in an infinite state.

③ Multiplicity :-

* The same algorithm can be represented into several diffrent ways.
* That means we can write in simple english the sequence of instruction or we can write it in form of pseudo code.
* Similarly, for solving the same problem we can write several diffrent algorithm.

④ Speed :-

*The algorithms are written using some specific ideas (which is popularly known as logics of algorithm)
* But such algorithms should be efficient and should produce the output with ~~speed~~ fast speed.

⑤ Fitness :-

* The algorithm should be finite. That means after performing requried operations it should terminate

Algorithm Specification :-

Algorithm is basically a sequence of instructions written in simple english language.

The algorithm is broadly divided into 2 sections

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 and assignment statement

Let us understand some rules for writting the algorithm.

① Algorithm is a procedure consisting of heading and body. The heading consists of keyword Algorithm and name of the algorithm and parameter list.

Syntax: Algorithm name (p1, p2 ..... pn)

- Algorithm: This keyword should be written first
- name: Here write the name of an algorithm
- (p1, p2 ..... pn): write parameters (if any)

② Then in the heading section we should write following things:

// Problem Description
// Input:
// output:

③ Then body of an algorithm is written, in which various programming constructs like if, for, while & 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 and not by digit. An identifier can be a combination of alpha-numeric string.

It is not necessary to write data types explicitly for identifier. It will be represented by content itself.

Basic data types used are Integer, float, char, Boolean and so on. The pointer type is also used to point memory location. The compound data type such as structure or record can also be used.

⑦ Using assignment operator ← an assignment statement can be given

For Instance:

Variable ← expression

⑧ There are other types of operators such as boolean operators such as true or false.

Logical operators such as and, or, not. And relational operators such as <, <=, >, >=, =, ≠

(9) The array indices are stored with in square brackets '[' ']'. The index of array usually start at zero. The multidemensional arrays can also be used in algorithm

(10) The Inputting and outputting can be done using read and write.

For example

```
write ("This message will be displayed on console");
read (val);
```

(11) The conditional statements such as if-then or if-then-else are written in following form.

```
if(condition)then statement
if (condition) then statement else statement.
```

If the If-then statement is of compound type then { and } should be used for enclosing block.

(12) while statement can be written as:

```
while (condition) do
  {
    Statement 1
    Statement 2
      .
      .
      .
    Statement n
  }
```

While the condition is true the block enclosed with { } gets executed otherwise statement after } will be executed.

(13) The general form for writing for loop is:

for variable ← value, to value_n do

{

Statement 1

Statement 2

:

Statement n

}

Here value, is initialization condition and value_n is a terminating condition.

Sometime a keyword step is used to denote increment or decrement the value of variable

for example

```
for i ← 1 to n step 1 ←
  {
     write (i)
  }
```

Here variable i is incremented by 1 at each iteration

(14) The repeat-until statement can be written as

```
repeat
     statement 1
     Statement 2
     :
     statement n
until (condition)
```

(15) The break statement is used to exit from inner loop. The return statement is used to return control from one point to another. Generally used while existing from function.

Some Examples

) Write an algorithm to count the sum of n numbers.

Algorithm Sum(1,n)

// Problem Description : The algorithm is finding the sum of given n number.

// Input : 1 to n number

// Output : The sum of n numbers

result ← 0

for i ← 1 to n do i ← i+1

result ← result + i

~~result~~ return result.

Write an algorithm for sorting the elements

Algorithm Sort (a,n)

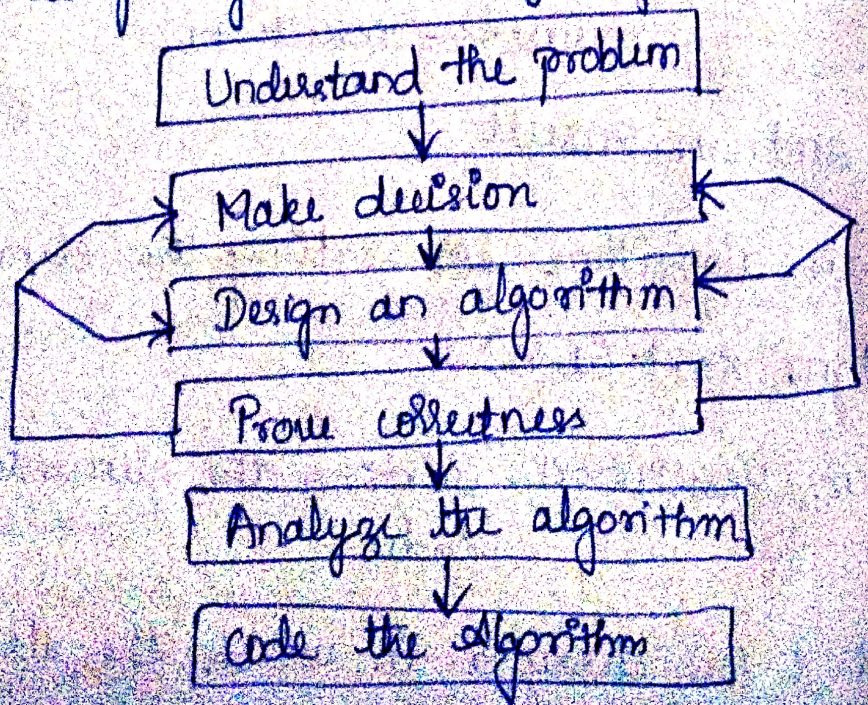
// Problem Description : Sorting the elements in ascending order

// Input : An array a in which the elements are stored and n is total number of elements in the array.

```
//Output: The sorted array.
for i ← 1 to n do
for j ← i+1 to n-1 do
{
  if (a[i] > a[j]) then
    {
      temp ← a[i]
      a[i] ← a[j]
      a[j] ← temp
    }
}
write ("List is sorted")
```

Fundamentals of Algorithmic problem solving

Steps for designing and analysing an algorithm



1. Understanding the problem:-

@ This is the first step in designing of algorithm
ⓑ Read the problem's description carefully to understand the problem statement completly.
© Ask questions for clarifying the doubt about the problem.
@ Identify the problem types and use existing algorithm to find solution
© Input (instance) to the problem and range of the input get fixed.

2. Make decision:

The decision making is done on the following
@ Ascertaining the capabilities of the computational device
* In random access machine (RAM) instructions are executed one after another.
* Accordingly, algorithms designed to be executed on such machines are called sequential algorithms.
* In some newer computers, operations are executed concurrently, i.e in parallel.
* Algorithms that take advantage of the capability are called parallel algorithms.
* Choice of computational devices like processor and memory is mainly based on the space & time efficiency

⑥ Choosing between exact and approximate problem solving

* The next principal decision is to choose to solving the problem exactly or solving it approximately.
* An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.
* If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm. it produces an approximate answer.
Eg: Extracting square root etc.

3. Design an algorithm.

* An algorithm design technique is general approach to solving problem authematically that is applicable to a variety of problems from different areas of computing

Algorithm + Data structures = Programs

* Though algorithm and data structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing

the algorithm.

* Implementation of algorithm is possible only with the help of algorithms and data structures.

* Algorithmic strategy / technique / paradigm are a general approach by which many problems can be solved algorithmically.

Eg: Brute force, Divide and Conquer, Dynamic Programming, Greedy technique.

Methods of Specifying an Algorithm

There are 3 ways to specify an algorithm.

(a) Natural language
(b) Pseudocode
(c) Flow chart.

(a) Natural language :-

It is very simple and easy to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear and thereby we get brief specifications.

Eg: An algorithm to perform addition of two number.

Step 1: Read the first number, say a
Step 2: Read the first number, say b.
Step 3: Add the above 2 numbers and store the result in c.
Step 4: Display the result in c.

* Such a specification creates difficulty while actually implementing it.
* Hence many programmers prefer to have specification of algorithm by means of pseudocode.

⑥ Pseudocode:

* Pseudocode is a mixture of a natural language & programming language constructs. Pseudocode is usually more precise than natural language.
* For Assignment operation left arrow "←" for comments & slashes "//". If condition, for, while loops are used.

Algorithm Sum(a, b)
// Problem Description: This algorithm performs addition of two numbers.
// Input : Two Integer a and b.
// Output : Addition of two Integer
c ← a+b
return c.

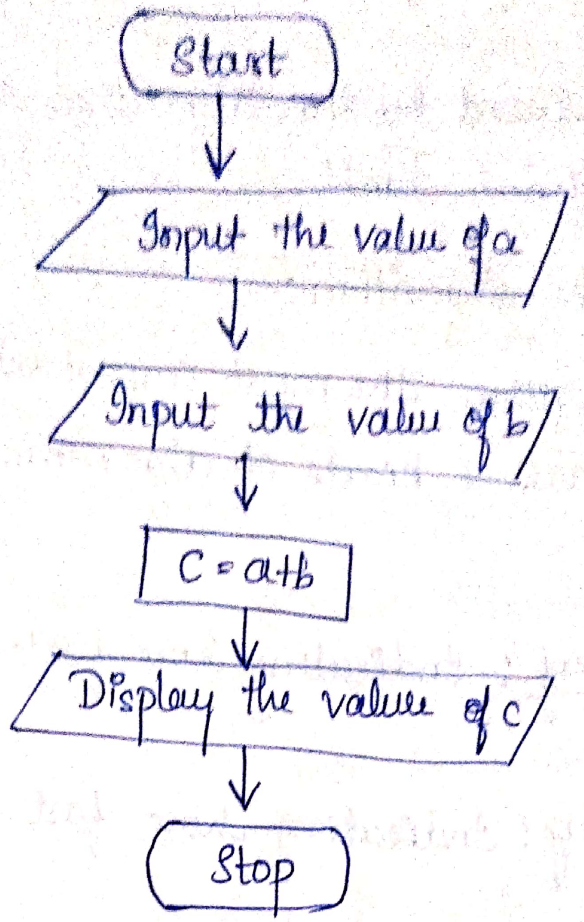
* This specification is more useful for implementation of any language

© Flow Chart

* Flow chart is a graphical representation of an algorithm. It is a method of expressing an algorithm by a collection of connected geometric shapes contain

descriptions of the algorithm's steps.

Example: Addition of a and b



4. Prove Correctness

* Once an algorithm has been specified then its correctness must be proved.
* An algorithm must yield a required result for every legitimate input in a finite amount of time.
* For example the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality $gcd(m,n) = gcd(n, m \bmod n)$.
* A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

* The notation of correctness for approximation algorithm is less straightforward than it is for exact algorithms.
* The error produced by the algorithm should not exceed a predefined limit.

⑤ Analyzing the Algorithm :-

* For an algorithm the most important is efficiency. In fact there are 2 kinds of algorithm efficiency. They are

ⓐ Time efficiency : Indicating how fast the algorithm runs.

ⓑ Space efficiency : Indicating how fast the algorithm it uses.

* The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.
* So factors to analyze an algorithm are:
  - → Time efficiency of an algorithm
  - → Space efficiency of an algorithm
  - → Simplicity of an algorithm.
  - → Generality of an algorithm.

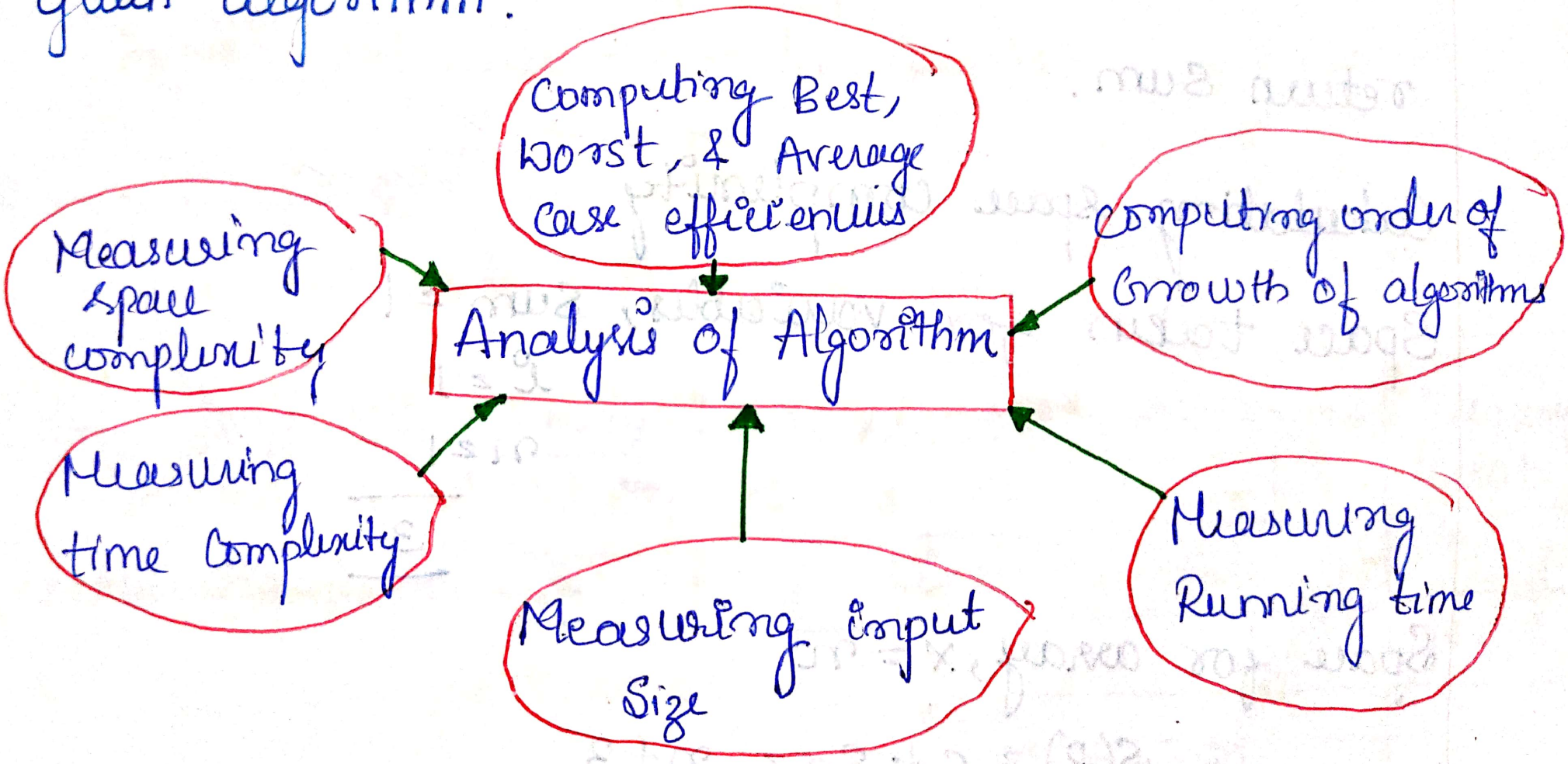
⑥ Coding an algorithm :-

* The coding / implementation of algorithm is done by suitable programming language like C, C++, JAVA

* The transition from an algorithm to a program can be done either incorrectly or very inefficiently, implementing an algorithm correctly is necessary. The algorithm power should not reduce by an efficient implementation.
* Standard tricks like computing loop's invariant outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.
* Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a ~~difference~~ difference in running time by order of magnitude. But once an algorithm is selected a 10-50% speedup may be worth an effort.
* It is very essential to write an optimized code to reduce the burden of compiler.

Analysis Framework

* It is a systematic approach applied for analyzing any given algorithm.



1. Measuring Space Complexity :-

Space Complexity

→ Amount of storage space or memory required to execute

Formula to measure space complexity

$$S(P) = C + S_p$$

Where,

$S(P)$ → Space Complexity of a program 'P'

C → Constants for Inputs and outputs.

S_p → Instance characteristics of a program

Example :-

```
Algorithm Add(x,n)
Sum ← 0
for i ← 1 to n do
        Sum ← Sum + x[i]
return Sum.
```

Calculating space complexity

Space taken for variables, Sum = 1

i = 1

n = 1

3

Space for array, x = n

$S(P) = C + S_p = n + 3$

2. Measuring Time Complexity :-

Time Complexity

- Amount of computer time required to completion.
- Difficult for physically clocked time.
- Multiuser systems executing time depends on
  - * System load
  - * Instruction set used
  - * Number of other programs running
  - * Speed of underlying hardware.
- Time complexity is given in terms of frequency count.

Frequency Count.

- How many numbers of times a statement gets executed
- Denoted by Big oh notation (O).

Example 1

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

Calculating time complexity :

Executable statement	Frequency count
i=0	1
i<n	n+1
i++	n
Sum = Sum + a[i]	n
Total	3n+2

Time Complexity, $O(n) = 3n+2$

Example 2:

Statements	Frequency Count		Total	
	n=0/1	n>1	n=0/1	n>1
Fibo(n)	–	–	–	–
{ if (n<=1) then	1	1	1	1
write(n);	1	0	1	0
else	–	–	–	–
{ a=0;	0	1	0	1
b=1;	0	1	0	1
for i=2 to n do	0	n	0	n
{ c=a+b;	0	n-1	0	n-1
b=a;	0	n-1	0	n-1
a=c;	0	n-1	0	n-1
} write(c);	0	1	0	1
} }	–	–	–	–
		Time Complexity	2	4n+1

Few rate n-1+1 = n

3. Measuring an Input Size :-

- The efficiency of an algorithm can be computed as a function
- Input size is passed as a parameter
- It can be
  - → Exact value
  - → Approximate value

Example

Spell - Checking Algorithm

→ Number of characters
→ Number of words
} Input size.

4. Measuring Running Time :-

From an algorithm

- Identify basic operation
- Understand the concept of basic operation.
- Compute total number of time taken by basic operation

Formula used is,

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

where,

$T(n)$ → Running time of basic operation.
C_{op} → Time taken by basic operation to execute.
$C(n)$ → Number of times the operation needs to be executed.

5. Computing Order of Growth.

- Measuring the performance of an algorithm in relation with the input size.

Example :

n	$\log n$ Just	$n \log n$ add	n^2 square	2^n square
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
.	.	.	.	.
.	.	.	.	.

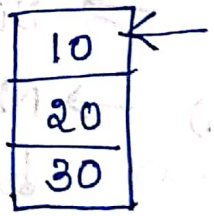
6. Computing Best, worst and Average case efficiencies

Best case efficiency.

- Efficiency of an algorithm for the best case input of size n
- Algorithm runs the fastest among all possible Input of that size n.

Example : Sequential Search(10)

If we search an element in a list of n elements and if the searched element present in the first position, then the running time is $C_{best}(n) = 1$



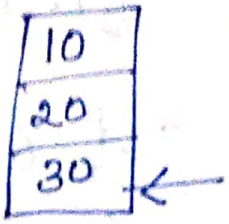
Worst Case Efficiency

- Efficiency of an algorithm for the worst case Input of size n.

- Algorithm runs the longest among all possible inputs of that size n.

Example : Sequential Search(30)

If we search an element in a list of n elements and if the searched element present in the last position, then the running time is $C_{worst}(n) = n$.

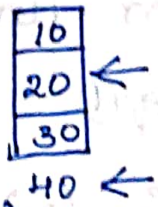


Average Case Efficiency

→ Efficiency of an algorithm for the average case lies between best and worst

→ To analyse it, we must make some assumptions about possible inputs of size n.

Example: Sequential search (20 and 40)



Two possibilities

(i) Probability of successful search (element 20)

(ii) Probability of unsuccessful search (element 40)

Let,

p → Probability of successful search

(1-p) → total number of unsuccessful search

n → total number of elements in the list.

$\frac{p}{n}$ → Probability of occuring first match for every i^{th} element.

$C_{avg}(n)$ = probability of successful + unsuccessful search

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

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

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

$$\therefore C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p) \longrightarrow (1)$$

For successful search of element do put $p=1$ in eq(1)

$$C_{avg}(n) = \frac{1(n+1)}{2} + n(1-1) = \frac{n+1}{2} \quad //\ \therefore$$

For unsuccessful search of element do put $p=0$ in eq(1)

$$C_{avg}(n) = \frac{0(n+1)}{2} + n(1-0) = \underline{\underline{n}}$$

Performance Analysis

① Space Complexity

② Time Complexity

① Space Complexity:- The space complexity of an algorithm is the amount of memory it needs to run to complete

Space needed by an algorithm is given by

$S(P) = C$(fixed part) + S_p(variable part)

Fixed part : independent of instance characteristics.

Eg: Space for simple variables, constants etc

Variable part : Space for variables whose size is dependent on particular problem instance.

```
1. algorithm Max(A,n)
2. // A is an array of size n.
3. {
4. Result := A[1];
5. for i:=2 to n do
6. if A[i] > Result then.
7. Result := A[i]
8. return Result;
9. }
```

Variables i, n, Result = 1 unit each

Variable A = n units.

Algo1 :-

```
Algorithm abc(a,b,c)
{
  return a+b*c+(a+b-c)/(a+b)+4.0;
}
```

a → 1
b → 1
c → 1

>= 3 units.

Algo 2 :-

```
Algorithm Sum(a, n)
{
  s = 0.0;
  for i = 1 to n do
  s = s + a[i]
  return s;
}
```

i, n, s → 1 unit each
a → n units
>= n+1 units.

Algo 3 :-

```
Algorithm RSum(a, n)
{
 If (n<=0) then return 0.0;
 else return Rsum(a, n-1) + a[n];
}
```

Rsum(a, n) → 1(a[n]) + 1(n) + 1(return) = 3 units
Rsum(a, n-1) → 1(a[n-1]) + 1(n) + 1(return)
⋮
Rsum(a, n-n) → 1(a[n-n]) + 1(n) + 1(return)

Total → >= 3(n+1) units.

② Time Complexity

- The time complexity of an algorithm is the amount of computer time it needs to run to complete

$T(P)$ = Compile time + execution time

$T(P) = tp$ (execution time)

Step count:

→ For algorithm heading → 0
→ For Braces → 0
→ For expressions → 1
→ For any looping statements → no. of times the loop is repeating.

Algorithm 1

1. Algorithm abc (a, b, c)	→ 0
2. {	→ 0
3. return a + b * c + (a+b-c)/(a+b)+4.0;	→ 1
4. }	→ 0
	1 unit

Algorithm 2

1. Algorithm sum (a, n)	→ 0
2. {	→ 0
3. S = 0.0;	→ 1
4. for i = 1 to n do	→ n+1
5. S = S + a[i];	→ n
6. return S;	→ 1
7. }	→ 0
	2n+3

Algorithm - 3

```
Algorithm Rsum(a, n)
{
if (n≤0) then return 0.0;
else return Rsum(a, n-1) + a[n];
}
```

$T(n) = 2$ if $n=0$

$= 2 + T(n-1)$ if $n>0$

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

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

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

$\vdots$

$= 2 * n + T(n-n) = 2n + T(0)$

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

Asymptotic Notations with examples :-

* whenever we want to perform analysis of an algorithm we need to calculate the complexity of that algorithm
* But when we calculate the complexity of an algorithm, it does not ~~present~~ provide the exact amount of resource required.
* So Instead of taking the exact amount of resource, we represent that complexity in a general form

which produces the basic nature of that algorithm.

* we use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

* For example, consider the following time complexities of 2 algorithm.

→ Algorithm 1 : $5n^2 + 2n + 1$

→ Algorithm 2 : $10n^2 + 8n + 3$.

* When we analyze an algorithm we consider the time complexity, for larger value of 'n' the term '2n+1' in algorithm 1 has least significant than the term $5n^2$ & the term $8n+3$ in algorithm 2 least significant is than the term $10n^2$.

* Here for larger value of 'n' the value of most significant terms ($5n^2$ and $10n^2$) is very large than the value of least significance term ($2n+1$ and $8n+3$).

* So for larger value of 'n' we ignore the least significant term ($5n^2$ and $10n^2$) is very larger than the value of least significant terms ($2n+1$ and $8n+3$). So for larger value of 'n' we ignore the least significant terms to represent the overall time required by an algorithm.

* In asymptotic notation we use only the most significant terms to represent the time complexity algorithm.

There are 3 types of Asymptotic notations they are:

① Big - oh (O)
② Big - Omega (Ω)
③ Big - Theta (Θ)

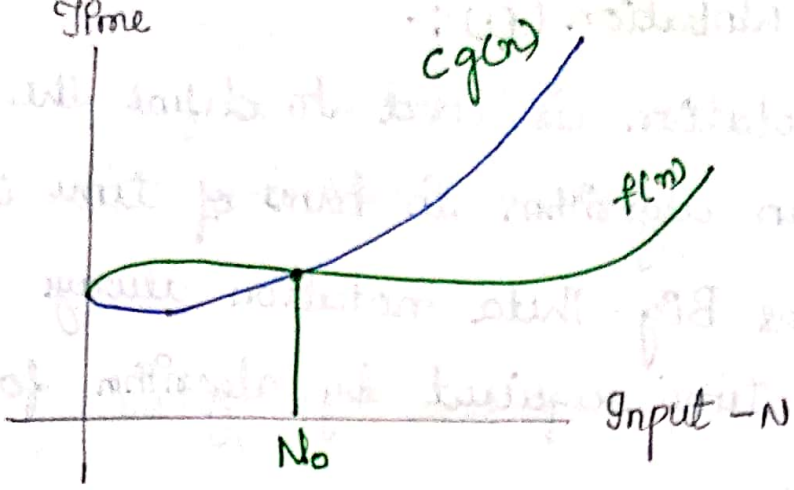
① Big-oh (O):-

* Big-oh (O) notation is used to define the upper bound of an algorithm in terms of time complexity.
* That means Big-oh notation always indicates the maximum time required by an algorithm for all input values.
* That means Big-oh notation describes the worst case of an algorithm time complexity
* Big-oh Notation can be defined as follows.

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If $f(n) <= c*g(n)$ for all $n >= n_0$, $c > 0$ and $n_0 >= 1$. Then we can represent f(n) as O(g(n)).

$$f(n) = O(g(n))$$

* Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-axis



* The above graph after a particular Input value no, always $cg(n)$ is greater than $f(n)$ which Indicates the algorithm's upper bound.

Example

Consider the following $f(n)$ and $g(n)$

$f(n) = 3n+2$

$g(n) = n$

If we want to represent $f(n)$ as $O(g(n))$ then it must satisfy $f(n) <= c\,g(n)$ for all values of $c>0$ and $n_0>=1$

$f(n) <= cg(n)$

$3n+2 <= cn$

$n=1$ ✗
$n=2$ ✗
$n=3$ ✗
$n=4$ ✓

Above condition is always TRUE for all values of $c=4$ and $n>=2$.

$3n+2 \le 4n$

$2 \le 4n-3n \quad n>=2.$

$2 \le n$

By using Big-oh notation we can represent the time complexity as follows.

$$\boxed{3n+2 <= O(n)}$$

$3n+2 \le c \cdot n$
$3n+2 \le 4n$
$2 \le 4n-3n$
$2 \le n$
$n \ge 2$

Big - Theta Notation (Θ) :-

Big-Theta notation is used to define the average bound of an algorithm in terms of time complexity.

* That means Big-Theta notation always indicates the average time required by algorithm for all input values.

* That means Big-Theta notation describes the average case of an algorithm time complexity.

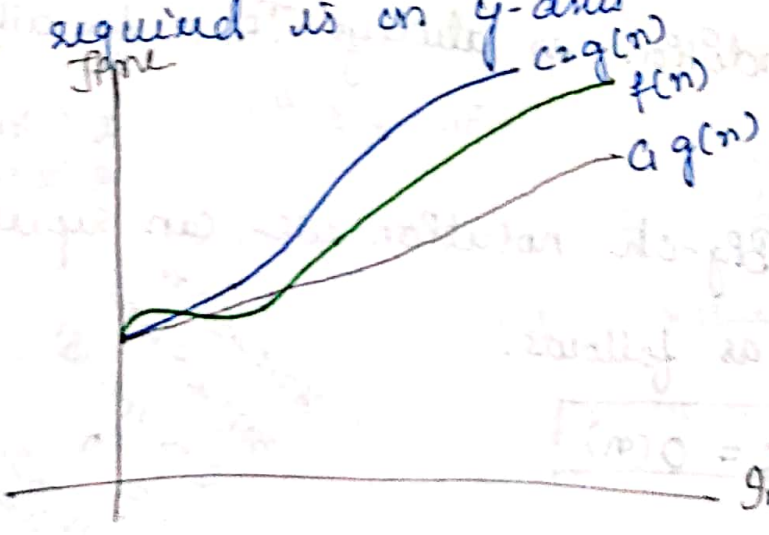
* Big-Theta Notation can be defined as follows.

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term.

If $C_1 g(n) <= f(n) <= C_2 g(n)$ for all $n >= n_0$, $C_1 > 0$, $C_2 > 0$ and $n_0 >= 1$. Then we can represent $f(n)$ as $\Theta(g(n))$

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

* Consider the following graph drawn for the values of $f(n)$ and $Cg(n)$ for input(n) value on x-axis and time required is on y-axis



In above graph after a particular input value n_0, always $C_1 g(n)$ is less than $f(n)$ and $C_2 g(n)$ is greater than $f(n)$ which indicates the algorithm's average bound.

Example

Consider the following $f(n)$ and $g(n)$

$f(n) = 3n+2$

$g(n) = n$.

If we want to represent $f(n)$ as $\Theta(g(n))$ then it must satisfy $C_1 g(n) <= f(n) <= C_2 g(n)$ for all value of $C_1 > 0$ $C_2 > 0$ and $n_0 >= 1$

$C_1 g(n) <= f(n) <= C_2 g(n)$

$= C_1 n <= 3n+2 <= C_2 n$

$C_1 n <= 3n+2$
$1n <= 3n+2$
$2 <= 3n - n$
$2 <= 2n$
$1 <= n$
$n >= 1$

$3n+2 <= C_2 g(n)$
$3n+2 <= C_2 n$
$3n+2 = 4n$
$2 <= n$
$n >= 2$.

Above condition is always TRUE for all values of $C_1 = 1$, $C_2 = 4$ and $n >= 2$.

By using Big - Theta notation we can represent the time complexity as follows.

$\boxed{3n+2 = \Theta(n)}$

Big - Omega Notation (Ω)

Big - Omega notation is used to define the lower bound of an algorithm in terms

Big oh notation (O):-

$f(n) = 2n^2 + n$

$f(n) = O(?)$ → time complexity

$2n^2 + n \leq c \cdot g(?)$

$2n^2 + n \leq c \cdot g(n^2)$

$f(n) = 2n^2 + n$

$g(n) = n^2$

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

$2n^2 + n \leq c \cdot n^2$

let $c = 1$ $(c > 0)$

$2n^2 + n \leq 1n^2$ ✗

let $c = 2$

$2n^2 + n \leq 2n^2$

$[2(1) + 1 \leq 2(1)$
$3 \leq 2$ ✗$]$

let $c = 3$

$2n^2 + n \leq 3n^2$ ✓

$\therefore \boxed{c = 3}$

$c = 4$ also work but we are not taking $c = 4$ because we are considering least upper bound

3, 4, 5, 6

$2n^2 + n \leq 3n^2 \Rightarrow 2n^2 + n \leq 3n^2$

$n \leq n^2$

If we divide with n on both sides

$n/n \leq n^2/n$

$1 \leq n$

$\therefore 2n^2+n \leq 3n^2$ is true for all $n \geq 1$

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

Omega Notation (Ω)

$f(n) = 2n^2+n$

$g(n) = n^2$

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

$2n^2+n \geq cn^2$

if $c=1$

$2n^2+n \geq 1n^2$ ✓

$[n=1, \; 2(1)+1 \geq 1, \; 3 \geq 1]$ ✓

if $c=2$

$2n^2+n \geq 2n^2$

$[n=2, \; 2(1)+1 \geq 2, \; 3 \geq 2]$ ✓

if $c=3$

$2n^2+n \geq 3n^2$

$[2(1)+1 \geq 3(1), \; 3 \geq 3 \quad 2(4)+1 \geq 6]$

Here we need to take 2 only Because in Ω lower bound but greatest lower bound.

$2n^2+n \geq 2n^2$

$2n^2+n = \Omega(n^2)$

3. Theta (Θ) Notation

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

$$c_1 \cdot n^2 \le 2n^2 + n \le c_2 \cdot n^2$$

$c_1 = 2$ & $c_2 = 3$

$$2 \cdot n^2 \le 2n^2 + n \le 3n^2$$

$c_1 \cdot n^2 \le 2n^2 + n$

$1 \cdot n^2 \le 2n^2 + n$

$4^2 \le 2(4) + 4$

$4 \le 10$ ✓

$2n^2 - 2n^2 \le n$

$0 \le n$

$2n^2 + n \le 3n^2$

$n \le 3n^2 - 2n^2$

$n \le n^2$

$1 \le n$

$n \ge 1$

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

$f(n) \ge c_1 g(n)$

$2n^2 + n \ge 1 \cdot n^2$

$2n^2 + n \ge n^2$

$n \ge n^2 - 2n^2$

$n \ge -n^2$

$f(n) \ge c_1 g n$

$3n + 2 \ge 1n$

$3n - 1n \ge -2$

$2n \ge -2$

$n \ge 1$

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

$3n + 2 \le 4n$

$2 \le 4 - 3$

$n \ge 2$

$f(n) \ge c_1 \cdot g(n)$

$3n + 2 \ge n$.

$2 \ge n - 3n$

$2 \ge -2n$

$n \le 1$

$c_1 g(n) \le 3n + 2 \le c_2 g(n)$ ∀ $n \ge 1$

$c_1 = 2$ & $c_2 = 3$ & $n \ge 3$

Basic Efficiency Classes

1. 1 or Constant

Whereas the time complexity of an algorithm is any constant value i $f(n) = 1$. If its a number $= 50$ $= 100$ then it will falls under a constant.

Eg:- Running time is constant.

2. log n

If the running time of the algorithm is $f(n) = \log N$ then running time is logarithmic

Problem size will be reduced by a constant factor (Initially n value the n value will be reduced by a constant factor) in each Iteration

Eg:- Binary search. → In binary search in each iteration the problem size n will be reduced by $n/2$

3. n

• Running time of an algorithm is $f(n) = n$ whereas the running time of an algorithm is N, it is linear.

The problem size is directly $\propto$ to the Input size N

Eg:- Linear search algorithm.

4. $n \log n$

- The running time of an algorithm is $n \log n$ is $f(n) = n \log n$
- Here the problem size is directly & to the input size n and also the logarithmic value of n)

Eg :- Quick Sort, Merge Sort, Heap Sort etc

5. n^2

The running time of an algorithm $f(n) = n^2$ then it is quadratic.

Generally the algorithm with 2 for loop or double for loop it falls under the category n^2

Eg: Bubble Sort, Matrix addition, Metrix Subtraction etc.

6. n^3

The running time of an algorithm is cubic

This algorithm will have 3 for loops

Eg :- Matrix Multiplication.

7. 2^n

If running time of an algorithm is 2^n then it is exponential

Eg:- Tower of Hanoi

8. $n!$

Running time is $n!$ (factorial)

Eg:- Algorithm generates permutation, combination etc

Mathematical Analysis of Non-recursive Algorithm

Example 1: Consider the problem of 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.
```

maxval = A[0] = 10

A[]	[0]	[1]	[2]	[3]	[4]
Elements	10	5	18	25	6

maxval = A[0] = 10

A[]	[0]	[1]	[2]	[3]	[4]
Elements	10	5	18	25	6

maxval = A[2] = 18

A[]	[0]	[1]	[2]	[3]	[4]
Elements	10	5	18	25	6

maxval = A[3] = 25

A[]	[0]	[1]	[2]	[3]	[4]
Elements	10	5	18	25	6

maxval = A[3] = 25

- Let us denote C(n) the number of times this comparision is executed
  - ↳ comparision is the main operation
  - ↳ How many times it will execute
    - C → comparision
    - n → instance characteristics
- The algorithm makes one comparison on each execution of the loop
  - ↳ i=1 one comparision
  - ↳ i=2 Another comparision
  - ↳ i=3 One comparision
  - ↳ Eg i=5 five no. of comparision

which is repeated for each value of the loop's variable i within the bounds 1 and n-1 (inclusively)

- Therefore, we get the following sum for C(n):

$$C(n) = \sum_{i=1}^{n-1} 1 = n-1-1+1 \quad \text{up} - \text{lb} + 1$$
$$= n-1 \in \Theta n$$

↳ comparision will execute only once so 1 is written

when we solve this upper limit - lower limit + 1
loop begin with 1 so n-1
of 0 n comparison

General Plan for Analyzing Time Efficiency of Non-recursive Algorithm.

① Decide on a parameter (or parameters) indicating an input's size
   ↳ what is the i/p size
   ↳ what is the parameter

② Identify the algorithm's basic operation (As a rule, it is located in its innermost loop)
   ↳ Identify the basic operation which is execute first
   ↳ It is located in inner most loop
   ↳ Eg if A[i] < maxval.

③ check whether the number of times the basic operation is executed depends only on the size of an input. If it is also depends on some additional property, the worst-case, average case, & if necessary, best case efficiencies have to be investigated seperately

④ Set up a sum expressing the number of times the algorithm's basic operation is executed

Example 2 (Uniqueness (refer PPT)

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

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

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

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

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

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

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

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

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

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

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

$$= \Theta(n^2)$$

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

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

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

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

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

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

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

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

Algorithm :- UniqueElement(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. and "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
```

0	1	2	3	4	5	6
5	8	9	7	10	3	2

n-2 ↓ (5) n-1 ↑ (6)

for i ← 0 to 7-2 = 5

for j ← 0+1 to 7-1
= 1 = 6

if A[0] = A[1]
5 = 8 ✗

j ← 1+1 to 6
= 2

if A[0] = A[2]
5 = 9 ✗

j = 3 to 6

if A[0] = A[3]
5 ≠ 7 ✗

j = 4 to 6

if A[0] = A[4]
5 ≠ 10 ✓

j = 5 to 6

if A[0] = A[5]
5 ≠ 3

j = 6 to 6.

A[0] = A[6]
5 ≠ 2 ✓

i ← 1 worst case

0	1	2	3	4	5	6
5	5	9	7	10	3	2

n-2 ↓ (5) n-1 ↑ (6)

for i ← 0 to 5

for j = 0+1 to 7-1
1 6.

if A[0] = A[1]
5 = 5 ✓

return true → Best case

Example 3:- Matrix Multiplication

```
Matrix Multiplication (A[0... n-1, 0,.. n-1], B[0.. n-1, 0 n-1])
// Multiplies 2 square matrices of order n by the
definition - based algorithm
// Input: Two n x 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.
```

Time Complexity

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

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

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

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

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

$\sum_{i=0}^{n-1} n^2$ $\quad [up - l + 1]$

$= [(n-1) - 0 + 1]\, n^2$

$= [n - 1 - 0 + 1]\, n^2$

$= [n]\, n^2$

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

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

Example 4:

Algorithm : Binary(n)

// Input : A positive decimal integer n

// Output : The number of binary digits in n's binary representation

Count ← 1

n ← $\lfloor n/2 \rfloor$

return count.

$$n = \sum_{i=1}^{\lfloor \log_2 n \rfloor} 1$$

= upper lm − low lm + 1

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

$T(n) \in \Theta(\log_2 n)$ ✓

~~* Notice that most frequently executed operation~~
~~just is not inside the while~~

Mathematical representation of recursive algorith

Algorithm Fact(n)

// Algorithm computes n! recursively

// Input : A non negative integer n

// Output : value of n!

~~if n<0~~ if n==0

return 1

else

return Fact(n-1) * n

return

~~State~~

F(3) Fact(2) * 3 2*3=6//

F(2) Fact(1) * 2 1*2=2

F(1) Fact(0) * 1 1*1=1

Fact(0) return 1

Recursive eqn Basic operation is multiplication

$n>1$ $M(n) = M(n-1) + 1$ → one multiplication

↳ if $n>0$

$n=1$ $M(n) = 0$ → if $n==0$.

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

Use method of backward substitution.

Put $n = n-1$ in eq (1)

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

$M(n-1) = M(n-2) + 1 \longrightarrow (2)$

Put $n = n-2$ in eq (1)

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

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

Put $n = n-3$ in eq (1)

$M(n-3) = M(n-4) + 1 \longrightarrow (4)$

⋮

Backward substitution.

(4) in (3)

$M(n-2) = M(n-4) + 1 + 1 \longrightarrow (5)$

Sub (5) in (2)

$M(n-1) = M(n-4) + 1 + 1 + 1 \longrightarrow (6)$

Sub (6) in (1)

$M(n) = M(n-4) + 1 + 1 + 1 + 1$

$M(n) = M(n-4) + 4$

For i^{th} term this can be written as

$$\boxed{M(n) = M(n-i) + i} \longrightarrow (7)$$

$M(n) = M(n-1) + 1$ → Genric Equation → (2)

$M(0) = 0$ → Base case

$M(n-i) = 0$

$n - i = 0$ comparision

$n = i$

so $M(0) = 0$

$n - i = 0$

$n = i$

Put $i = n$ in eq (2)

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

$= M(0) + n$

$= 0 + n$

$M(n) = n$

$M(n) = O(n)$

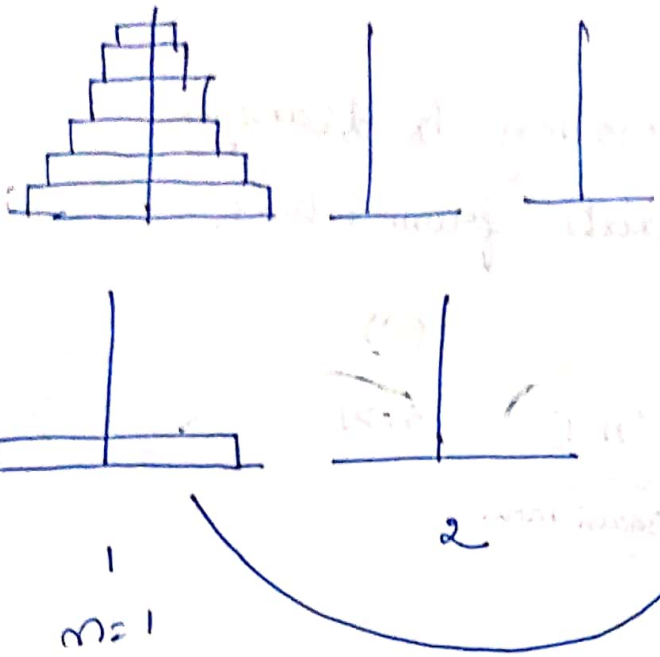
Tower of Hanoi

In this puzzle, there are n disks of diffrent size that can slide onto any of three peg.

Initially, all the disks are on the first peg in order of size, the largest on the bottom and smallest on the top.

The goal is to move all the disks to third peg using the second auxiliary, if necessary.

we can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.



if $n = 1$

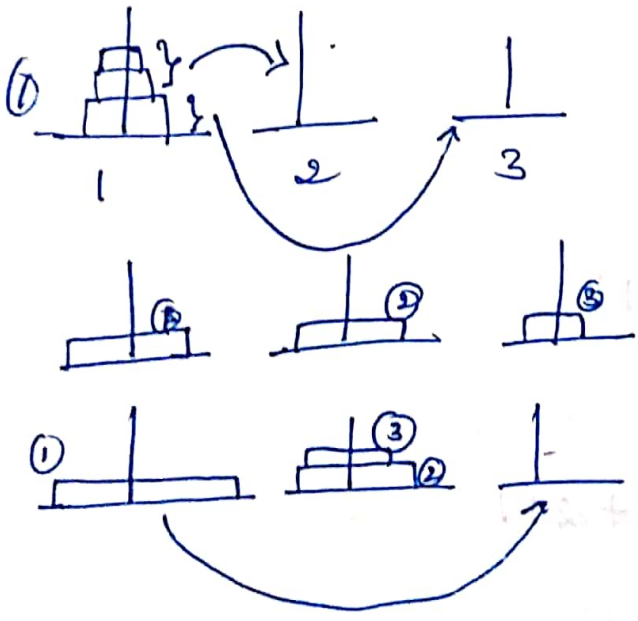
To move $n > 1$ disk from peg 1 to peg 3.

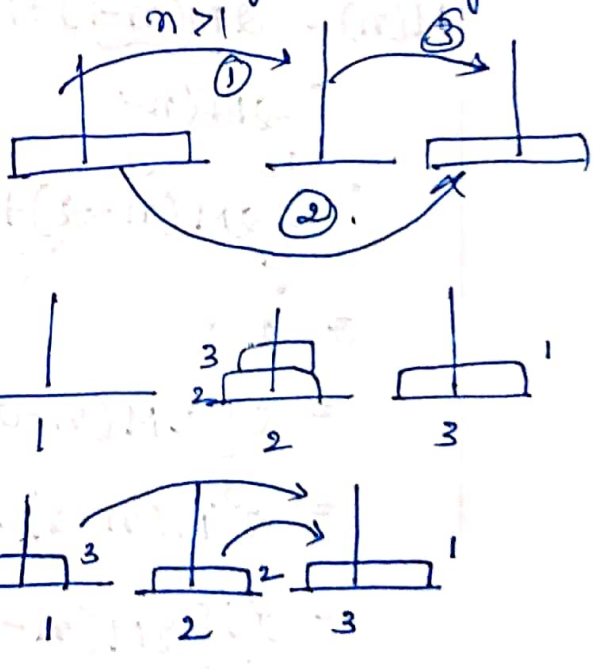
(1) Recursively move $n-1$ disk from peg 1 to peg 2. (Peg 3 as auxiliary)

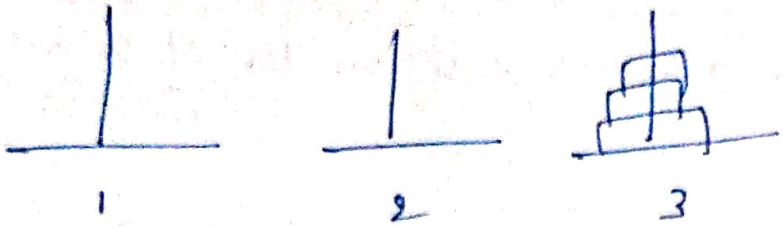
(2) Move largest disk from peg 1 to peg 3 directly

(3) Recursively move $n-1$ disks from peg 2 to peg 3 (Peg 1 as auxilary)

$n = 3$







⇒ Input size n

⇒ Basic operation – moving to transfer n disks from 1 to 3.

$M(n)$

$M(n) = \underbrace{M(n-1)}_{1 \text{ to } 2} + 1 + \underbrace{M(n-1)}_{\text{Again move}} \quad n>1$

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

$M(1) = 1 \quad n=1$

Backward Substitution $n>1$

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

$M(1) = 1$

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

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

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

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

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

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

After i sub

$$M(n) = 2^i m(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1$$

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

Initial condition $n = 1$

i.e. $i = n-1$

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

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

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

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

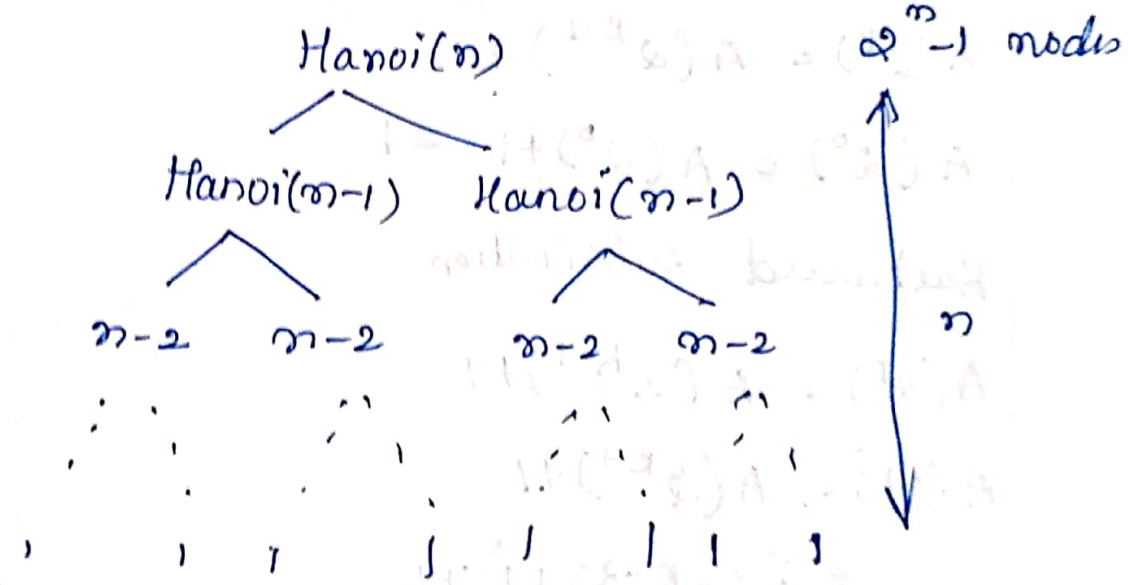
$2 \cdot 2^{n-1}$

2^{1+n-1}

2^n

$$\boxed{M(n) = 2^n - 1}$$ moves for n disks

Other way: Draw a tree of recursive calls.



$$C(n) = \sum_{l=0}^{n-1} 2^l \quad (\text{where } l \text{ is level as above}) = 2^n - 1$$

Example 3 :-

Counting bits in binary expression of a decimal number.

Algorithm BinRec(n)

// Input : A positive decimal integer n

// Output : The number of binary digits in n's binary representation

if n=1 return 1

else return BinRec($\lfloor n/2 \rfloor$)+1 (floor)

$A(n) = A(\lfloor n/2 \rfloor) + 1$

Assum n is a power of 2 : $n = 2^k$

$1 = 2^0$

$A(2^k) = A(2^{k-1}) + 1$

$A(2^0) = A(2^0) + 1 = 1$

Backward substitution

$A(2^k) = A(2^{k-1}) + 1$

$A(2^k) = A(2^{k-1}) + 1$

$= (A(2^{k-2}) + 1) + 1$

$= A(2^{k-2}) + 2$

$= A(2^{k-3}) + 3$

$= A(2^{k+1}) + 1$

$= A(2^{k-k}) + k \Rightarrow$ For k steps.

$= 1 + k$

After returning to the original variable $n = 2^k$ & hence $k = \log_2 n$

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

Brute Force Technique:-

* Important algorithm design technique.
* Brute force is a straight forward approach to solving a problem, usually direct based on the problem statement and definations of the concepts involved.

Eg:-

→ Algorithm to find GCD of 2 numbers
→ Matrix Multiplication
→ Selection Sort
→ Bubble Sort
→ Linear Search.

Sorting Problem:-

Given a list of n orderable elements, need to rearrange them in non-decreasing order.

• Two prime algorithms in Brute force Approach

① Selection Sort

② Bubble Sort.

① Selection Sort :-

→ Array of elements will be given

Eg: 50, 30, 25, 15, 10

→ Scan the entire list and find the smallest element

50, 30, 25, 15, (10)

→ Exchange it with the first element.

10, 30, 25, 15, 50

→ Again scan the list for next smallest element and exchange it with second element

→ 10, 15, 25, 30, 50

→ Continue the procedure until all elements get sorted

→ After n-1 passes, the list is sorted.

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

// Sorts a given array by selection sort

// Input: An array A[0...n-1] of orderable elements

// Output: Array A[0...n-1] sorted in nondecreasing order.

```
for i ← 0 to n-2 do
min ← i
for j ← i+1 to n-1 do
if A[j] < A[min]
    min ← j
Swap A[i] and A[min]
```

Selection Sort Example.

89 45 68 90 29 34 17

1 | 17 | 45 68 90 29 34 89

2 | 17 29 | 68 90 45 34 89

3 | 17 29 34 | 90 45 68 89.

4 | 17 29 34 45 | 90 68 89

5 | 17 29 34 45 68 90 89.

Let us trace the algorithm by consider one simple example

	0	1	2	3	4
Take	50	30	10	25	15

$i = 0$ to $5-2 = 3$

$min = 0$

$j = 1$ to $5-1 = 4$

$A[j] < A[0]$ = $A[1] < A[0]$

$30 < 50$ ✓

(b) 9, 43, 26, 4, 15

$i = 0$ to $n = 3$

min ⇐ 0

$j = 0+1 = 1$ to $n = 4$

A[1] < A[0]

43 < 9 X

$j = 2$

A[2] < A[0]

26 < 9 X

$j = 3$

A[3] < A[0]

4 < 9 ✓

min = 3

Swap A[0] & A[min]

0 1 2 3 4

4, 43, 26, 9, 15

$j = 4$

A[4] < A[0]

15 < 4 X

$i = 1$ to $n = 3$

min ⇐ 1

$j = 1+1 = 2$ to $n = 4$

A[2] < A[1]

26 < 43 ✓

min = 2

Swap A[1] & A[2]

~~9, 43~~ ~~4, 43~~

~~9, 26~~ 0 1 2 3 4

4, 26, 43, 9, 15

$j = 3$ to $n = 4$

A[3] < A[1]

9 < 26 ✓

min = 3

Swap A[i] & A[min]

A[1] & A[3]

0 1 2 3 4

4, 9, 43, 26, 15

$i = 2$ to $n = 3$

min ⇐ 2

$j = 2+1 = 3$ to $n = 4$

A[3] < A[2]

26 < 43 ✓

min ⇐ 3

Swap A[2] & A[3]

0 1 2 3 4

4, 9, 26, 43, 15

$j = 3+1 = 4$ to $n = 4$

~~A[3] < A[2]~~

A[4] < A[2]

15 < 26 ✓

min ⇒ 4

Swap A[2] & A[4]

0 1 2 3 4

4 9 ~~26~~ 15, 43, 26

$i = 3$

min ⇐ 3

$j = 3+1 = 4$

A[4] < A[3]

26 < 43 ✓

min = 4

Swap A[3] & A[4]

4, 9, 15, 26, 43

Pass ①, ②, ③

• Input Size : n

• Basic Operation : Key Comparision $A[j] < A[min]$

• The number of times it is executed depends only on the array size and is given by the following sum; with the help of 2 for loop.

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \quad (A[j] < A[min])$$

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

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

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

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

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

$$= \sum_{i=0}^{n-2} [n-3-i]$$

$$= [n-3-i][n-2-0+1] \qquad [n-1-1][n-1-0+1]$$

$$= [n-3-i][n-1] \qquad [n-1-i][n]$$

$$= n[n-3-i]-1[n-1] \qquad \frac{[n-1]n}{2}$$

$$= n^2-3n-$$

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

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

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

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

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

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

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

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

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

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

$$= \Theta(n^2)$$

Sequential Search :-

Algorithm SequentialSearch(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: The index of the first element in 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
```

a[0]	a[1]	a[2]	a[3]	a[4]
15	12	7	10	8

Key K = 7
n = 5

i = 0

0 < 5 & a[0] != 7
15 != 7 ✓

i = i+1
i = 0+1 = 1 ✓

1 < 5 & a[1] != 7
12 != 7 ✓

i = i+1
i = 1+1 = 2

2 < 5 ✓ & a[2] != 7
7 != 7 X

terminates while loop and moves to if condition

2 < 5 ✓

return 2 ⇒ index value where Key is matched.

If there is no match with the key element then else block will get executed.

Case

① Best case efficiency

Suppose if the key value is 15 and at the beginning itself we got the key value while loop executes only one time & directly move to if block.

0	1	2	3	4
15	12	7	10	8

$C_{best}(n) = 1$

② Average Case Efficiency.

Algorithm has to search almost half of the list

$C_{avg}(n) = \frac{n+1}{2}$ Here 5 elements

$\frac{5+1}{2} = \frac{6}{2}^{3} = 3$

③ Worst case Efficiency

$C_{worst}(n) = n$

Brute force string matching:-

Algorithm BruteForceStringMatch (T[0...n-1], P[0..m-1])

// Implements brute force string matching

// Input: An array T[0...n-1] of n characters representing a text and an array P[0...m-1] of m characters representing a pattern

// Output: The index of the first character in the text that starts a matching substring or -1 if the search is unsuccessful.

```
for i ← 0 to n-m do
    j ← 0
    while j < m and P[j] = T[i+j] do
        j ← j+1
    if j = m return i
return -1
```

NOBODY_NOTICED_HIM

NOT

NOT

NOT

NOT

NOT

NOT

NOT