UNIT I

What is an Algorithm?

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

In addition, all algorithms must satisfy the following criteria:

① Input : Zero or more quantities are externally supplied.

② Output : At least one quantity is produced.

③ Definiteness : Each instruction is clear and unambiguous.
meaning that it must be perfectly clear what should be done.
Ex:- compute 5/0 not permitted.

④ Finiteness : If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite no. of steps.
Ex :- chess.

⑤ Effectiveness :- Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper.

Each operation must be definite and also feasible.
Algorithms that are definite and effective (effective) are also called computational procedures.

There are 4 distinct areas of study one can identify:

1) How to devise algorithms :- Study various design techniques such as divide-and-conquer, greedy method, dynamic programming etc which help in algorithm formulation. Do not assume that each algorithm must derive from only a single technique. Analyze each solution for each of the problems.

2) How to validate algorithms :- Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. A proof of correctness requires that the solution be stated in 2 forms:
   i) A program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus.
   ii) The second form is called a specification, this may also be expressed in the predicate calculus.

   A proof consists of showing that these 2 forms are equivalent, i.e. for every legal input, they describe the same output.

3) How to analyze algorithms :- Analysis of alg. refers to the task of determining how much computing time and storage an algorithm requires - efficiency of algorithms.

4) How to test a program :- Testing of program has 2 phases: debugging and profiling (performance measurement)
   Debugging is the process of executing programs on sample data sets to determine whether ~~faculty~~ faulty results occur if so, correct them.

Profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results. (2)

Algorithm Specifications

pseudocode

1) // comments
2) { } block of stmts
   ; stmts delimited

3) assignment
   <value> := <expression>;

4) array [and]
   A[i, j] denotes (i, j)th element of array A.

5) loops

while

```
while <condition> do
{
    <Stmt 1>
    .
    .
    <Stmt n>
}
```

for

```
for variable := value1 to value2 step <step> do
{
    <Stmt 1>
    .
    .
    <Stmt n>
}
```

For loop can be Implemented as while loop as;

```
Variable := value1;
fin := value2;
Incr := step;
while ((value - fin) * step <= 0) do
{
    <stmt 1>
    :
    <stmt n>
    variable := variable + incr;
}
```

repeat - until

```
repeat
    <stmt 1>
    :
    <stmt n>
until <condition>
```

Conditional

```
if <condition> then <stmt>
if <condition> then <stmt 1> else <stmt 2>
```

Case

```
{ : <condition 1> : <stmt 1>
                :
  : <condition n> : <stmt n>
  : else : <stmt n+1>
}
```

I/p & o/p → read and write

GCD of two numbers

Greatest common Divisor of two or more integers, when at least one of them is not zero, is the largest +ve integer that divides the numbers without a remainder.

for eg :- GCD of 8 and 12 is 4

Alg.1 : Simple
Alg.2 : Euclid

```
Euclid (m, n)
{
  while 'm' does not divide n
     r = n mod m
     n = m
     m = r
  end
  return m
}
```

Alg 1 :-

1. Factorize m : Find primes $m_1, m_2 \ldots m_k$

$$m = m_1 \times m_2 \times m_3 \cdots \times m_k$$

2. Factorize n : Find primes $n_1, n_2 \ldots n_l$

$$n = n_1 \times n_2 \times n_3 \cdots \times n_l$$

3. Identify common factors
   multiply and return result.

Eg :-
$m = 36, n = 48$

$m = 2 \times 2 \times 3 \times 3$

$n = 2 \times 2 \times 2 \times 2 \times 3$

gcd(36, 48)
= 12

9 divisions

m n
36 divides 48 ×
r = 48 mod 36
= 12
n = 36
m = 12
12 divides 36 ✓
return 12

2 divisions

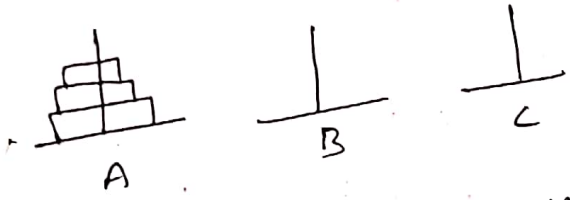
Recursive algorithm

An algorithm is said to be recursive if the same algorithm is invoked in the body.

Direct recursive : alg. calls itself.

Indirect recursive : It calls another algorithm which in turn calls it.

Ex: Towers of Hanoi

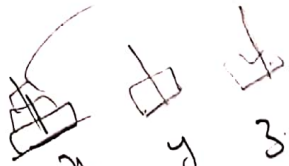


Move disks from A to B

- Moved only one at a time.
- larger (Bigger) disk cannot be on top of smaller one.

Recursive Solution :

- Assume no. of disks is n
- To get the largest disk to the bottom of tower B we move remaining n-1 disks to tower C and then move the largest disk to tower B.
- Now we are left with the task of moving the disks from tower C to tower B. To do this we have towers A and B available.




```
1  ALGORITHM TowersOfHanoi (n, x, y, z)
2  // Move the top n disks from tower x to tower y
3  {
4      if (n >= 1) then
5      {
6          TowersOfHanoi (n-1, x, z, y);
7          write ("move top disk from tower", x, "to top
8                                  of tower", y);
9          TowersOfHanoi (n-1, z, y, x);
10     }
11 }
```

x, z, y

→ T(3, a, b, c)

④ move a to b

T(2, c, b, a)

⑥ move c to b

T(1, a, b, c)

③ move a to b

T(0, c, b, a) ↑

T(0, a, c, b) ↑

T(1, c, a, b)

⑤ move disk c to a

T(0, b, a, c) ↑

T(0, c, b, a) ↑

T(2, a, c, b)

② move from a to c

T(1, b, c, a)

③ move from b to c

T(0, a, c, b) ↑

T(0, b, a, c) ↑

T(1, a, b, c)

① move from a to b

T(0, c, b, a) ↑

T(0, a, c, b) ↑

The space requirement $S(P)$ of any algorithm P is

$$S(P) = C + S_p(\text{instance characteristics}),$$

where C is a constant.

S_p is instance characteristics.

The space needed by each algorithms is the sum of the following components:

1. A fixed part i.e. independent of the characteristics (eg. number size) of the inputs and outputs.

This part typically includes

i) the instruction space (i.e. space for the code)

ii) space for simple variables and fixed size component variables (also called aggregate)

iii) space for constants

So on..

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics) and the recursion stack space

The recursion stack space includes space for the formal parameters, the local variables and the return address.

Ex:-

Algorithm:

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

Performance Analysis

There are many criteria upon which we can judge an algorithm:

1. Does it do what we want it to do?
2. Does it work according to original specifications correctly?
3. Is there documentation that describes how to use it?
4. Are procedures created in such a way that they perform logical subfunction?
5. Is the code readable?

These are Indirect criteria, the direct criteria is to check their computing time and storage requirements to judge their performance.

Space complexity of an algorithm is the amount of memory it needs to run to completion.

Time complexity is the amount of computer time it needs to run to completion.

Performance evaluation can be divided in 2 phases:

1) Priori estimates - performance analysis

2) Posterior Testing - performance measurement.

Priori means we do analysis (space and time) of an alg. prior to running it on specific system.

Posterior - we do analysis of algorithm only after running it on system. It directly depends on system and changes from system to system.

Ex 1.

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

Here the problem instance is characterized by the specific values of a, b, c.

abc is independent of instance characteristics.

Making the assumption that one word is adequate to store the values of each of a, b, c. S_p(instance characteristics) = 0

$S(P) = 3$

Ex 2.

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

Here the space needed by n is one word.
Since it is of type Integer.
The space needed by array a of n elements is n words.

n, i, s — each one word total er 3

$\therefore S_{sum}(n) \geq (n+3)$

Space needed to store a[] = n floating point words (or at least n words)

Space needed to store n and s = 2 words

$S_p(n) = (n+3)$. Hence $S(P) \geq (n+3)$

Ex 3 :- Algorithm RSum(a, n)
{
if (n ≤ 0) then return 0.0;
else return RSum(a, n-1) + a[n];
}

(b) The recursion stack space includes space for the formal parameters, the local variables and the return address.
Assume that the return address requires only one word of memory.
Each call to RSum requires at least 3 words ~~(including space)~~.

1 for n
1 for return address
1 for pointer to a[]

depth of recursion is n+1
∴ Recursion stack space for RSum ≥ 3(n+1)

Time Complexity

→ The time T(P) taken by a program P is the sum of the compile time and the run time (execution time).

→ The compile time does not depend on the instance characteristics.

Also, we may assume that a compiled program will be run several times without recompilation. The runtime is denoted by t_p (instance characteristics).

→ $t_p(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \cdots$

where n denotes the instance characteristics, and c_a, c_s, c_m, c_d respectively denote the time needed for an addition, subtraction, multiplication and division.

ADD, SUB, MUL, DIV are functions whose values are the no. of additions, subtractions, multiplications, divisions and so on when the code for P is used on an instance with characteristic n.

Obtaining such an exact formula is an impossible task since the time needed for the above operations depends on the compiler, the number being added and the processor capacity.

We might as well dump all the operations together (provided that the time required by each is relatively independent of the instance characteristics) and obtain a count for the total no. of operations.

— We can go one step further and count only the no. of program steps.

→ A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.

Ex: return a+b+b*c + (a+b-c)/(a+b) + 4.0

The entire statement can be regarded as a step since its execution time is independent of the instance characteristics.

→ The no. of steps any program statement is assigned depends on the kind of statement.

Ex: - Comments count as zero steps.

- assignment statement which does not involve any calls to other algorithms is counted as one step.
- for, while and repeat until, we consider the step counts only for the control part of the statement.

We can determine the no. of steps needed by a program to solve a particular problem instance in one of two ways.

(1) A new variable, count, is introduced into the program. This is a global variable with initial value 0. Statements to increment count by the appropriate amount are introduced into the ~~proc~~ program. This is done so that each time a statement in the original program is executed, count is incremented by the step count of that statement.

Time complexity:

The time T(P) taken by a program P is the sum of the compile time and the run (or execution) time. Also, we may assume that a compiled program will be run several times without recompilation. We concern ourselves with just the run time of a program. This run time is denoted by tp (instance characteristics).

A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.

The no. of steps any program statement is assigned depends on the kind of statement.

for. Eg:- comments count as zero steps;

An assignment statement which does not involve any calls to other algorithms is counted as one step;

In an iterative statement such as the for, while and repeat until statements, we consider the step counts only for the control part of the statement. The control parts for for and while stmts have the following forms:

for i := <expr> to <expr1> do

while (<expr>) do

Each execution of the control part of a while stmt is given a step count equal to the no. of step counts assignable to <expr>.

The step count for each execution of the control part of a for stmt is one, unless the count attributable to <expr> and <expr1> are functions of the instance characteristics.

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

```
Algorithm Sum(a, n)
{
    s := 0.0 ;
    count := count + 1 ;                    2n+3
    for i := 1 to n do
    {
        count := count + 1 ; // For for
        s := s + a[i] ; count := count + 1 ; // for assignment
    }
    count := count + 1 ; // For last time of for
    count := count + 1 ; // For the return
    return s ;
}
```

Ex 2: Recursion

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

```
Algorithm RSum(a, n)
{
    count := count + 1;
    if (n ≤ 0) then
    {
        count := count + 1; // for the return
        return 0.0;
    }
    else
    {
        count := count + 1; // for the addition, fun.
                            // invocation and return
        return RSum(a, n-1) + a[n];
    }
}
```

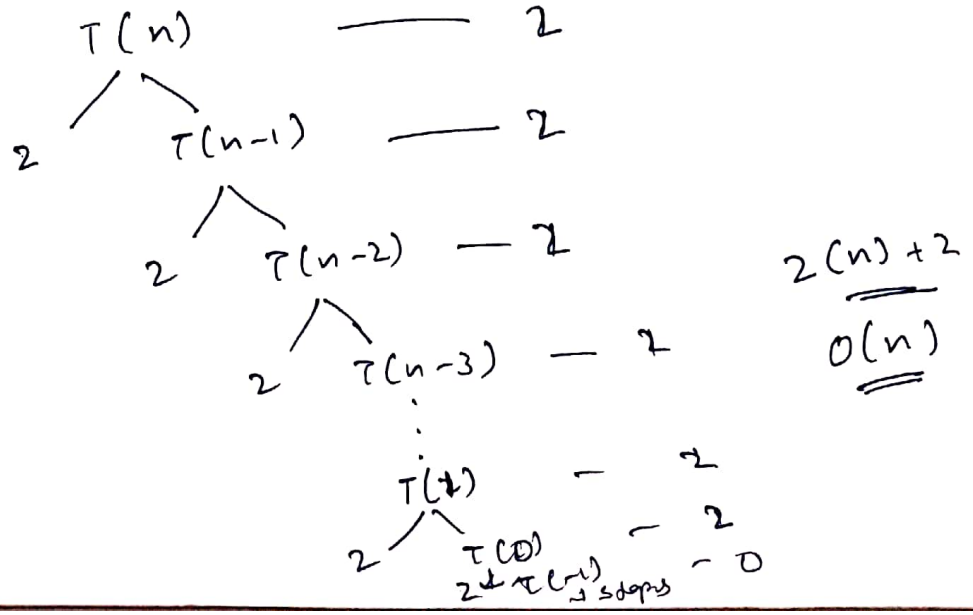
$$t_{RSum}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{RSum}(n-1) & \text{if } n > 0 \end{cases}$$

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

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

$$= n(2) + t_{RSum}(0)$$

$$= 2n + 2 \qquad n \geq 0$$



The second method to determine the step count of an alg. is to build a table in which we list the total no. of steps contributed by each statement. This figure is often arrived at by first determining the no. of steps per execution (s/e) of the stmt and the total no. of times each stmt is executed.

The s/e of a stmt is the amount by which the count changes as a result of the execution of that statement.

By combining these two quantities, the total contributions of all statements, the step count for the entire alg. is obtained.

Ex 1

Statement	s/e	frequency	total steps
Algorithm Sum(a,n)	0	–	0
{	0	–	0
s := 0.0;	1	1	1
for i := 1 to n do	1	n+1	n+1
s := s + a[i];	1	n	n
return s;	1	1	1
}	0	0 –	0
total			2n+2

Here the frequency of ~~each of~~ the ~~statements in~~ for statement is n+1 and not n. This is so. because i has to be incremented to n+1 before the for ~~statement is n+1 and not n~~ loop can terminate.

Recursion

Statement	s/e	Frequency n=0	Frequency n>0	total steps n=0	total steps n>0
Algorithm RSum(a,n)	0	-	-	0	0
{	0	-	-	0	0
if (n ≤ 0) then	1	1	1	1	1
return 0.0;	1	1	0	1	0
else return RSum(a,n-1)+a[n];	1+x	0	1	0	1+x
}	0	-	-	0	0
total				2	2+x

$x = t_{RSum}(n-1)$

MATRIX Add

Statement	s/e	frequency	total steps
Algorithm Add(a,b,c,m,n)	0	-	0
{	0	-	0
for i:=1 to m do	1	m+1	m+1
for j:=1 to n do	1	m(n+1)	mn+m
c[i,j] := a[i,j]+b[i,j];	1	mn	mn
}	0	-	0
total			2mn+2m+1

Matrix Addition Algorithm

```
Algorithm Add(a,b,c,m,n)
{
   for i:=1 to m do
     for j:=1 to n do
        c[i,j] := a[i,j] + b[i,j];
}
```

by introducing statements to increment count :

⇒

```
Algorithm Add(a,b,c,m,n)
{
   for i:=1 to m do
   {
      count := count +1 ;  // For "for i"
      for j:=1 to n do
      {
         count := count+1 ;  // For "for j"
         c[i,j] = a[i,j] + b[i,j];
         count := count +1 ; //for assignment
      }
      count := count +1 ; // last time of "for j"
   }
   count := count +1 ; // last time of "for i"
}
```

Simplified algorithm with counting only

⇒ ①

```
1. Algorithm Add(a, b, c, m, n)
2 {
3     for i := 1 to m do
4     {
5         count := count + 2 ;(m)    = 2m
6         for j := 1 to n do
7             count := count + 2 ;(mn) = 2mn
8     }
9     count := count + 1 ;  1  = 1
10. }
```

2mn + 2m + 1

⇒ Line 7 is executed 'n' times for each value of i
or a total of mn times.

- Line 5 is executed m times
- Line 9 is executed 1 once.

∴ Step count = 2mn + 2m + 1

②

Statement	s/e	frequency	Total Steps
Algorithm Add(a, b, c, m, n)	0	—	0
{	0	—	0
for i := 1 to m do	1	m + 1	m+1
for j := 1 to n do	1	m(n+1)	mn+m
c[i,j] := a[i,j] + b[i,j];	1	mn	mn
}	0	—	0
Total			2mn+2m+1

Fibonacci - takes any non-negative integer n as input and prints fn - where fn is the nth fibonacci number.

```
Algorithm Fibonacci (n)
{
  if (n ≤ 1) then
     count := count + 1 ; // for if condition
  write (n);
     count := count + 1 ; // for writing
  else
  {
     fnm2 := 0 ;
     count := count + 1 ; // for assignment
     fnm1 := 1 ;
     count := count + 1 ; // for assignment
     for i := 2 to n do
     {
        count := count + 1 ; // for "for i"
        fn := fnm1 + fnm2 ;
        count := count + 1 ; // for assignment
        fnm2 := fnm1 ;
        count := count + 1 ; // for assignment
        fnm1 := fn ;
        count := count + 1 ; // for assignment
     }
     count := count + 1 ; // for last of "for"
     write (fn) ;
     count := count + 1 ; // for writing
  }
}
```

$T(n-3) + 3T(n-4) + 3T(n-5) + T(n-6) \rightarrow \theta$

$t(Fib) = \begin{cases} 2 & \text{if } n = 0 \text{ or } 1 \\ 4n+4 & \text{if } n > 1 \end{cases}$

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

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

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

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

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

$T(n-3) + T(n-4) + T(n-5)$

$+ 2[T(n-4) + T(n-5) + T(n-6)]$

n>1 $\quad 1 + 1 + 1 + 0 + (n-1)4 + 1 + 1$

$= 3 + 4n - 4 + 1 + 1$

$= 5 + 4n - 4$

$= 4n + 1$

(2)

Statement	S/e	Frequency		Total Steps	
		n=0 or 1	n>1	n=0 or 1	n>1
if (n ≤ 1) then	1	1	1	1	1
write (n);	1	1	0	1	0
else					
{					
fnm2 := 0; fnm1 := 1;	2	0	1	0	2
for i := 2 to n do	1	0	n	0	n
{					
fn := fnm1 + fnm2;	1	0	n-1	0	n-1
fnm2 := fnm1; fnm1 := fn;	2	0	n-1	0	2n-2
}					
write (fn);	1	0	1	0	1
}					
Total				2	4n+1

Summary of time complexity :-

- The time complexity of a algorithm is given by the no. of steps taken by the algorithm to compute the function it was written for.
- The no. of steps is itself a function of the instance characteristics (eg the no. of inputs, the no. of outputs, the magnitude of inputs and outputs). The characteristics are chosen based on their importance to us.
- Situations when the chosen parameters are not adequate to determine the step count (Eg : searching for an element ~~the steps count~~ may terminate in 1 step, 2 steps or n steps).

  We define three kinds of step counts :
  - best case : minimum no. of steps that can be executed for the given parameters
  - average case : average no. of steps executed on instances with the given parameters.
  - worst case : maximum no. of steps that can be executed.
- Our motivation to determine step counts is to be able to compare the time complexities of two algorithms that compute the same function and also to predict the growth in runtime as the instance characteristics change.
- Determining the exact step count is not very crucial. Since the notion of a step is itself inexact.

  Eg :

  both count as one step.

  Therefore minor differences in step count are not significant. Eg : $2n+3$, $2n+2$, $3n+4$ etc

  An exception is when difference between step counts is very large Eg : $3n+3$ vs $100n+10$

- Two algorithms with complexities $c_1 n^2 + c_2 n$ and $c_3 n$; $c_3 n$ will be faster than $c_1 n^2 + c_2 n$ for sufficiently large value of n. For small values of n, either could be faster.

Eg: $c_1 = 1$, $c_2 = 2$ and $c_3 = 100$

$c_1 n^2 + c_2 n \leq c_3 n$ for $n \leq 98$ and

$c_1 n^2 + c_2 n > c_3 n$ for $n > 98$

$c_1 = 1$, $c_2 = 2$, $c_3 = 1000$

$c_1 n^2 + c_2 n \leq c_3 n$ for $n \leq 998$ and

$c_1 n^2 + c_2 n > c_3 n$ for $n > 998$

No matter what the values of c_1, c_2 & c_3 are, there will be an 'n' beyond which complexity $c_3 n < c_1 n^2 + c_2 n$. This value of n is called "breakeven" point.

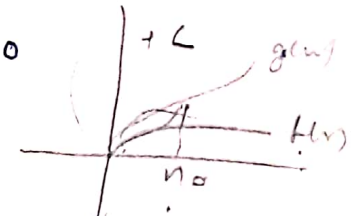
Asymptotic Notation (O, Ω, Θ)

The function f and g are non-negative functions

1) Big Oh (g(n) is upper bound on value of f(n))

The function $f(n) = O(g(n))$ if and only if there exist positive constants c and n_0 such that

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



Examples :- The functions

→ $3n+2 = O(n)$ as $3n+2 \leq 4n$ for all $n \geq 2$

→ $3n+3 = O(n)$ as $3n+3 \leq 4n$ for all $n \geq 3$

→ $100n+6 = O(n)$ as $100n+6 \leq 101n$ for all $n \geq 6$

→ $10n^2+4n+2 = O(n^2)$ as $10n^2+4n+2 \leq 11n^2$ for all $n \geq 5$

→ $1000n^2+100n-6 = O(n^2)$ as $1000n^2+100n-6 \leq 1001n^2$ $\forall$ $n \geq 100$

→ $6*2^n+n^2 = O(2^n)$ as $6*2^n+n^2 \leq 7*2^n$ for $n \geq 4$

→ $3n+3 = O(n^2)$ as $3n+3 \leq 3n^2 \; \forall \; n \geq 2$

→ $10n^2+4n+2 = O(n^4)$ as $10n^2+4n+2 \leq 10n^4, n \geq 2$

$3n+2 \neq O(1)$ as $3n+2$ is not $\leq c$ for any constant c and all $n \geq n_0$

$10n^2+4n+2 \neq O(n)$

We write :

$O(1)$ - when computing time is a constant

$O(n)$ - is called linear

$O(n^2)$ - quadratic

$O(n^3)$ - cubic

$O(2^n)$ - exponential.

$O(\log n)$ - when we need not look at all of the inputs, when we discard large chunks of unexamined input with each step.

Example : With each step of a binary search, you rule out an entire half of the space you're searching, without having to examine what is in it.

Since you can only split a space of n elements in half $\log_2 n$ times before you're looking at one item, ∴ we can get the desired element after $\log_2 n$ steps are worst.

∴ binary search works in $O(\log(n))$ time

$\frac{n}{2^x} = 1$

$\Rightarrow 2^x = n$

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

$x \log 2 = \log n$

$x = \frac{\log n}{\log 2} = \log_2 n$

n : no. of elements

x is the no. of times you can split a space of size n in half before it is narrowed down to size 1.

∴ maximum no. of steps you need to search a space of n elements is $\log_2 n$

Since $\log_2 n$ only differs from $\log_{10} n$ by a constant factor, binary search is in $O(\log n)$

→ $O(n \log n)$ - when performing search on unsorted list
- $\log n$ partitioning operations
- each partitioning taken $O(n)$ operations.

- If an algorithm takes time $O(\log n)$ it is faster for sufficiently large n, than if it had taken $O(n)$
- Similarly, $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$.

"g(n) should be as small a function of n as one can come up with for which $f(n) = O(g(n))$".

∴ $3n + 3 = O(n)$

- A functions Big-O notation is determined by how it responds to different inputs.

2. [Omega] (g(n) is lower bound on f(n))
The function $f(n) = \Omega(g(n))$ iff and only if there exist positive constants c and n_0 such that
$f(n) \geq c * g(n)$ for all n, $n \geq n_0$.

Examples
- $3n+2 = \Omega(n)$ as $3n+2 \geq 3n$ for $n \geq 1$
- (the inequality holds for $n \geq 0$, but definition of Ω requires an $n_0 > 0$)
- $3n+3 = \Omega(n)$ as $3n+3 \geq 3n$ for $n \geq 1$
- $100n+6 = \Omega(n)$ as $100n+6 \geq 100n$ for $n \geq 1$
- $10n^2+4n+2 = \Omega(n^2)$ as $10n^2+4n+2 \geq n^2$ for $n \geq 1$
- $6*2^n+n^2 = \Omega(2^n)$ as $6*2^n+n^2 \geq 2^n$ for $n \geq 1$

Also,

- $3n+3 = \Omega(1)$
- $10n^2+4n+2 = \Omega(n)$
- $10n^2+4n+2 = \Omega(1)$
- $6*2^n+n^2 = \Omega(n)$
- $6*2^n+n^2 = \Omega(1)$
- $6*2^n+n^2 = \Omega(n^2)$

There are several functions $g(n)$ for which $f(n) = \Omega(g(n))$

"$g(n)$ should be as large a function of n as possible for which the statement $f(n) = \Omega(g(n))$ is true."

$\therefore 3n+3 = \Omega(n)$, $6*2^n+n^2 = \Omega(2^n)$.

3) [Theta]

The function $f(n) = \Theta(g(n))$ if and only if there exists positive constants $c1$, $c2$ and n, such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$ for all $n \geq n_0$.

Examples:-

- $3n+2 = \Theta(n)$ as $3n+2 \geq 3n$ for all $n \geq 2$
  $3n+2 \leq 4n$ for all $n \geq 2$
  $\therefore c_1 = 3$, $c_2 = 4$, $n_0 = 2$
- $10n^2+4n+2 = \Theta(n^2)$
- $6*2^n+n^2 = \Theta(2^n)$

$3n+2 \neq \Theta(1)$, $10n^2+4n+2 \neq \Theta(n)$, $6*2^n+n^2 \neq \Theta(n^2)$

The theta notation is more precise than both the big Oh and omega notations.

The function $f(n) = \Theta(g(n))$ if and only if $g(n)$ is both ~~an~~ an upper and lower bound on $f(n)$.

Theorem (for big oh):

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$

Proof: $f(n) \le \sum_{i=0}^{m} |a_i| n^i$

$$\le n^m \sum_{i=0}^{m} |a_i| n^{i-m}$$

$$\le n^m \sum_{i=0}^{m} |a_i| \quad \text{for } n \ge 1$$

so, $f(n) = O(n^m)$ assuming that m is fixed.

→ we need not first determine the step count to determine the asymptotic complexity of the algorithm, we can first determine the asymptotic complexity of each statement in the algorithm and then add the complexities.

Each step takes only $O(1)$ time to execute.

1)

Statement	S/e	Frequency	total steps
1. Algorithm Sum(a,n)	0	-	$\theta(0)$
2. {	0	-	$\theta(0)$
3. S:=0.0;	1	1	$\theta(1)$
4. for i:=1 to n do	1	n+1	$\theta(n)$
5. S:=S+a[i];	1	n	$\theta(n)$
6. return S;	1	1	$\theta(1)$
7. }	0	-	$\theta(0)$
Total			$\theta(n)$

2)

Statement	s/e	frequency	Total steps
1. Algorithm Add (a,b,c,m,n)	0	-	$\theta(0)$
2. {	0	-	$\theta(0)$
3. for i:=1 to m do	1	m+1	$\theta(m)$
4. for j:=1 to n do	1	m(n+1)	$\theta(mn)$
5. c[i,j] := a[i,j] + b[i,j];	1	mn	$\theta(mn)$
6. }	0	-	$\theta(0)$
Total			$\theta(mn)$

Practical Complexities

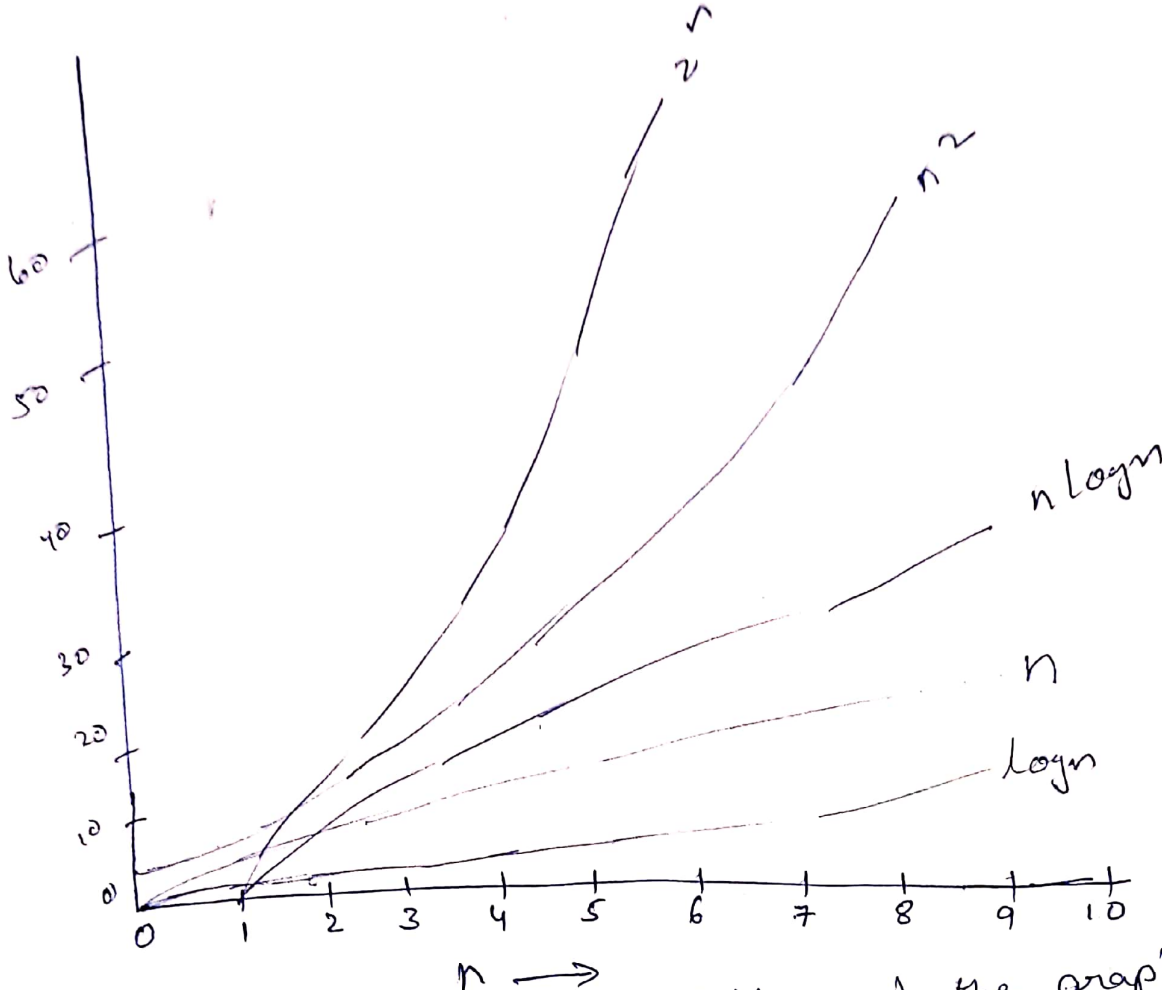
The complexity function can be used to compare two algorithms P and Q that perform the same task. Assume that algorithm P has complexity $\theta(n)$ and algorithm Q has complexity $\theta(n^2)$. we can assert that algorithm P is faster than ~~algorithm~~ algorithm Q for sufficiently large n.

- computing time of P is bounded from above by cn for some constant c, for all $n \geq n_1$
- computing time of Q is bounded from below by dn^2 for some constant d, for all $n \geq n_2$
- Since $cn \leq dn^2$ for $n \geq c/d$, algorithm P is faster than Algorithm Q whenever $n \geq \max\{n_1, n_2, c/d\}$

Table Function Values

logn	n	nlogn	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Plot of function values



It is evident from the table and the graph that the function 2^n grows very rapidly with n.

If an algorithm needs 2^n steps for execution, On a computer performing one billion steps per second then when n = 40, no. of steps = approx. $1.1 * 10^{12}$
= 18.3 minutes

n = 50, = ~~18.3~~ 13 days

n = 60, = 310.56 years

So, we may conclude that the utility of algorithms with exponential complexity is limited to small n (typically $n \leq 40$).

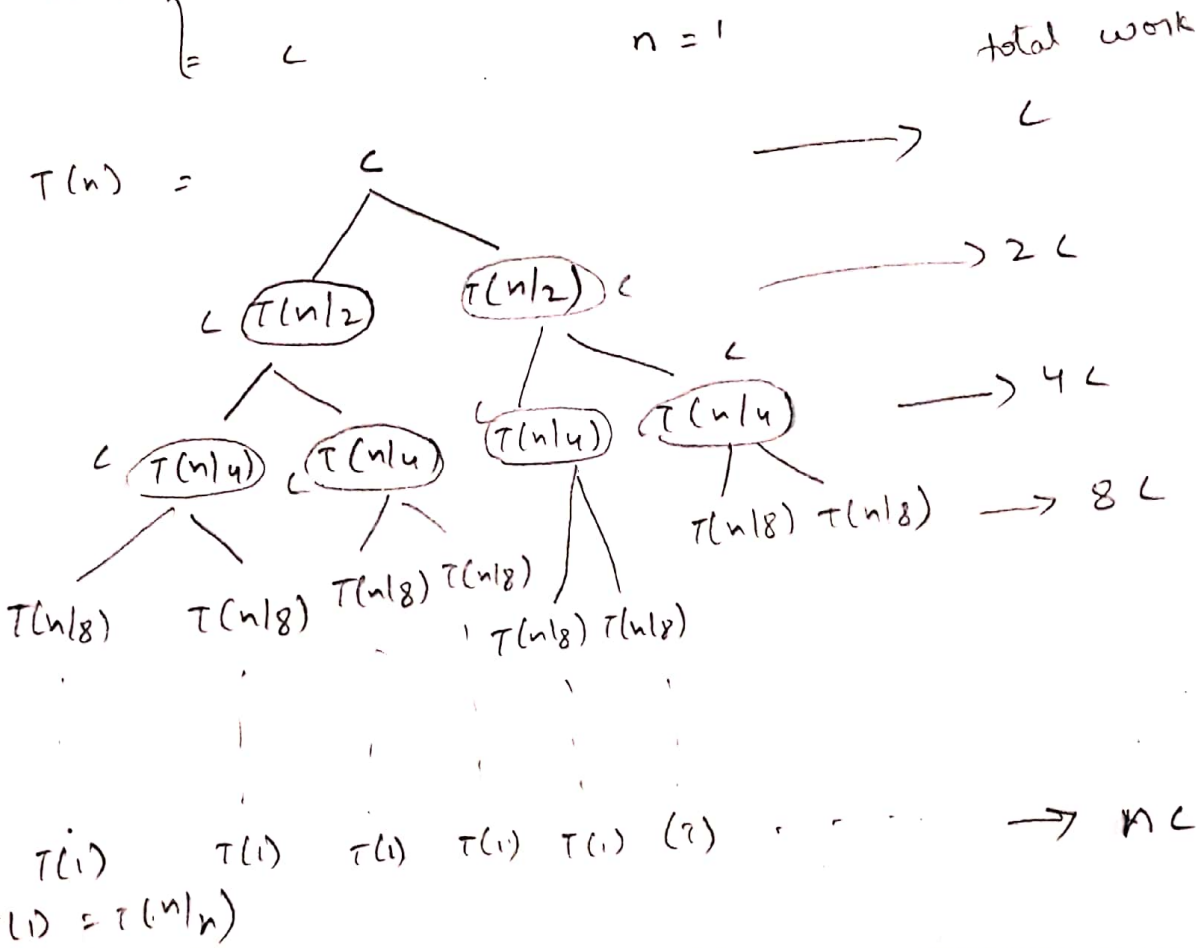
- Algorithms that have a complexity that is a polynomial of high degree (Eg n^{10}) are also of limited utility.

Recurrence tree Method

The steps involved in solving a recurrence equation using the recurrence tree method are as follows:

1. Formulate the recurrence equation by visualizing the calls as a tree.
2. Collect the following information from the recurrence tree:
   a) Level : Determine the level of the generated tree.
   b) Cost per level : Cost per level has to be calculated for every level of the generated tree using the level count and the amount of work done by the subproblems.
   c) Total cost : It is the sum of the costs of all levels.
3. Express the complexity in terms of the total cost.
4. Verify the summation using the method of substitution or some other method if necessary.

$$T(n) = \begin{cases} 2T(n/2) + c & n \geq 1 \\ c & n = 1 \end{cases}$$



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

$c + 2c + 4c + \ldots + nc$

assume $n = 2^k$

$c(1 + 2 + 4 + \ldots + 2^k)$

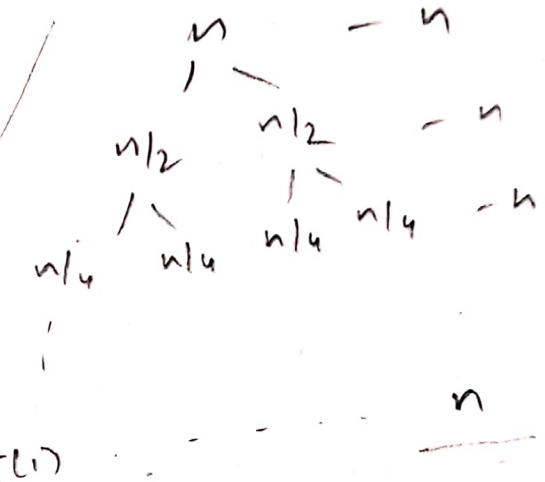
GP

$$c\left(\frac{1(2^{k+1} - 1)}{2 - 1}\right)$$

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

$$= c(2n - 1)$$

$$= O(n)$$



Geometric series

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n$$

$$= \frac{x^{n+1} - 1}{x - 1}$$

Arithmetic series

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

Harmonic series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \log n$$

$\log x^y = y \log x$

$\log xy = \log x + \log y$

$\log \log n = \log(\log n)$

$a^{\log_b x} = x^{\log_b a}$

level

$n/2^0$ $n/2^1$ $n/2^2$ ... $n/2^k$

total level $(k+1)$ level

$n = 2^k$

$k = \log n$

$(\log n + 1)$

at every level total work done is n

$\therefore n(\log n + 1)$

$= n \log n + n$

$O(n \log n)$.

Masters Theorem

$\log^2 n = \log \log n$

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

~~$T(n) = 2T(\frac{n}{2}) + O(n)$~~

$T(n) = aT(\frac{n}{b}) + \Theta(n^k \log^p n)$

$a \geq 1$, $b > 1$, $k \geq 0$, p is a real number. then:

1) If $a > b^k$ then

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

2) if $a = b^k$ then

a) If $p > -1$, then

$T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

b) If $p = -1$, then

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

c) If $p < -1$, then

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

3) If $a < b^k$

a) If $p \geq 0$ then $T(n) = \Theta(n^k \log^p n)$

b) If $p < 0$, then $T(n) = O(n^k)$.

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

$a = 3, \quad b = 2, \quad k = 2, \quad p = 0$

$a = 3, \quad b^k = 2^2 = 4$

$\Rightarrow a < b^k$ and $p = 0$

The 3.a. $\Rightarrow$ $T(n) = \Theta(n^k \log^p n)$

$= \Theta(n^2 \log^0 n) = \Theta(n^2)$

Q) $T(n) = 3T(n/2) + n^2$ $\Theta(n^2)$ 3.a

Q) $T(n) = 4T(n/2) + n^2$ $\Theta(n^2 \log n)$ 2.a

Q) $T(n) = 7T(n/2) + n^2$ $\Theta(n^2)$ 3.a

Q) $T(n) = 2^n T(n/2) + n^n$ Does not apply. a is not constant

Q) $T(n) = 16T(n/4) + n$ $\Theta(n^2)$ case1

Q) $T(n) = 2T(n/2) + n\log n$ $\Theta(n\log^2 n)$ 2.a

Q) $T(n) = 2T(n/2) + \frac{n}{\log n}$ $\Theta(n \log\log n)$ 2.b

Q) $T(n) = 2T(n/4) + n^{0.51}$ $\Theta(n^{0.51})$ 3.b

Q) $T(n) = 0.5T(n/2) + 1/n$ does not apply; a<1

Q) $T(n) = 4T(n/2) + \log n$ $\Theta(n^2)$ Case 1

Performance Measurement :-

- We do not concern ourselves with the space and time needed for compilation. We justify this assumption that each program (after being fully debugged) is compiled once and then executed several times.
- we do not consider measuring the runtime space requirements of a program. we focus on measuring the computing time of a program.

To obtain the computing (or run) time of a program we need a clocking procedure. we assume the existence of a program GetTime() that returns the current time in milliseconds.

```
1. Algorithm for SeqSearch (a, x, n)
2. // Search for x in a[1:n], a[0] is used as additional space
3. {
4.     i := n ; a[0] := x ;
5.     while (a[i] ≠ x) do i := i - 1 ;
6.     return i ;
7. }
```

To measure the worst case performance of the sequential search algorithm:

① we need to decide on the values of 'n' for which the times are to be obtained.

② Determine, for each of the above values of n, the data that exhibit the worst case behaviour.

To search for x, given the size n of a, An asymptotic analysis reveals that the worst case time is O(n). So, we expect a plot of the times to be a straight line.

Theoretically, if we know the times for any two values of n the straight line is determined, and we can obtain the time for all other values of n from this line.

In practice, we need the times for more than two values of n. This is so for the following reasons:

1) Asymptotic analysis tells us the behavior only for sufficiently large values of n. For smaller values of n, the run time may not follow the asymptotic curve. To determine the point beyond which the asymptotic curve is followed, we need to examine the times for several values of n.

2) Even in the region where asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve because of the effects of low-order terms that are discarded in the asymptotic analysis.

For the sequential search algorithm:

- for $n > 100$, we obtain runtime for just a few values

  Eg $n = 200, 300, 400, \ldots, 1000$

- for n in the range of $[0, 100]$ we carray out a more refined measurement, to check where the asymptotic behavior begins.

∴ Times in this range are obtained in steps of 10 beginning at $n = 0, 0, 10, 20, 30, 40, \ldots 100$

Algorithm to obtain worst-case times:

```
1. Algorithm TimeSearch()
2. {
3.     for j:=1 to 1000 do a[j]:=j;
4.     for j:=1 to 10 do
5.     {
6.         n[j]:=10*(j-1); n[j+10]:=100*j;
7.     }
```

```
8.    for j:=1 to 20 do
9.    {
10.       h:=GetTime();
11.       k:=SeqSearch(a,0,n[j]);
12.       h1:=GetTime();
13.       t:=h1-h;
14.       write(n[j],t);
15.   }
16. }
```

Steps 4 to 6 initialize the values of n[0] to n[20] as {0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000}

Result of steps 8 to 15 would be:

n	time	n	time
0	0	100	0
10	0	200	0
20	0	300	1
30	0	400	0
40	0	500	1
50	0	600	0
60	0	700	0
70	0	800	1
80	0	900	0
90	0	1000	0

- The times obtained are too small to be of any use.
- The non-zero times are just noise and are not representative of the time taken.
- Most of the times are zero, this indicates that the precision of our clock is inadequate.

∴ we change the algorithm as

```
Algorithm TimeSearch()
{ // Repetition Factors
  n[21] := { 0, 200000, 200000, 150000, 100000, ... 25000};
  for j:=1 to 1000 do a[j]:=j;
  for j:=1 to 10 do
  {
      n[j] := 10 * (j-1) ; n[j+10] := 100 * j ;
  }
  for j:=1 to 20 do
  {
      h := GetTime();
      for i:=1 to n[j] do k := SeqSearch(a,0,n[j]);
      h1 := GetTime();
      t1 := h1 - h;
      t := t1 ; t := t/n[j];
      write(n[j], t1, t);
  }
}
```

In the above algorithm, n[i] is the no. of times the search is to be repeated when the no. of elements in the array is n[i].

An alternate timing construct is:

```
h := GetTime() ; t := 0;
while (t < DESIRED_TIME) do
{
    k := SeqSearch(a,0,n[j]);
    h1 := GetTime();
    t := h1 - h;
}
```

Result of the above algorithm for values 0 to 20:

n	t1	t	n	t1	t
0	308	0.002	100	1683	0.034
10	923	0.005	200	3359	0.067
20	1181	0.008	300	4693	0.094
30	1087	0.011	400	6323	0.126
40	1384	0.014	500	7799	0.156
50	1691	0.017	600	9310	0.186
60	999	0.020	700	5419	0.217
70	1156	0.023	800	6201	0.248
80	1306	0.026	900	6994	0.280
90	1460	0.029	1000	7723	0.509

Work-case times for sequential search

→ Generating a data set in the worst case performance of an algorithm is not easy. We generate a suitably large no. of random ~~test~~ test data, run time for each of these data are obtained. The maximum of these times is used as an estimate of work-case. Average time is also estimated using random data.

Amortized Analysis

Usually when one i/p sequence doesnot affect the running time of the next set of i/p sequence. If first i/p effects the running time of the next i/p then we go for Amortized analysis.

Ex:- Problem stmt

Given an array of 'n' elements, find k^{th} smallest element.

① Sort (Quick sort (nlogn))

② return A[k]

Worst case

```
global variable Sorted
int findkthSmallest (int A[], int n, int k)
{
    if (!Sorted)
    {
        Sort (A, n)
        Sorted = 1;
    }
    return A[k];
}
```

Here Sort operation is performed only for the first time and then for the remaining ~~post~~ Smallest elements no need to perform Sort operations.

So here the first set is effecting the other set.

So if one i/p is effecting the running time of the other i/p's then we go for amortized analysis.

So for the first one we are spending nlogn and remaining only 1

$\Rightarrow$ $n\log n + (1 + 1 + \dots (n-1))$ (for n times over all running time)

$= n\log n + n$

if we take avg.

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

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

$\Rightarrow O(\log n)$ is over all running time.

Amortized cost:- Given a sequence of n operations, the amortized cost is $\frac{\text{cost (n operations)}}{n}$

Amortized analysis refers to finding the avg. running time per operation over a worst case sequence of operations. Amortized analysis differs from average-case performance in that probability is not involved. It guarantees the time per operation over worst case performance.

There are several techniques used in amortized analysis:

→ Aggregate analysis : determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the average cost to be $T(n)/n$.

→ Accounting method - determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations.

→ Potential method - It is like the accounting method, but overcharges operations early to compensate for undercharges later.

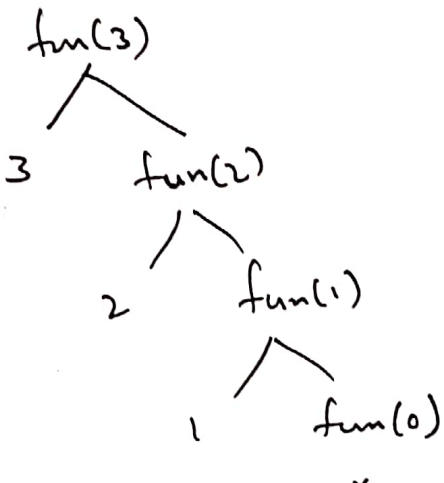
Recursion


```
type fun(parameters)
{
    if(base-condition)
    {
        1. ...
        2. fun(parameters)
        3.
    }
}
```

ex:

```
void fun(int n)
{
    if(n>0)
    {
        printf("%d", n);
        fun(n-1);
    }
}
void main()
{
    int x=3;
    fun(3);
}
```

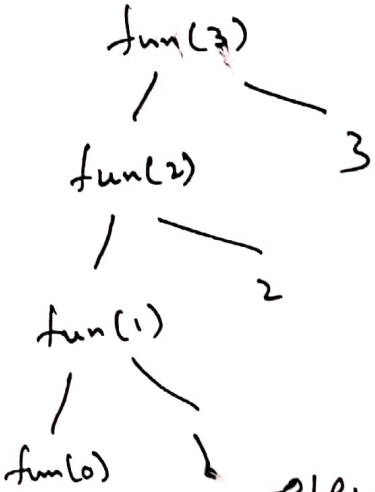
Traced in the form of a tree.



o/p :- 3, 2, 1

eg:-

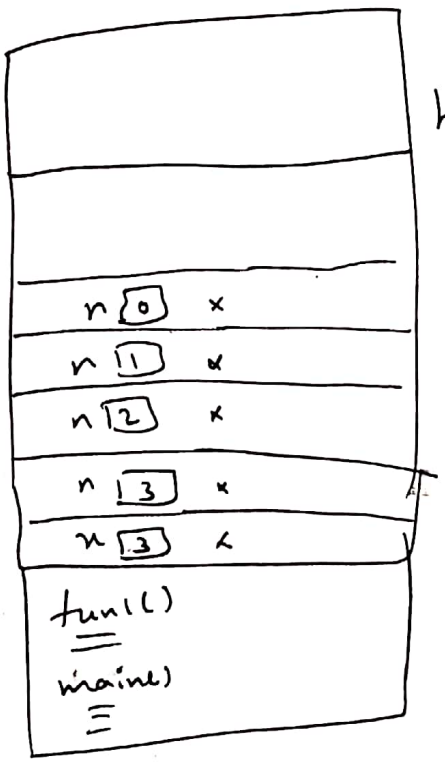
```
void fun1(int n)
{
    if(n>0)
    {
        fun1(n-1);
        printf("%d", n);
    }
}
void main()
{ int x=3; fun1(3); }
```



o/p:- 1 2 3

```
void fun(int n)
{
    if(n>0)
    {
Ascending — 1. executes at calling time
        2. fun(n-1);
Descending — 3. executes at returning time
    }
}
```

How recursion uses stack

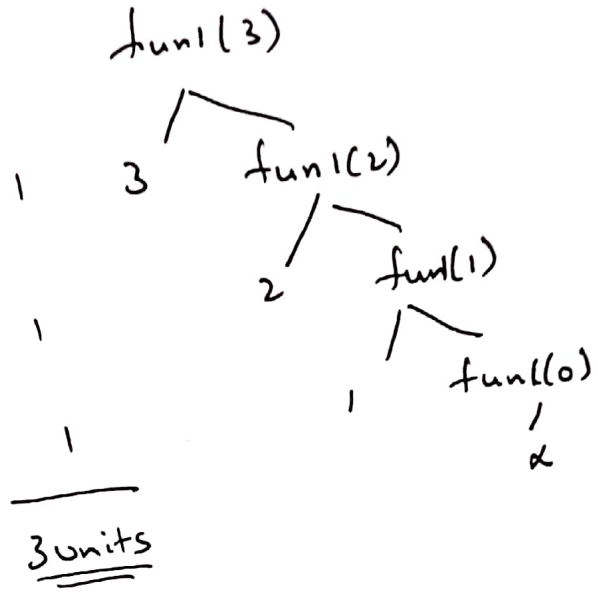


Space = n+1
O(n)

fun(fun(n))
A B

Recurrence relation time complexity

```
void fun1(int n)
{
  if (n>0)
  { printf("%d", n);
    fun1(n-1);
  }
}
void main()
{ int n=3;
  fun1(n);
}
```



1
3 units

if n = n
then n units of times
O(n)

Recurrence relation

```
void fun1(int n)          ——— T(n)
{
  if(n>0)                 ——— 1
  { printf("%d", n);      ——— 1
    fun1(n-1);            ——— T(n-1)
  }
}
```

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

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

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

n - k = 0
n = k
T(0) + n
1 + n
O(n)

Types of Recursions

1. Tail
2. Head
3. Tree
4. Indirect
5. Nested

(1) Tail

```
void fun(int n)
{
    if(n>0)
    {
        printf("%d", n);
        fun(n-1);
    }
}

fun(3);
```

Time - O(n)

Space - O(n)

```
void fun(int n)
{
    while(n>0)
    {
        printf("%d", n);
        n--;
    }
}
```

Time - O(n)

Space - O(1)

(2) Head

```
void fun(int n)
{
    if(n>0)
    {
        fun(n-1);
        printf("%d", n);
    }
}
fun(3)
```

```
Void fun(int n)
{
    int i=1;
    while(i<=n)
    {
        printf("%d", i);
        i++;
    }
}
```

Linear

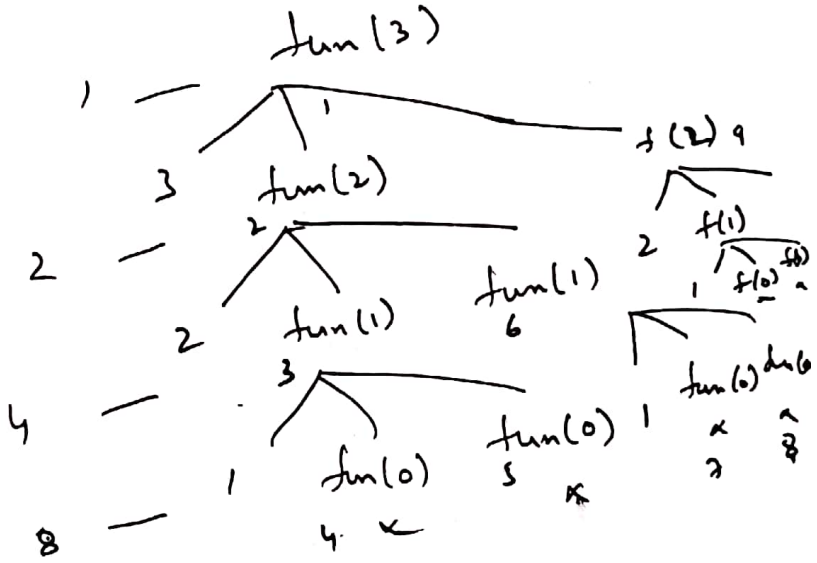
```
fun(int n)
{
    if(n>0)
    {
        ==
        fun(n-1);
        ==
    }
}
```

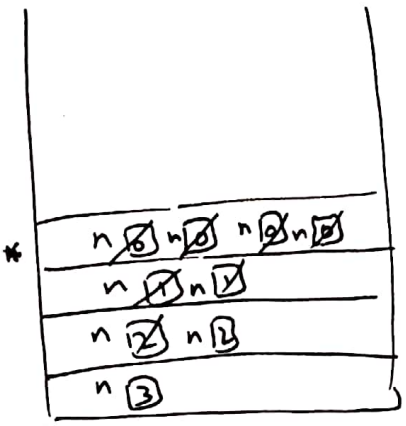
Tree

```
fun(int n)
{
    if(n>0)
    {
        ==
        fun(n-1);
        ==
        fun(n-1);
        ==
    }
}
```

```
void fun(int n)
{
    if(n>0)
    {
1.      printf("%d", n);
2.      fun(n-1);
3.      fun(n-1);
    }
}

fun(3);
```





$1+2+4+8 = 15$

$2^{3+1}-1$

$2^{n+1}-1$

$O(2^n)$

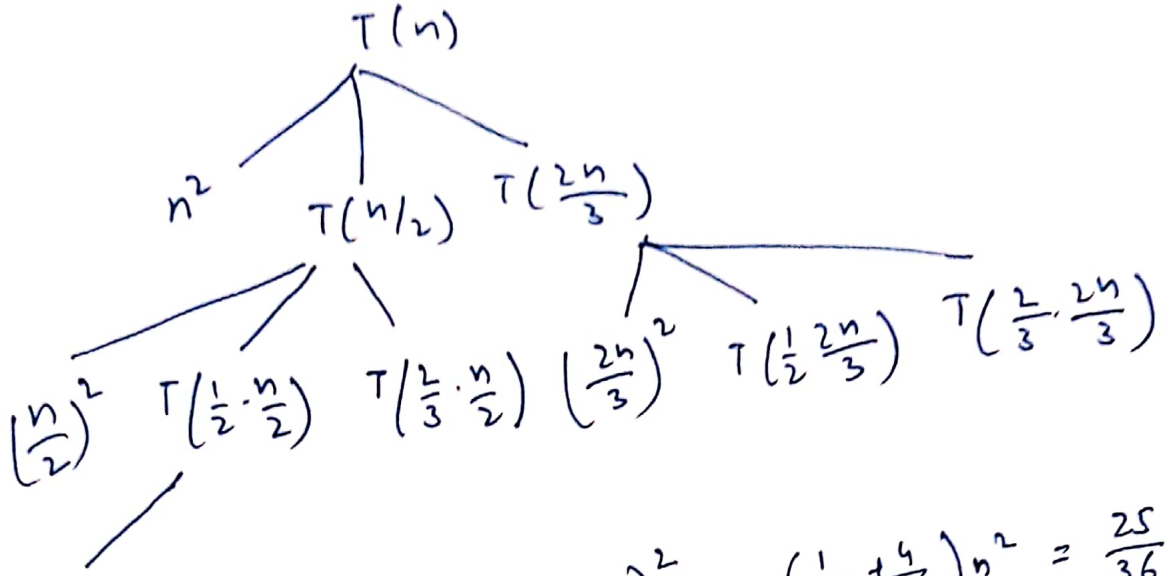
Indirect

```
Void funA(int n)
{
    if(n>0)
    {
        printf("%d", n);
        funB(n-1);
    }
}
void funB(int n)
{
    if(n>1)
    {
        printf("%d", n);
        funA(n/2);
    }
}
```

Nested Recursion

```
int fun(int n)
{
    if(n>100)
        return n-10;
    else
        return fun(fun(n+11));
}
    fun(95);
```

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{2n}{3}\right) + n^2$$



level 1 : $\left(\frac{1}{2}n\right)^2 + \left(\frac{2}{3}n\right)^2 = \left(\frac{1}{4} + \frac{4}{9}\right)n^2 = \frac{25}{36}n^2$

level 2 : $\left(\frac{1}{4}n\right)^2 + \left(\frac{1}{3}n\right)^2 + \frac{1}{3}n^2 + \frac{4}{9}n^2 = \frac{625}{1296}n^2$

$= \left(\frac{25}{36}\right)^2 n^2$

level k : $\left(\frac{25}{36}\right)^k n^2$

let $\alpha = \frac{25}{36}$

$$T(n) = \sum_{k=0}^{\infty} \alpha^k n^2$$

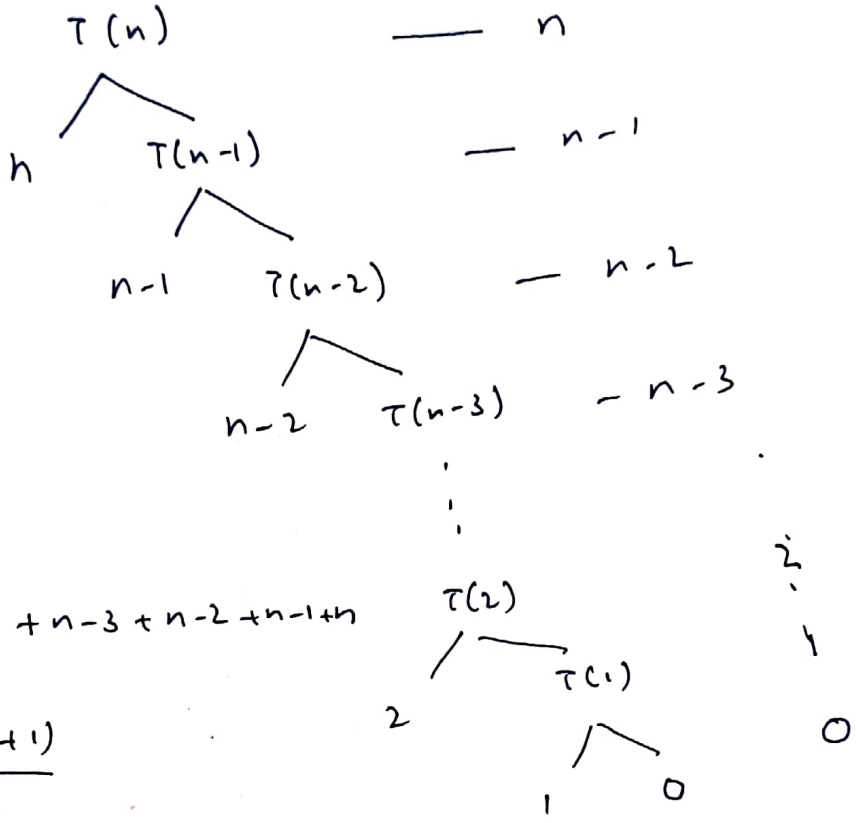
$$= \frac{1}{1-\alpha} n^2$$

$$= \frac{1}{1 - \frac{25}{36}} n^2$$

$$= \frac{1}{\frac{11}{36}} n^2 = \frac{36}{11} n^2 = O(n^2)$$

=

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



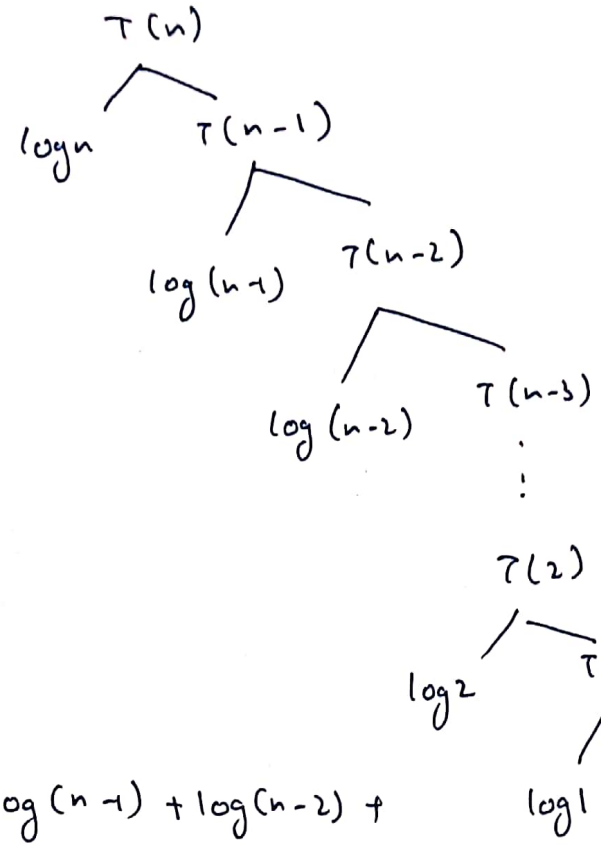
$0 + 1 + 2 + \cdots + n-3 + n-2 + n-1 + n$

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

$= O(n^2)$

```
void fun(int n)
{
    if (n > 0)
    {
        for (i=0; i<n; i++)
        {
            printf("%d", n);
        }
        fun(n-1);
    }
}
```

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



$\log n + \log(n-1) + \log(n-2) + \ldots + \log 2 + \log 1$

$= \log(n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 2 \times 1)$

$= \log(n!)$

$= O(n \log n)$

```
void fun(int n)
{  if (n > 0)
   {  for (i=1; i<n; i=i*2)
         printf("%d", i)
   }
   fun(n-1);
}
```

SM

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

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

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

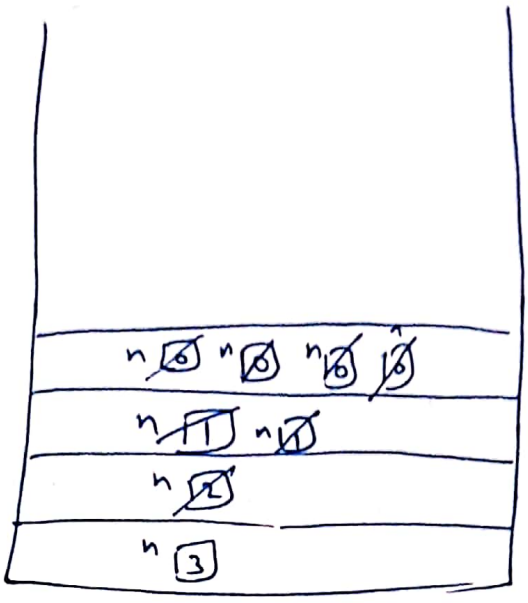
$= [T(n-3) + \log(n-2)] + \log(n-1) + \log n$

$n - k = 0$

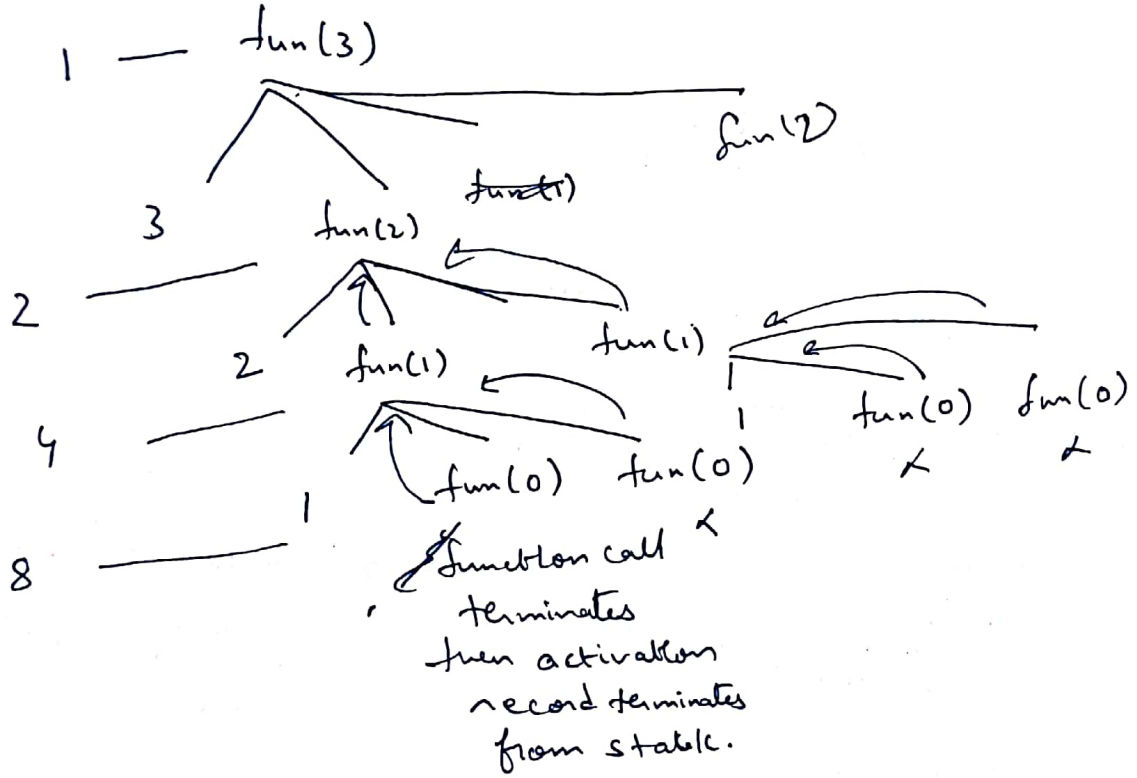
$n = k$

$T(n) = T(0) + \log n! = 1 + \log n! = O(n \log n)$

```
void fun(int n)
{  if (n>0)
   {
1    printf("%d", n);
2    fun(n-1);
3    fun(n-1);
   }
}
fun(3);
```



o/p: 3, 2, 1, 1



T.C

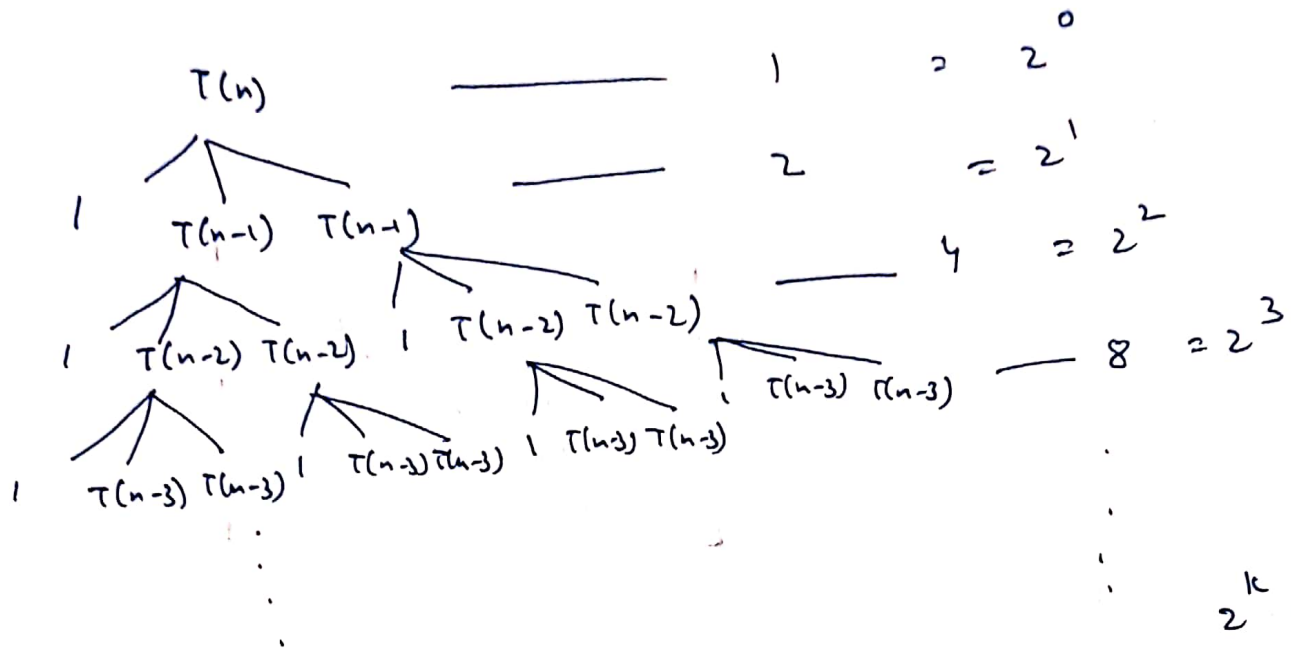
$1+2+4+8 = 15$

$2^0+2^1+2^2+2^3 = 2^{3+1}-1$

$2^0+2^1+2^2+\cdots+2^n = 2^{n+1}-1 = O(2^n)$

S.C $O(n)$ [Maximum stack size].

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



$T(0)$

$$2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^k = 2^{k+1} - 1$$

G.P series

$$a + ar + ar^2 + ar^3 + \cdots + ar^k = \frac{a(r^{k+1} - 1)}{r - 1}$$

$$a = 1,\ r = 2 \Rightarrow \frac{1(2^{k+1} - 1)}{2 - 1} = 2^{k+1} - 1$$

Assume $n - k = 0$

$\therefore n = k$

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

$=$

SM

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

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

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

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

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

$$\vdots$$

$$2^k T(n-k) + 2^{k-1} + 2 \cdots + 2^2 + 2 + 1 \rightarrow (4)$$

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

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

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

$$\Rightarrow 2^{n+1} - 1$$

$$\Rightarrow O(2^n)$$

$=$

Guidelines for Asymptotic Analysis

There are some general rules to help us determine the running time of an algorithm.

1) **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the no. of iterations.

```
for (i=1 ; i<=n ; i++)      // executes n times
    m = m+2 ;               // constant time, c
```

Total time = $c \times n = cn = O(n)$

2) **Nested loops**: Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
for (i=1 ; i<=n ; i++)          // executes n times
{
    for (j=1 ; j<=n ; j++)      // executes n times
        k = k+1 ;               // constant time, c
}
```

Total time = $c \times n \times n = cn^2 = O(n^2)$

3) **Consecutive statements**: Add the time complexities of each statement.

```
x = x+1 ;                       // constant time c0
for (i=1 ; i<=n ; i++)          // executes n times
    m = m+2 ;                   // constant time c1
for (i=1 ; i<=n ; i++)          // executes n times
{
    for (j=1 ; j<=n ; j++)      // executes n times
        k = k+1 ;               // constant time c2
}
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$

4) If - then - else Statements:

Worst-case running time: the test, plus either the then part or the else part (whichever is the longer).

```
if (length() == 0)          // test: constant c0
{
   return false;            // then part constant c1
}
else
{ // else part:
  //(constant + constant) * n
    for(int n=0; n < length(); n++)   // n
    {
       if( !list[n].equals(otherList.list[n])) // constant c2
          return false;     // constant c3
    }
}
```

Total time: $c_0 + c_1 + (c_2 + c_3)n = O(n)$

5) Logarithmic complexity: An algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction (usually by 1/2). As an example let us consider the following program:

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

Here the value of i is doubling every time.

Initially i=1, 2, then 4, 8 so on.

Assume the loop is executing some k times.

At kth step $2^k = n$, at (k+1)th steps we come out of the loop.

Taking alg. both sides $\log(2^k) = \log n$

$k\log 2 = \log n$ $\quad k = \log_2 n \Rightarrow O(\log n)$

Q) Find the complexity of the function given below.

```
void function(int n)
{
    int i, count = 0;
    for(i=1; i*i<=n; i++)
        count++;
}
```

Here the loop will end if $i^2 > n$

$\Rightarrow T(n) = \sqrt{n}$

Q) What is the running time of the following function?

```
void Function(int n)
{ int i=1, s=1;
  while(s<=n)
  {
    i++;
    s=s+i;
    printf("*");
  }
}
```

We can define the 's' terms according to the relation $S_i = S_{i-1} + i$.

The value of 'i' increases by 1 for each iteration.

The value contained in 's' at the i^{th} iteration is the sum of the first 'i' positive integers.

If k is the total no. of operations taken by the program, then the while loop terminates if

$$1 + 2 + \cdots + k$$

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

$$\Rightarrow k = \sqrt{n}$$

Q) What is the complexity of the program given below:

```
void function (int n)
{   int i, j, k, count = 0;
n/2 — for (i = n/2 ; i <= n ; i++)
log n — for (j = 1 ; j <= n ; j = 2 * j)
log n — for (k = 1 ; k <= n ; k = k * 2)
            count ++ ;
```

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

Q) Find the complexity of the program below:

```
function (int n)
{   if (n == 1) return ;
n = for (int i = 1 ; i <= n ; i++)
    {
1 — for (int j = 1 ; j <= n ; j++)
    {   printf ("a");
        break ;
    }
}
}
```

$O(n)$

a) Write a recursive function for the running time T(n) of the function given below. Prove using the iterative method that $T(n) = O(n^3)$.

```
function (int n)
{
  if (n == 1) return;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      printf ("*");
  function (n-3);
}
```

→ $T(n) = T(n-3) + cn^2$ for $c > 0$

Masters → $\Theta(n^3)$

Q) Determine Θ bounds for the RR:

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

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

Q) P.7. the running time of the code below is $\Omega(\log n)$.

```
void recd (int n)
{
  int k = 1;
  while (k < n)
    k = 3 * k;
}
```

the while loop will terminate once the value of 'k' is > or = to the value of 'n'. In each iteration the value of k is multiplied by 3.

If i is the no. of iterations, the 'k' has the value of 3^i after i iterations.

The loop is terminated upon reaching 'n' iterations when $3^i \geq n$

$\Rightarrow i \geq \log_3 n$,

$\Rightarrow \Omega(\log n)$.

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

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

$\vdots$

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

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

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

$$= O(n^3)$$

Q) function(n)

n — for(int i=1; i<=n; i++)

(n/i) times for each value of i — for(int j=1; j<=n; j+=i)

printf("*");

Running Time $= n \times \sum_{i=1}^{n} n/i = O(n \log n)$.

Q) Complexity of $\sum_{i=1}^{n} \log i$?

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

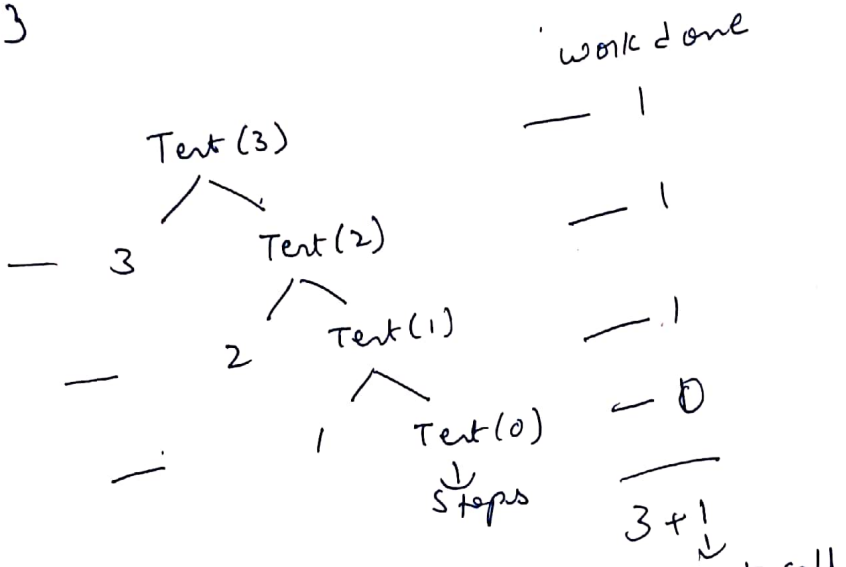
$= \log(1 \times 2 \times \cdots \times n)$

$= \log(n!) \leq \log(n^n) \leq n \log n$

$O(n \log n)$.

Recurrence Relation

①
```
Void Test(int n)
{
    if (n > 0)
    {
        printf("%d", n);   → 1
        Test(n-1);
    }
}
```



if Test(n) = n+1

⇒ O(n)

```
Void Test(int n)                    → T(n)
{
    if (n > 0)
    {
        printf("%d", n);   → 1
        Test(n-1);         → T(n-1)
    }
}
```

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

⇒ $T(n) = T(n-1) + 1$

$T(n) = T(n-1) + 1$ → ①

$T(n-1) = T(n-2) + 1$ → ②

Substitute ② in ①

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

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

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

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

⋮ continues for k times

$T(k) = T(n-k) + k \longrightarrow (4)$

Since $T(0) = 1$

Assume $n - k = 0$

$\therefore n = k \longrightarrow (3)$

Substitute (4) in (3)

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

$= T(0) + n$

$= 1 + n$

$\Rightarrow O(n)$

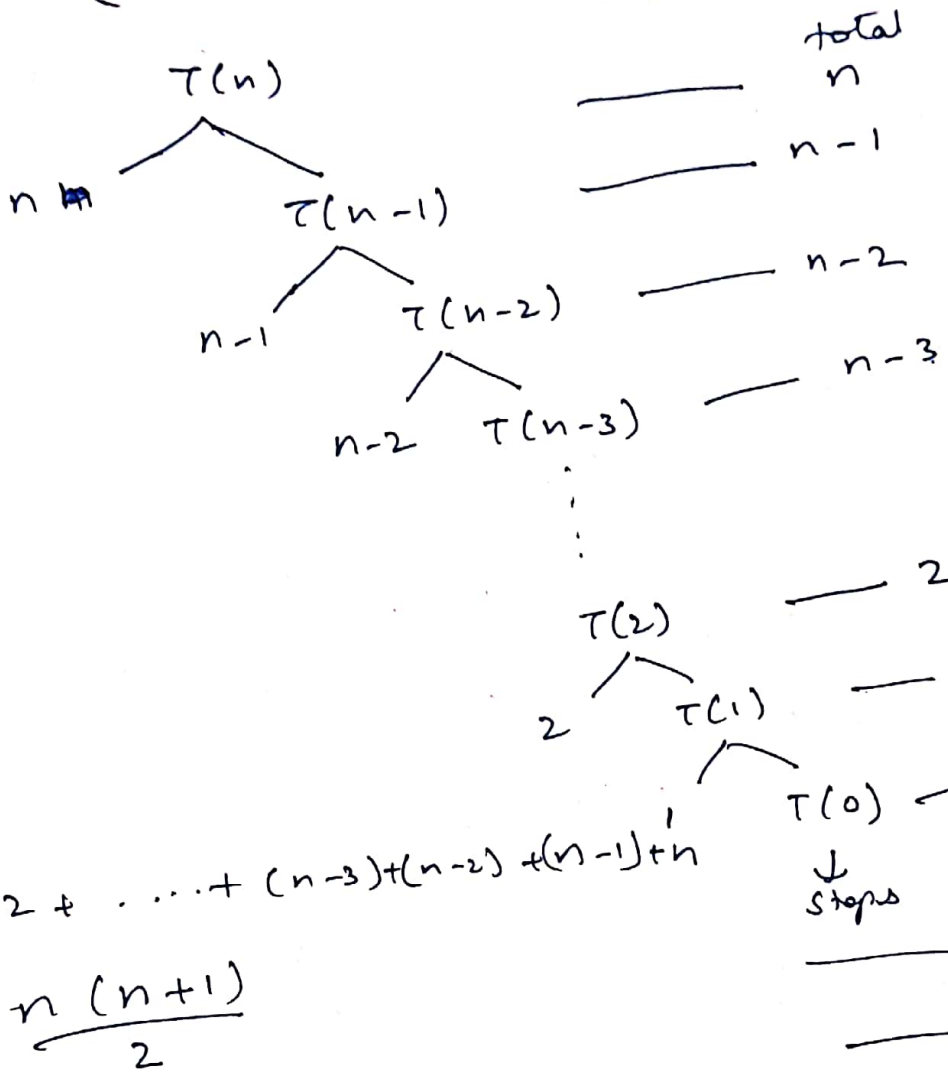
(2)

```
void Test (int n)                    → T(n)
{
  if (n > 0)                         ——— 1
  {
    for (i=0; i<n; i++)              —— n+1
    {
      printf ("%d", n);              —— n
    }
    Test (n-1);                      —— T(n-1)
  }
}
```

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

finding solution is a difficult task we consider it as n.

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



$$0 + 1 + 2 + \cdots + (n-3) + (n-2) + (n-1) + n$$

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

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

$$O(n^2)$$

Substitution method

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

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

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

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

$T(n-2) = T(n-3) + (n-2) \longrightarrow (4)$

Substitute eq. (2) in (1)

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

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

(4) in (3)

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

$T(n) = T(n-3) + (n-2) + (n-1) + n$ ⟶ (5)

continues for k times

$T(k) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \cdots + (n-1) + n$ ⟶ (6)

Assume $n - k = 0$

$n = k$ ⟶ (7)

Substitute (7) in (6)

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

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

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

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

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

$T(n) = T(n-1) + \log_2 n$

$T(1) = 0$

$T(n) = T(n-1) + \log_2 n$

$= T(n-2) + \log_2 (n-1) + \log_2 n$

$= T(n-3) + \log_2 (n-2) + \log_2 (n-1) + \log_2 n$

$\vdots$

$= T(1) + \log_2 2 + \log_2 3 + \cdots + \log_2 n$

$= \log_2 2 \times 3 \times 4 \cdots n$

$= \log_2 n!$

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

$T(0) = 2$

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

$= 3 \times \{3T(n-2) + 2\} + 2$

$= 3^2 T(n-2) + 2 \times 3 + 2$

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

$\vdots$

$= 3^n T(0) + 2\{1 + 3 + 3^2 + \cdots + 3^{n-1}\}$

$= 2\{1 + 3 + 3^2 + \cdots + 3^{n-1} + 3^n\}$

$1 + x + x^2 + \cdots + x^{n-1} + x^n = \frac{x^{n+1} - 1}{x - 1}$, when $x > 1$

$\Rightarrow 2\frac{\{3^{n+1} - 1\}}{3 - 1} = 3^{n+1} - 1$

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

write we order as degree

$O(n)$

```
for(i=n ; i>0 ; i--)
{
    Stmt;             → n
}
```

$O(n)$

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

$O(n)$

→ degree of polynomial is n so $O(n)$ even $\frac{n}{20}, \frac{n}{30}$ $= O(n)$

(2)
```
for(i=0 ; i<n ; i++)          → n+1
{
    for(j=0 ; j<n ; j++)      → n(n+1)
    {
        Stmt;                 → n x n
    }
}
```

$O(n^2)$

③
```
for (i=0; i<n; i++)
{
    for (j=0; j<i; j++)
    {
        stmt;  —— ?
    }
}
```

i	j	no. of times
0	0	0
1	0	1
	1	
2	0	2
	1	
	2	
3	0	3
	1	
	2	
	3	
⋮	⋮	⋮
n	. . .	n

$0+1+2+\cdots+n$

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

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

$= O(n^2)$

④
```
p = 0
for (i=1; p<=n; i++)
{
    p = p+i;
}
```

i	p
1	0+1=1
2	1+2=3
3	1+2+3
4	1+2+3+4
⋮	
k	1+2+3+4+..+k

Assume $p > n$

Since $p = 1+2+3+\cdots+k$

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

$\Rightarrow \frac{k(k+1)}{2} > n \Rightarrow O(\sqrt{n})$

$\Rightarrow k^2 > n$

$k = \sqrt{n}$

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

i
1
1×2 = 2
2×2 = 2^2
$2^2 \times 2 = 2^3$
⋮
2^k

Assume $i >= n$

Since $i = 2^k$

$\therefore 2^k >= n$

$2^k = n$

$k = \log_2^n$

$\therefore$ ~~the~~ the statement will execute for $O(\log_2^n)$.

Another way

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

$f = 1 \times 2 \times 2 \times 2 \times \ldots = n$

$2^k = n$

$k = \log_2^n$

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

$f = 1+1+1+1 \ldots +1 = n$

$k = n$

$\log n$ will give decimal values

So whether to take floor or ceil is a question.

To know that

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

$\lceil \log n \rceil$

$\log_2^8 = 3$

$\log_2^{10} = 3.2$ ~~but~~ but it repeats for 4.

So we take ceil value.

Suppose $n = 8$

i
1
2
4
8×

3 times it repeats.

$n = 10$

i
1
2
4
8
×16

4 times it repeats.

⑥ for(i=n ; i>=1 ; i = i/2)
{
Stmt;
}

Assume $i < 1$

$\therefore \frac{n}{2^k} < 1$

$\frac{n}{2^k} = 1$

$n = 2^k$

$k = \log_2^n$

$O(\log_2^n)$.

$\frac{n}{n}$
$\frac{n}{2}$
$\frac{n}{2^2}$
$\frac{n}{2^3}$
$\vdots$
$\frac{n}{2^k}$

⑦ for(i=0 ; i*i<n ; i++)
{
Stmt;
}

Execute until $i*i < n$

terminate when $i*i >= n$

$i^2 = n$

$i = \sqrt{n}$

⑧ for(i=0 ; i<n ; i++)
{
Stmt;
}

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

n

n

2n

O(n)

9) p=0
for(i=1; i<n; i=i*2)
{
  p++;
}
for(j=1; j<p; j=j*2)
{
  stmt;
}

—— log n
—— log p
⇒ log(log n)
⇒ O(log(log n))

(10) for(i=0; i<n; i++)
{
  for(j=1; j<n; j=j*2)
  {
    stmt;
  }
}

— n
— n * log n
— n * log n
2n log n + n
⇒ O(n log n)

for(i=0; i<n; i++) → O(n)
for(i=0; i<n; i=i+2) → n/2 → O(n) ⇒ n/200 = O(n)
for(i=n; i>1; i--) → O(n)
for(i=1; i<n; i=i*2) → O(log n)
for(i=1; i<n; i=i*3) → O(log_3 n)
for(i=n; i>1; i=i/2) → O(log n)

Analysis of if and while

①
```
i = 0                  —— 1
while (i<n)            —— n+1
{ Stmt;                —— n
  i++;                 —— n
}                        3n+2
```
O(n)

②
```
a = 1;
while (a < b)
{ Stmt;
  a = a * 2;
}
```

a
1
1×2
1×2×2 = 2^2
1×2×2×2 = 2^3
⋮
2^k (Assume it stops here)

Terminate
$a \geq b$
$\therefore a = 2^k$
$2^k \geq b$
$\Rightarrow 2^k = b$
$k = \log_2 b$
$\Rightarrow O(\log_2 b)$

Since we write in terms of n
So $b = n$
$\Rightarrow O(\log_2 n)$

③
```
i = n;
while (i > 1)
{ Stmt;      O(log n)
  i = i/2;
}
```

```
for (i = n; i > 1; i = i/2)
{ stmt;
}
```

④
```
i = 1;
k = 1;
while (k <= n)
{ Stmt;
  k = k + i;
  i++;
}
```

i	k
1	1
2	1+1 = 2
3	2+2 = 4
4	2+2+3
5	2+2+3+4
⋮	⋮
m	2+2+3+4+...+m

when $k \geq n$ it must stop then $\frac{m(m+1)}{2}$

$\Rightarrow \frac{m(m+1)}{2} \geq n$

$\Rightarrow \frac{m^2+m}{2} \geq n$

$\Rightarrow m^2 \geq n$

$\Rightarrow m = \sqrt{n}$

$= O(\sqrt{n})$

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

$\frac{m}{6}$	$\frac{n}{3}$	
3	3	→ executed 1 time
$m=5$	$n=5$	→ executes 0 times

(5) Let one of the procedure

```
while (m != n)
{ if (m > n)
    m = m - n;
  else
    n = n - m;
}
```

$m=16$	$n=2$	
14	2	
12	2	
10	2	
8	2	→ 7 times
6	2	almost
4	2	$\frac{16}{2} = \frac{n}{2}$
2	2	

max. time $O(n)$

* min. time $O(1)$

(6)

```
Algorithm Test (n)
{
    if (n<5)                          ---> 1
    {
        printf ("%d", n);
    }
    else
    {
        for (i=0; i<n; i++)
        {
            printf ("%d", i);         ---> n
        }
    }
}
```

If condition is true - $\underline{O(1)}$ Best case

false - $O(n)$ worst case

and it not necessary to be best and worst.

```
Algorithm Test (n)
{
    if (n>5)
    {
        for (i=0; i<n; i++)
        {
            printf ("%d", i);
        }
    }
}
```

O - WC

θ - AC

$\underline{\Omega}$ - BC

Time complexity is $n>5$, $\underline{O(n)}$

So they need not be rule of best and worst.

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