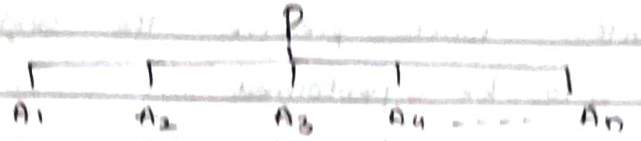
Design and Analysis of Algorithm

Camlin Page
Date / /

The problem (P) can have many solutions.
Let "P" be the problem and $A_1, A_2, A_3, \ldots A_n$ are the algorithms (Solutions)



Design represents how can we design different algorithms for a problem and analyze these algorithms in terms of memory used (Space) and execution time.

Algorithm	Program
→ Algorithm is written during design phase of a problem	→ Program is written during Coding or implementation phase
→ Hardware and OS independent	→ Hardware and OS dependent
→ Domain knowledge is required	→ Programming language is required
→ Algorithm can write in any language [English like langugage (or) mathematical notations]	→ Implemented Using Programming languages [C, Java, Python, etc].
→ An algorithm is analyzed after completion.	→ Program is tested after Completion.

Definition of Algorithm:

It is a step by step procedure for executing a program. (or) for finding a solution for a problem.

Characteristics of Algorithm:

1) Input : Algorithm must take zero (or) more input.
2) Output : Algorithm should produce atleast one output
3) Sequence : An algorithm must preserve the sequence of operations to be performed.
4) Definiteness : Algorithm must be clear an unambiguous
5) Finiteness : For all inputs algorithm should terminate after finite no of steps.
6) Effectiveness : Algorithm must produce correct output for all types of inputs.

Example :

Algorithm to swap two numbers

Method-1

```
Algorithm Swap(a,b)
{
  temp = a
   a = b
   b = temp
}
```

Method-2

```
Algorithm Swap(a, b)
begin
  temp ← a
  a ← b
  b ← temp
end
```

Write an algorithm to find largest element in an array.

Step-1: Start

Step-2: Read a[i]

Step-3: If a[i] > a[i-1] Goto to Step 4 else go to Step 5

Step-4: Print a[i] is largest element in an array then Goto to Step 6

Step-5: Print a[i-1] is largest element in an array then Goto to Step 6.

Step-6: Stop

Different approaches for designing an algorithm:-

- The process of dividing an algorithm into diff modules are known as modularization. Mainly 2 categories

i) Top-down approach

- It starts by dividing an algorithm into one (or) more sub modules. These modules are again divided into smaller units and this process is repeated till we reach desired granularity

ii) Bottom-up approach

- It starts from designing the module and then proceed towards higher modules. Submodules are combined to form larger modules.

CONTROL STRUCTURES USED IN ALGORITHM:-

i) Sequence:-

Where each and every step in the algorithm is executed one after the other.

ex:- addition of two numbers

Step1:- Start

2:- Read input values a, b,

3:- find the sum of two numbers

Sum = a+b

4:- Display the sum

5:- Stop

ii) Decission Making:-

Where execution of few statements depends on particular condition.

ex:- Odd (or) even checking

Step 1:- Start

Step 2:- Read input number 'a'

Step 3:- Check whether number 'a' is complete divisible by 2
If yes go to step 4. otherwise go to step 5

Step 4:- Display the numbes is even

Step 5:- Display the number is odd

Step 6:- Stop

iii) Repetition / Iterative :-

Where two statements will be executed till one particular condition becomes false

ex:- Display of numbers from 1 to n

Step 1:- Start

Step 2:- Initialize the variable i to 1

Step 3:- Read input 'n'.

Step 4:- Repeat step 5 and 6 till the 'i' value greater than n (i>n)

Step 5:- Display the value of i

Step 6:- Increment the loop variable i by 1

Step 7:- Stop.

Performance Analysis :-

The performance analysis of an algorithm mainly depends on two criteria.

i) The memory used by the algorithm [Space Complexity]

ii) The time required to execute the algorithm [Time Complexity]

We give more priority to time complexity than space Complexity. Why?

Space can be added (or) expanded if we want to increase but time cannot be add on.

Algorithm swap(a, b)

{

temp = a

a = b

b = temp

}

⇒ In the above algorithm we have three variables and the space complexity function $S(n) = 3$

⇒ Mathematically we use the function $f(n)$ to denote time complexity.

⇒ Each and every simple executable statement takes one unit of time.

⇒ In the above example, we have three statements So the time complexity function $f(n) = 3$.

Frequency Count Method for calculating the time Complexity function $f(n)$

Here, we mainly use two parameters,

1) Steps for execution which hold either a value 0 (or) 1. The value is 1 if it is an executable statement, 0 Otherwise.

2) Frequency:-

It represent no. of times each statement is executed. For non executable statements the frequency will be 0.

Based upon the algorithm steps each one may get different value but time complexity would be same.

```
Alg   Sum(a[], n)
{
  int sum = 0;
  int i;
for(i=0; i<n ; i++)
{
  Sum= Sum + a[i];
}
otherwise
return Sum;
}
```

S/e	frequency	S/e * frequency
0	0	0
1	1	1
0	0	0
1	n+1	n+1
0	0	0
1	n	n
0	0	0
1	1	1
0	0	0
		2n+3 $O(n)$

Using Frequency Count method find time Complexity function f(n) for matrix multiplication.

```
Alg mult matrix(m,n)
{
 int a[i][i];
 int b[j][j];
int m,n,k;
for(i=0;i<=m;i++)  → m+1
{
for(j=0;j<=m;j++) → m(m+1)
{
for(k=0;k<=m;k++) → m(m)(m+1)
    {
     c[i][j]=          → m(m)(m)
    }}}
```

S/e	frequency	s/e * frequency
0	0	0
0	0	0
0	0	0
0	0	0
1	m+1	m+1
0	0	0
1	m(m+1)	m(m+1)
0	0	m+1
1	m(m)(m+1)	m(m)(m+1)
0	0	0
1	m(m)(m)	m^3

$$[m+1]+[m^2+m]+[m^3+m^2]+[m^3]$$

$$2m^3+2m^2+2m+1=0$$

$$=O(m^3)$$

How can we say the algorithm is efficient?

Algorithm Efficiency:

If the algorithm doesnot contain any looping statement (or) recursive call [linear algorithm] then the efficiency of the algorithm can be given as no. of instructions it include. otherwise, the efficiency is calculated as follows.

i) Linear Loop:

In this case efficiency depends on how many times the statements in the loop will be executed.

ex:-

```
for(i=0; i<n; i++)
{
    Statement 1;
}
```

Here the statement inside the for loop will execute for n times. So $F(n) = n$.

ex:-

```
for(i=0; i<n; i=i+2)
{
    Statement;
}
```

In this loop the statement will be executed n/2 times.

So, function $F(n) = \frac{n}{2}$

ii) Logarithmic loop:

Here, loop controlling variable is either multiplied (or) divided in each iteration

eg:-

```
for (i=1 ; i<n ; i=i*2)
{
  Statement;
}
```

The i value varies as follows in each iteration

i

$i_1 = 1 = 2^0$

$i_2 = 2^0 \times 2 = 2^1$

$i_3 = 2^1 \times 2 = 2^2$

$i_4 = 2^2 \times 2 = 2^3$

$i = k = 2^k$

Let us consider i loop work k times

Termination condition for given loop is $i >= n$

While equating $i = n$.

We assume that $i = 2^k$.

$2^k = n$ So, $\boxed{k = \log_2 n}$ i.e., the statement inside the loop will execute $\log n$ times, So the function

$\boxed{f(n) = \log_2 n}$

eg:- `for (i=n ; i>1 ; i=i/2)`

```
{
  Statement;
}
```

i/n

$n/2$

$n/2^2$

⋮

$n/2^k$

$i <= 1$

$i = 1$

$n/2^k = 1$

$n = 2^k$

$k = \log_2 n$

$f(n) = \log_2 n$

iii) Nested loops

a) Linear Logarithmic :-

```
for(i=1; i<n; i++)      for i loop   f(n)=n
for(j=1; j<n; j=j*2)    for j loop   f(n)=log2 n
{
  Stmt 1;
}
```

$f(n) = n \log_2 n = O(n \log n)$

b) Quadratic loop :-

```
for(i=1; i<n; i++) -> i loop    f(n)=n
for(j=1; j<n; j++)  j loop      f(n)=n
{
  Stmt 1;
}
```

Quadratic loop $f(n) = n^2 = O(n^2)$

iii) Dependency quadratically :-

```
for(i=0; i<n; i++)
{
for(j=0; j<i; j++)
{
  Stmt 1;
}
}
```

Here value of j depend on i value whenever we have dependent loop we need to unfold dependency to find time complexity function (f(n))

Unfolding of loop

i	j	no. of times execution
0	0	0
1	0	1
	1	
2	0	2
	1	
	2	
3	0	3
	1	
	2	
	3	

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

Examples :-

i)
```
for (i= n/2 ; i<=n ; i++)
{
  for (j=1; j<=n/2 ; j++)
  {
   for(k=1; j<=n; k=k*2)
    {
      Stmt;
    }
  }
}
```

The statement inside.

i loop execute from $n/2$ to till n i.e., $n/2$ times

Statement inside

j loop execute from 1 to $n/2$ i.e., $n/2$ times

the statement inside k loop will execute $\log_2 n$ time

So, f(n) for above skiplet is $\frac{n}{2} \times \frac{n}{2} \times \log_2 n$

$= n^2 \log_2 n$

~~$O(n^2 \log n)$~~

$= O(n^2 \log n)$

ii)

```
for(i=1; i<=n; i++)
{
  for(j=1; j<=i; j++)
  {
    for(k=1; j<=100; k++)
    {
      Stmt 1;
    }
  }
}
```

Since there is dependency between j and we need to unfold the loop.

i	j	k
1	1	100
2	2	100*2
3	3	3x100
⋮	⋮	⋮
n	n	nx100

$100*1 + 100*2 + \cdots 100*n$

$= 100(1+2+\cdots n)$

$$f(n) = 100\left(\frac{n(n+1)}{2}\right) = O(n^2)$$

iii)

```
for(i=n/2; i<=n; i++)   i loop = n/2
{
  for(j=1; j<=n; j=j*2)   j loop = log2 n
  {
    for(k=1; k<=n; k=k*2)   k loop = log2 n
    {
      Stmt;
    }
  }
}
```

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

$$f(n) = \frac{n}{2} * \log_2 n * \log_2 n \quad = O(n \log n)$$

loop	f(n)	
for(i=1; i<n; i++)	n	O(n)
for(i=1; i<=n; i=i*2)	$\log_2 n$	$O(\log n)$
for(i=1; i<=n; i=i/2)	$\log n$	$O(\log n)$
for(i=n/2; i<n; i++)	n/2	O(n)
for(i=1; i<n; i=i+2)	n/2	O(n)

3)
```
for(i=1; i<=n; i++)
{
  for(j=1; j<=i^2; j++)
  {
    for(k=1; k<=n/2; k++)
    {
      Stmt;
    }
  }
}
```

It is dependent loop we unfold it as follows

i	j	k
1	1	$1 \times n/2$
2	4	$4 \times n/2$
!	!	!
n	n^2	$n^2 * n/2$

Total no. of executions $= 1 \times n/2 + 4 \times n/2 + 9 * n/2 + \dots n^2 * n/2$

$= (1 + 4 + 9 + \dots n^2) n/2$

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

Camlin | Page
Date / /

Exercise-1 on freq count

Using frequency count method find the total no. of steps required for given algorithm

```
Algorithm disp(i, x)
{                                   0
  int i=1, ------                   1
  while(i<=n) do ---  (n-i+1)       n+1
  {  ----                0          0
    x=x+1 ----1          (n-i)      n
    i=i+1 ---1           (n-i)      n
  }  ----                0          0
  return x  ---          1          1
}  ----                   0          0
```

s/e	frequency	s/e * frequency
0		
1	1	1
1	n+1	n+1
0		
1	n	n
1	n	n
0	0	
1	1	1
0	0	
		3n+3 = O(n)

ex-2: Using frequency count method find the time Complexity for n x n matrix addition

```
Algorithm (matrixadd n/n , a[n][n] , b[n][n])
{
 int i, j;                                  - - 1        1
 for (i=0; i<n; i++)                        - - - - 1    n+1
5 {                                         - - - - 0
  for (j=0; j<n; j++)                       - - - - 1    n*(n+1)
  {                                         - - - 0
  c[i][j] = a[i][j] + b[i][j] ;             - - 1        n * n
  }
10 }
}
```

S/e	frequency	S/e * frequency
1	1	1
1	n+1	n+1
0	-	-
1	n(n+1)	n^2+n
0	-	-
1	n*n	n^2
0	-	-
0	-	-
0	-	

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

Comlin Page
Date / /

essay

Assymptotic Notation :

These notations gives simple characterization of algorithm efficiency. This method describe the limiting behaviour. Mainly 5 types

i) BIG-OH (O)
ii) Big-Omega (Ω)
iii) Theta Notation (Θ)
iv) little-oh notation (o)
v) little-Omega notation (ω)

i) Big-Oh notation (O) :

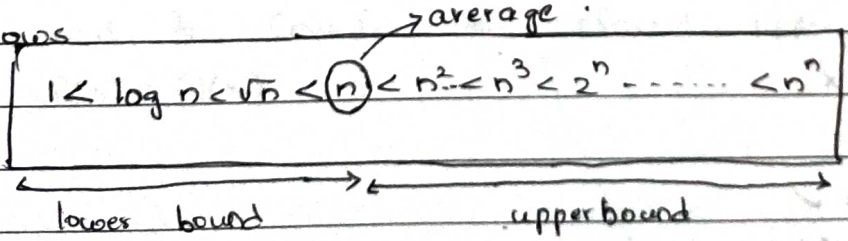
The function $f(n) = O(g(n))$ if and only if there exist a positive constant 'c' and n_0 such that $|f(n)| \leq c * |g(n)|$

⇒ Bigh-Oh notation specify the upper bound of a function
⇒ Based on Big-Oh values the algorithm can be categorized as follows :

Types Of Time Functions :

a) Constant time algorithm - $O(1)$
b) Linear time algorithm - $O(n)$
c) Logrithmic time algorithm - $O(\log n)$
d) Polynomal time algorithm - $O(n^k)$
e) exponental time algorithm - $O(2^n)$

The increasing order of time class function is as follows



Ex-1 :- Let us take $f(n) = 3n+2$

$3n+2 \leq c * g(n)$

** In Big-Oh notation we consider highest degree polynomial as $g(n)$ & c as coeff of highest degree polynomial + 1

In the above example $g(n)$ is n and c value = 3+1 = 4

$3n+2 \leq 4(n)$. We need to find out the value of n, which is called as "break even point" [Point (or) value from which equality condition is satisfying].

$3n+2 \leq 4(n)$

for $n=0$

$2 \leq 0$ ✗

for $n=1$

$5 \leq 4$ ✗

for $n=2$

$6+2 \leq 4(2)$ ✓ (True)

for $n=3$

$9+2 \leq 4(3)$ ✓ (True)

Since that point equality condition satisfy.

Example :-

$f(n) = 100n+2$

$100n+2 \leq c * g(n)$

$100n+2 \leq 101 * (n)$

for $n=0$

$2 \leq 0$ ✗

for $n=1$

$102 \leq 101$ ✗

for $n=2$

$202 \leq 202$ ✓

$f(n) = O(n)$

example :-

$f(n) = 10n^2 + 4n + 4$

$10n^2 + 4n + 4 \leq c * g(n)$

$10n^2 + 4n + 4 \leq 11(n^2)$

for $n = 0$.

$4 \leq 0$ ×

for $n = 1$

$18 \leq 11$ ×

for $n = 2$

$40 + 8 \leq 44$ ×

for $n = 3$

$90 + 16 \leq 99$ ×

for $n = 4$

$160 + 20 \leq 176$ ×

for $n = 5$.

$250 + 24 \leq 275$ ✓

for $n = 6$

$360 + 28 \leq 396$ ✓

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

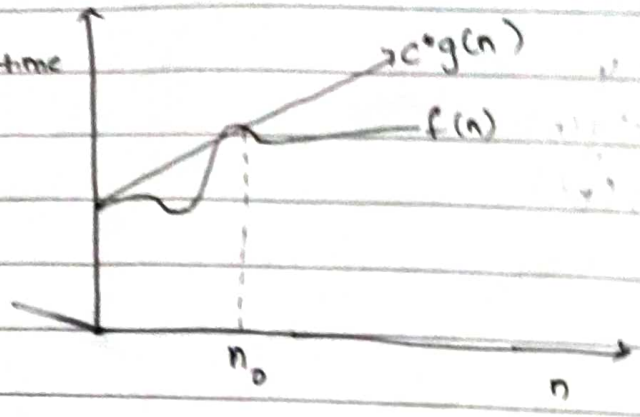
Do example 1 $f(n) = 3n + 2$ the value of n is bound to $f(n)$.

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

Since n is bound to $f(n)$ the values $n^2 < n^3 < \ldots 2^n$ also bound to $f(n)$.

So, $f(n) = O(n)$ (or) $f(n) = O(n^2)$ (or) $f(n) = O(n^3)$ but we need to consider the closest upper bound i.e., 'n'.

So, $f(n) = O(n)$



- Big-Oh notation gives the least upper bound

ii) Big-Omega Notation :-

The function $f(n) = \Omega(g(n))$ if and only if there exist two positive numbers c and n_0 such that $f(n) \geq c * g(n)$

- In Omega notation we consider g(n) as the highest degree polynomial and c as Coeff of that polynomial.

ex-1 :-

$f(n) = 3n+2$

$g(n) = n$

$c = 3$

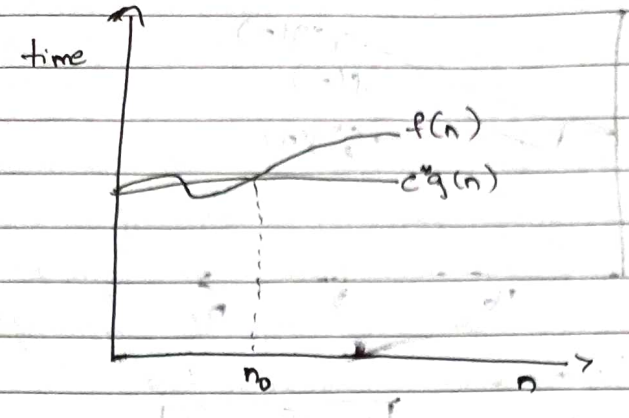
$3n+2 \geq 3n$

for $n=0$ ✓

$2 \geq 0$

for $n=1$

$5 \geq 3$ ✓



Omega notation represent the tightest lower bound

In example, $f(n) = 3n+2$ value of n is bound to f(n)

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

Since n is bound to f(n) values $1 < \log n < \sqrt{n} < n$ also bound to f(n)

So, $f(n) = \Omega(1)$ (or) $f(n) = \Omega(\log n)$ (or) $f(n) = \Omega(\sqrt{n})$ (or) $f(n) = \Omega(n)$

but we need to consider the closest lower bound.

So, $f(n) = \Theta(n)$ $\Omega(n)$

iii) Theta Notation (Θ):-

function $\boxed{f(n) = \Theta(g(n))}$ if and only if there exist three positive constants c_1, c_2 and n_0 such that

$$\boxed{c_1 * |g(n)| \leq |f(n)| \leq c_2 * |g(n)|}$$

Ex:- $f(n) = 3n+2$.

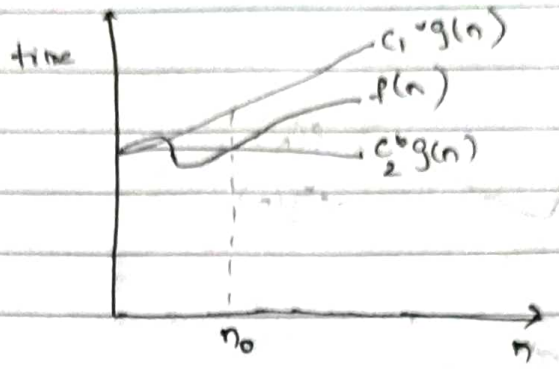
$$\underbrace{c_1 * |g(n)| \leq |f(n)|}_{\text{Big Omega}} \underbrace{\leq c_2 * |g(n)|}_{\text{Big-Oh}}$$

$3 * n \leq 3n+2 \leq 4 * n$

c_1 $g(n)$ c_2 $g(n)$

Already we found the value of the f(n)

Since $g(n) = n$ on both sides $f(n) = \Theta(n)$



iv) little-Oh(o) :- The function $f(n) = o(g(n))$ if and only if there exists the positive constant c and n_0 such that $f(n) < c * g(n)$

v) little-omega(ω) :- The function $f(n) = \omega(g(n))$ if and only if there exists two positive constants such that $f(n) > c * g(n)$

Between

i) Best Case :- The minimum time taken by an algorithm to complete its execution.

Worst case :- The maximum time taken by an algorithm to complete its execution.

ex :- In case of linear search consider the array

a[i] = | 2 | 4 | 10 | 5 | 7 |

If we want to search a value into this array. If value is present in a[0] itself. So for the input 2 time taken by algorithm is considered as best case of this algorithm

i.e., within one iteration we find the answer. So, best case for linear search is $O(1)$

If the number to be searched is 7 (or) a number which is not present in the array we need to search the entire array to find the result. So, the time taken by the algorithm depends on the size of array 'n' which we consider as the maximum time taken by an algorithm. ie., worst case complexity.

Worst Case Complexity of linear Search is $O(n)$

If the number to be searched is 10 we need to perform the search process till the middle of the array. The size is 'n' the average time is $n/2$. So for a linear search average case time complexity is $O(n/2) = O(n)$

Examples :-

1) $f(n) = 2n^2 + 3n$

$g(n) = n^2$ $O(n^2)$

Constant values will be ignored because purpose of Big-Oh notation is to analyse the algorithm in general fashion.

2) $f(n) = 2^{100}$

$O(1)$ - Constant time algorithm

3) $f(n) = 2^{3n+3}$

$O(2^n)$ - exponential.

4) $f(n) = 3\log n + 10\log n + 3$

$O(\log n)$

Space Complexity:

Space Complexity of a program S(P) can be calculated

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

C → independent variable memory space.

Independent variable also known as normal variable.

Sp → Dependent variable memory space.

Dependent variable also known as instance variable.

The memory space used by independent variable will be fixed and that of dependent variable will be varying.

$$S(P) = \text{fixed path} + \text{variable path}$$

Find Space Complexity for Sum of array elements.

```
Alg Sum(a[], n)
{
 int sum=0;
 int i;
 for (i=0; i<n; i++)
 {
  sum = sum + a[i];
 }
 return sum;
}
```

⇒ In above piece of code independent variables are sum, i, n. for uniformity let us consider each word take one word space. So, the value C (or) fixed path = 3 words.

⇒ The size of the array depends upon value of n. So, this is a dependent variable.

Variable part (or) $S_p = n$. So, the space complexity for the above algorithm is $3+n$.

Here, the memory requirement is increasing linearly with the input size 'n'. So, it is called as "Linear Space Complexity"

$S(P) = 3+n \rightarrow O(n)$ for above algorithm.

ALGORITHM SPECIFICATIONS:

Algorithm can be represented

1) Simple english: (using simple english)

We are specifying the operations clearly, without following any programming syntax

ex: Addition of 2 numbers

Step 1: Start

Step 2: Read the two numbers

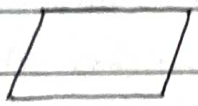
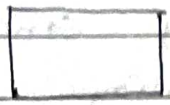
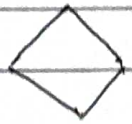
Step 3: Find the sum of two numbers by adding first number with the second number.

Step 4: Display the sum

Step 5: Stop.

2) FLOWCHART:

Flowchart is pictorial representation of algorithm

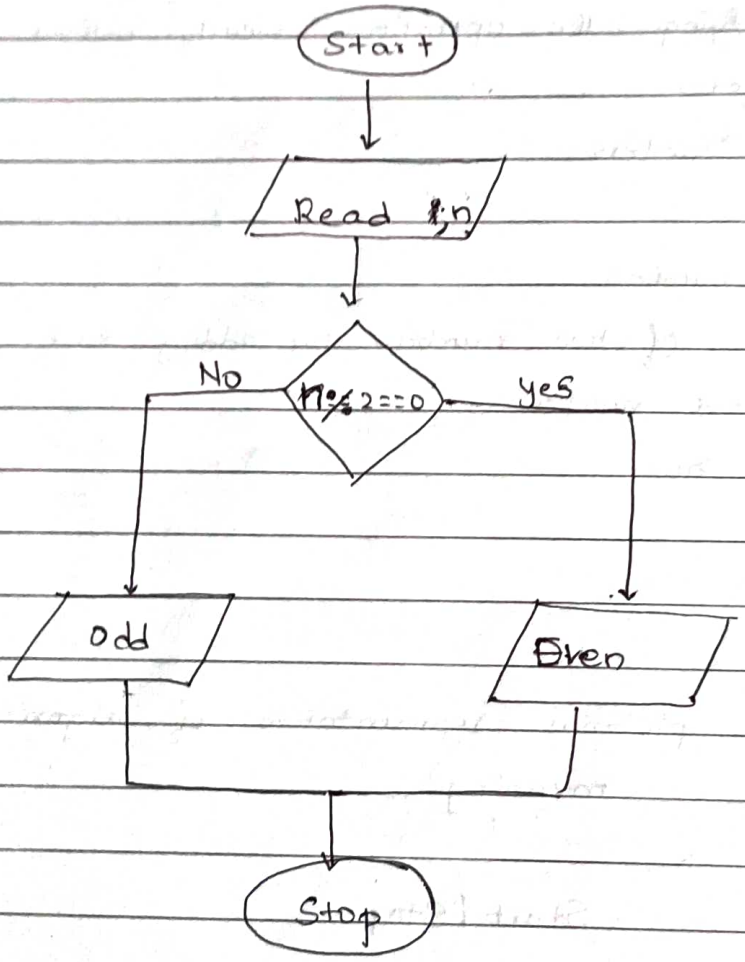
Symbol	meaning
	Start / Stop
	Input / output
	Processing
	decission making

↑↓→← flow of data

o Connector

Example :-

Draw the flowchart to check whether given number is odd / even.



3) Pseudocode :-

It is an informal way of programming description that doesnot require any strict programming syntax for consideration.

Rules for writing Pseudocode :

1) Comments begins with //
2) Blocks are indicated either using matching braces (or) begin, end.
3) Statements are delimited by semicolon.
4) Identifier begins with a letter, data type of identifier will not be specified.
5) Assignment of values to a variable is done using assignment operator. `variable = value;`
6) Array are represented single dimensional, multi dimensional
   a[i] - one Dimensional array
   a[i,j] - two dimensional array
7) The following statements are used as looping statements.

```
while (condition) do
{
  Stmt 1;
  .
  .
  Stmt n;
}
```

```
for var = v1 to vn then
do
{
  Stmt 1;
  .
  .
  Stmt n;
}
```

```
Repeat
  Stmt 1;
  .
  .
  Stmt n;
until(condition);
```

8) A conditional statements has following forms :

a)
```
if (Condition)
{
  Stmt 1;
}
```

b) if (condition)
{
Stmt 1;
}
else
{
Stmt 2;
}

a) The heading of a pseudocode takes the form
Algorithm name (parameter list)

example:-
Write Pseudocode for finding factorial for a given number.

```
Algorithm factorial(n, fact, i)
{
long fact = 1;
int i, n;
for (i=0; i<n; i++)
{
  fact = fact * i;
}
return fact;
}
```

Recursive Algorithm:

An algorithm is said to be recursive if the same algorithm is invoked by itself until a particular condition occurs.

Design Methodology & Implementation of Recursive Algo:

⇒ Each call to a recursive algorithm either solve one part of the problem (or) it reduces the size of the problem.

⇒ The statement that solve the problem is known as base condition (or) base case (or) halting case

⇒ Every recursive algorithm must have a base case

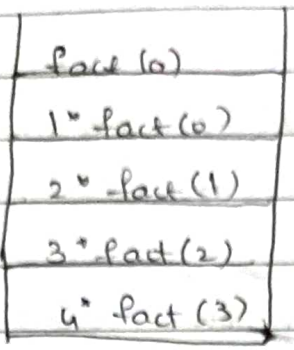
⇒ The rest part of the algorithm is known as general case which contains the logic needed to reduce size of the problem.

=

Write a recursive Algorithm to find factorial of a given number.

```
Algorithm fact(n)
{
  if (n==0) // halting case
  return 1;
  else
  return n*fact(n-1); // general case
}
```

To the above algorithm if we take the input as 4, the stack entry is as below:



For the value n=4, we have 5 stack entry
If the value n=7, we have 8 stack entry
=> In general for input n, depth of stack = n+1 when we calculate Space Complexity for a recursive algorithm, the following factors has to be taken into a count.

=> 1) Depth of stack
2) no. of formal parameters
3) no. of actual parameters
4) return address.

In the above example for each value of n.
Space Complexity Sp = 2*(n+1) → stack n.
1 word for return address
1 word for argument.

Calculating Time Complexity of Recurrence Relation
=> In recurrence relation, we represent the time complexity function as t(n).
=> We use back substitution method for solving recurrence relation.

$t(n) = 1 \quad n=0$

$T(n) = T(n-1) + 1 \quad n>0$

Find time Complexity of recurrence relation

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

$T(n) = T(n-1) + 1$ ——— (I)

Calculate the value of $T(n-1)$ Substitute $(n-1)$ in the above equation (in place of n)

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

$T(n-1) = T(n-2) + 1$ ——— (II)

Substitute the value of $T(n-1)$ in the equation (I)

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

$T(n) = T(n-2) + 2$ ——— (III)

Find out $T(n-2)$ by placing the value $n-2$ in place of n in equation (I)

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

$T(n-2) = T(n-3) + 1$ — (IV)

Substitute value of $T(n-2)$ in equation (III) $T(n-3)+3$ $T(n-k)$

If we observe the $T(n)$ value it varies as follows

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

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

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

⋮

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

Let us consider this process continues till k times

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

We are assuming $n-k=0$ (In the eq^n smallest value of n is given as zero)

$\therefore \boxed{n=k}$

Substitute k as n in above geeral equation

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

$= T(0) + n$

$= 1 + n$

In the question $T(0)$ is given as 1.

So, $T(n) = 1 + n$

which is $O(n)$.

Ex 2: $T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+n & n>0 \end{cases}$

$T(n) = T(n-1) + n$ —— (I)

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

$= T(n-2) + n-1$ — (II)

(2) in (1)

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

$= T(n-2) + 2n-1$ — (III)

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

$= T(n-3) + n-2$ — (IV)

put (IV) in (III)

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

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

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

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

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

⋮

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

Assume $n-k=0$

$n = k$

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

$= T(0) + 1 + 2 + \cdots n.$

$= 1 + 1 + 2 + \cdots + n$

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

Ex: $T(n) = \begin{cases} 1 & n=1 \\ T(n/2)+1 & n>1 \end{cases}$

$T(n) = T(n/2) + 1 \longrightarrow (I)$

$T(n/2) = T(n/4) + 1 \longrightarrow (II)$

II in I

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

$= T(n/4) + 2 \longrightarrow (III)$

$T(n/4) = T(n/8) + 1 \longrightarrow (IV)$

(IV) in (III)

$T(n) = T(n/8) + 2 + 1$

$= T(n/8) + 3$

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

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

$T(n) = T(n/8) + 3$

$\vdots$

$T(n) = T(n/2^k) + k$

$n/2^k = 1$

$n = 2^k$

$T(n) = T(n/n) + \log_2 n$

$= T(1) + \log_2 n$

$= 1 + \log_2 n$

$= O(\log n)$

Camlin
Date

Ex:- $T(n) = \begin{cases} 1 & n=1 \\ 2T(n/2)+n & n>1 \end{cases}$

$T(n) = 2T(n/2) + n$ —— (I)

$T(n/2) = 2T(n/4) + n/2$ —— (II)

(II) in (I)

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

$= 4T(n/4) + n + n$

$= 4T(n/4) + 2n$ —— (III)

$= 2T(n/2) + n$ —— (IV)

$T(n/4) = 2T(n/8) + n/4$ — (IV)

(IV) in (III)

$T(n) = 8T(n/8) + n/4 + 2n$

$= 8T(n/8) + 3n$ — (V)

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

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

$T(n) = 8T(n/8) + 3n$

$T(n) = 2^k T(n/2^k) + kn$

$n/2^k = 1$

$2^k = n$

$k = \log n$

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

$= nT(1) + n\log n$

$= n + n\log n$

$O(n \log n)$ //

$$T(n) = \begin{cases} 1 & n=0 \\ 2T(n-1)+1 & n>0 \end{cases}$$

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

$n = n-1, n-2, n-3, \ldots$

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

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

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

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

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

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

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

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

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

Assume, $n-k=0 \Rightarrow k=n$

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

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

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

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

$= O(2^n)$

21-3-22

$$T(n) = \begin{cases} 1 & n=1 \\ 2T(n-3) & n>1 \end{cases}$$

$T(n) = 2T(n-3)$ ——— (1)

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

$= 2T(n-6)$ ——— (2)

(2) in (1)

$T(n) = 2[2T(n-6)]$

$= 4T(n-6)$ ——— (3)

$T(n-6) = 2T(n-6-3)$

$= 2T(n-9)$ ——— (4)

(4) in (3)

$T(n) = 4[2T(n-9)]$

$= 8T(n-9)$ ——— (5)

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

$T(n) = 4T(n-6)$

$T(n) = 8T(n-9)$

⋮

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

$n-3k = 1$

$n = 3k+1$

$n = 3k+1$

$\frac{n-1}{3} = k$

$T(n) = 2^{n/3} T(n-r$

$T(n) = 2^{n-1/3} T(n-(n-1))$

$= 2^{n-1/3} T(1)$

$= 2^{n-1/3}(1)$

$= 2^{n-1/3}$

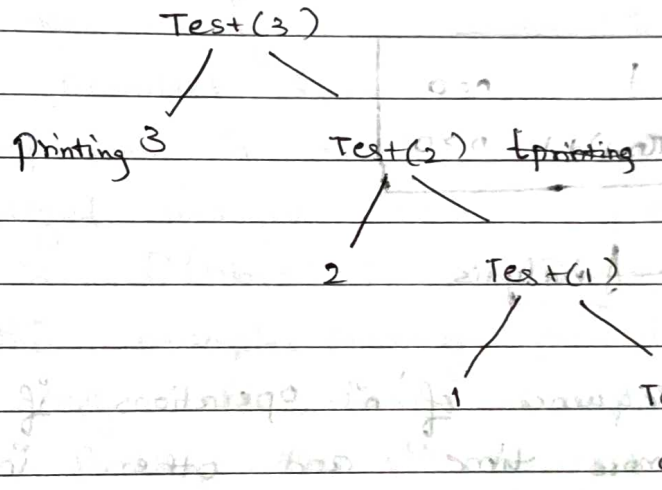
$= O(2^n)$

Solving recurrence relation using tree method

Consider the code

```
Algm Test(n)
{
  if(n>0)
  {
  print(n);
  Test(n-1);
  }
}
```

If the value of n=3 the recurrence tree for the above code is as shown below



⇒ Depth of the tree for n input is n+1. i.e., when n value is 3, test function is being called 4 times. Similarly, n value is 7 function will be called for 8 times. So, In general the value n test algorithm is being called for n+1 times. So, time complexity: O(n)

⇒ For recursive program we need to write recurrence relation

1) This algorithm do nothing when value of n=0 $T(n) = 1$ for $n=0$ No operation is being done when $n=0$ but we need to represent this using a constant value like 1, a, c...

2) If the time Complexity of test algorithm is represented as $T(n)$, in the above code test(n-1) is being called for each $n>0$ value of So, time Complexity of test(n-1) is represented as $T(n-1)$

3) Inside the if condition we have a print statement (which takes a constant time) and the recurrence call test(n-1) (Time Complexity $T(n-1)$)
So, $T(n-1)+1$ for $n>0$

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

Amortized Analysis

⇒ In sequence of 'n' operations, if some inputs are taking more time and other inputs are taking less execution time we need to perform amortized analysis rather than generalized analysis

⇒ Aggregate analysis method is the simplest way to perform amortized analysis which Consider the average time taken

⇒ In this method, initially we found out the time taken by costlier operation and time taken by cheaper operation.

⇒ We find out average time

= Cost of Costlier operation + Cost of cheaper operation Total no. of Operations

Example :-

If we have an input array of size 'n' we need to find 'k' Smallest element.

⇒ In order to perform this we need to sort the input array & let us assume that we are using quick Sort method for this Sorting procedure

⇒ Time Complexity of quick Sort is $O(n \log n)$

⇒ Once if the array is sorted in order to pick the k^{th} Smallest item, we need Constant time let us represent it as 1.

⇒ In generalized analysis time complexity of the above problem is $O(n \log n) + 1 = O(n \log n)$.

⇒ But in this example, Sorting procedure [Costlier operation] performed only for first element. Once the array is Sorted, finding the k^{th} Smallest item, can be done in Constant time.

Total Cost = $\underbrace{n \log n + 1}_{\text{first element}} + \underbrace{1 + 1 + 1 + \cdots + (n-1)}_{\text{for remaing } (n-1) \text{ elements}}$

⇒ let us assume we are performing the Constant time taken operation for n elements. So,

total cost : $n \log n + n$

⇒ Avg time = $\frac{\text{Total cost}}{\text{Total no. of operations}}$

$= \frac{n \log n + n}{n}$

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

$= \log n + 1$

$= O(\log n)$

Time Complexity is $O(\log n)$

Introduction

- Alg / Program
- defintion
- Characteristics

Diff approaches

- Top down
- Bottom up

Performance Analysis

- Time Complexity
  - Freq Count method
- Space Complexity
  - Dependent
  - independent

Alg efficiency

- linear
- log
- Nested
  - linear
  - Quadratic
  - dependent quadrig

Assymptotic Notation

- Big Oh
- Big Omega
- Theta
- little Oh
- little omega

Camlin Page
Date / /

Alg Specifications
i) Simple english ii) Flowchart iii) Pseudo Code

Recursive Alg

Recurrence relation using tree method

Time Complexity using recurrence relation

Amortized Analysis
definition
example

Using frequency Count method find time Coroplexity of below algorithm.

```
Algorithm Count(n)                    S/e          freq          Total
{                                      1            n+1           n+1
  for i=1 to n do                      1            n             n
  {                                  =0 | ≠0      =0 | ≠0       =0 | ≠0
    if (i%2==0)                       1 | -       n/2 | -        n/2 | -
      x=x+1                           - | 1        - | n/2         - | n/2
  else                                1            1             1
    y=y+1                                                       ------
  }                                                             2n+2+n/2
  return;
}
```

$n/2$ (at any time exited if / else)