

UNIT - 1

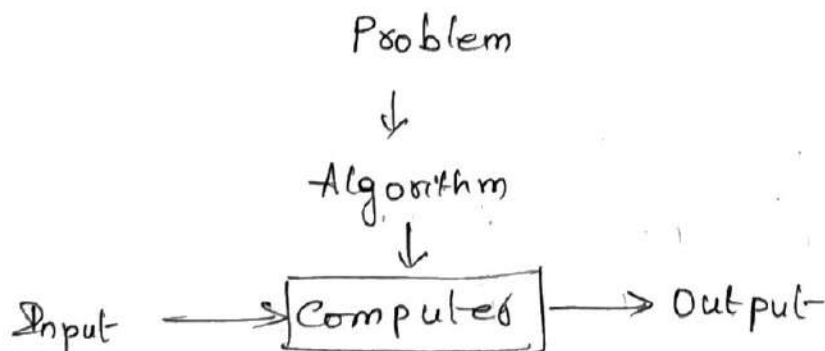
Introduction

Algorithm:-

Algorithm is a finite set of instructions, that is followed, to accomplish a particular task.

All algorithms must satisfy the following criteria

1. Input: zero or more quantities are externally supplied
Ex. $\downarrow$ ^{key of} Random no: prgs
2. Output: Atleast one quantity is produced
3. Definiteness: Each ~~ex~~ ins is clear and unambiguous
It must be perfectly clear not like, Add 7 or 6 to x, Compute 5/0.
4. Finiteness: If we trace out the ins of an algo, then for all cases, the algo terminates after a finite no. of steps.
5. Effectiveness: Every ins must be very basic, so that it can be carried out, in principle using only pencil & paper, i.e tracing each step should be possible.



Study of algos includes 4 distinct areas of research

1. How to devise algos? - It is an art, which can never be automated. Use any of the design techniques.
2. How to validate algos? - After devising, it is necessary to show that algo computes correct answer for all possible legal i/p's.

ie once Algo is transformed to prog lang, it should work correctly.
Once validation is done, prg is written.

Then the second phase, called prog proving or prg verification is being done. ie on real data ops are calculated & verified.

3. How to analyze algo :- ^{- check whether it meets client's. true needs & expectations}
- how accurately code carries out task

Analysis of algo or performance analysis refers to task of

determining how much computing time & storage an algo requires.

once an algo is executed, it uses the CPU to perform operations & its memory to hold prg & data.

This helps us to judge which algo is better than other.

" to predict whether slw meets efficiency constraint.

4. How to test a prg?

Testing has two phases — debugging

— Profiling.

Debugging - is a process of executing prg on sample data set to determine whether the faulty results occur & also if so, correct them.

Profiling or performance measurement - Process of executing a correct prg on datasets and measuring the time & space it takes to compute the results.

Algorithm Specification :- ^{- usage of particular instructions, or freq & duration of fun calls.}

Diff b/w algo and prg :- OS continues in a "wait" loop until more jobs are entered, it does not terminate unless the sys crashes. Since our prgs always terminate, we use algo & prg interchangeably.

- Algo uses natural lang like English.

We use pseudocode that resembles C and Pascal.

1. Comments begin with // and continue until end of line

2. Blocks are indicated with matching braces: { and }.

It represents any compound stmt Ex: procedure {
stmt1;
stmt2;
}

3. An identifier begins with a letter. Ex: var1, var2, x, y

Data types of vars are not explicitly declared, i.e types will be clear from the context.

Compound datatypes can be formed with records.

Ex node = record

```
{  
    datatype_1 data_1;  
    ;  
    datatype_n data_n;  
    node *link;  
}
```

link is a ptr to record type node.

Individual data items can be accessed with '→' or period.

Ex a) Let p points to record, then p → data_1, is the value of first field in the record. ~~or p.data_1~~

b) Let q ~~points~~ is a record of type node, q.data_1 denotes its first field.

4. Assignment of values to vars is done using the assignment stmt

<var> := <expr>;

5. There are two bool vals true and false

Logical operators + and, or & not

Relational " " =, <=, >=, ≠, >, and <

6. Elements of multidimensional arrays are accessed using
[and];

Ex: $A[i, j]$

7. Looping stmts

```
while (condn) do
{
    <stmt-1>
    :
    <stmt-n>
}
```

```
for var := Val1 to Val2 step
step-count do
```

```
{
    <stmt-1>
    :
    <stmt-1>
}
```

repeat-until

```
repeat
    <stmt-1>
    :
    <stmt-n>
until <condn>
```

write an algo to find the no. of
digits in binary rep of a positive
decimal integer.

Algorithm bitcount (n)

```
{ count := 0;
  while (n is != 0) do
      count++; n = n/2;
  write (count);
```

- ~~repeat~~ break

- used to force exit loops

- If break is used in nested loops, results in the

exit of innermost loop that is a part of

- return :- exits the loops, functions

8. Conditional stmt

if <condn> then <stmt>

if <condn> then <stmt1> else <stmt2>

- Case stmt

case

```
{
  : <condn1> : <stmt1>
  :
```

:

```
  : <condn1> : <stmt n>
  :
```

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

}

9. i/p and o/p are ^{done} using read and write

10. Algorithm Name (<par_list>)

An algo consists of a heading and a body.

Ex

Algorithm Max (A, n)

// A is an array of size n.

```
{
```

```
  Result := A[1];
```

```
  for i := 2 to n do
```

```
    if A[i] > Result then Result := A[i];
```

```
  return Result;
```

```
}
```

Result & i are local vars.

Ex

Algorithm TowersOfHanoi (n, x, y, z)

// Move the top 'n' disks from tower x to tower y.

```
{
  if (n > 1) then
```

```
    TowersOfHanoi (n-1, x, z, y);
```

Write ("move top disk from tower", x, " to top of

```
    tower ", y);
  TowersOfHanoi (n-1, z, y, x);
```

If 64 disks = $2^{64} - 1$ moves
If each move take one sec,
585 billion years to complete

Performance Analysis:-

Criteria upon which we can judge the algo

Ex:

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

Space Complexity:-

- Amount of mem it needs to run to completion.

Algo 1.

Algorithm abc(a, b, c)

{

return $a + b + b * c + (a + b - c) / (a + b) + 4 * c$;

}

Let one word is needed to store single var. a, b, c, result.
 $S_{PFO} \Rightarrow S(p) = 3 + 0 = 3 \therefore \text{space complexity} = O(1)$

The space needed by algo is sum of following components

1. Fixed part:- independent of characteristics such as ^{number} size of

I/Ps & O/Ps. It includes - instruction or code space

- space for simple vars.

- fixed-size component vars

- space for constants etc.

2. Variable part :- space needed by component vars whose size is dependent on the particular problem instance being solved
- space needed by reference vars.
 - recursion stack space

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

P - is any algo
 $S(P)$ - space reqd by P.
 C - constant
 S_p - instance characteristics.

Algo 2:

Algorithm Sum(a, n)

```

{
  S = 0.0;
  for i = 1 to n do
    S := S + a[i];
  return S;
}

```

The space needed by n is one word since it is an integer.

Space for a - n words.

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

↓
n, i, Total

∴ space complexity is $O(n)$

Algo 3:

Algorithm RSum(a, n)

```

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

```

The recursion stack space includes space for formal params, local vars, return address.

Return addr - req one word.

Each RSum call - " three words - space for vals of n
 " " return addr

Depth of recursion is $n+1$ $\Rightarrow$ recursion stack space needed is $\geq 3(n+1) \Rightarrow O(n)$

Time Complexity

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

$T(P)$ taken by program P is the sum of the compile time & run (or execution time).

Compile time does not depend on the instance characteristics. Assume that a compiled prog will be run several times without recompilation.

→ A program step is a meaningful segment of a prog that has an execution time that is independent of the instance characteristics.

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

step count = 1

2) comments → step count = 0

3) Assignment → = 1, if it does not have any other algo calls.

4) for, while, repeat-until → step count only for control parts

if i.e. for $i := \langle \text{expr} \rangle$ to $\langle \text{expr} \rangle$ do

while ($\langle \text{expr} \rangle$) do

For each execution of the control part of the for stmt is one.

There are two ways to calculate step count.

Method 1: Introduce a new var, count, into the prog, as a global var with initial val = 0.

Ex. 1 - Algo sum(a, n)

{ $s := 0.0;$

count := count + 1;

for $i := 1$ to n do

{ count = count + 1;

$s := s + a[i];$

}

// for for

count := count + 1; // for assignment

Count = count + 1; // For last time of for

count = count + 1; // For the return

return s;

} count after sum is executed = $2n+3$.
 $\Rightarrow$ Total no. of steps Sum takes = $2n+3$

Ex-2 Algorithm RSum(a, n)

{

count := count + 1; // if cond n

if (n ≤ 0) then

{

count := count + 1; // For return

return 0.0;

}

else

{

count := count + 1;

return RSum(a, n-1) + a[n];

}

}

Stepcount :-

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

These recursive formulas are referred to as recurrence relations.

To solve this, make repeated substitutions.

$$\begin{aligned} t_{\text{RSum}}(n) &= 2 + t_{\text{RSum}}(n-1) \\ &= 2 + 2 + t_{\text{RSum}}(n-2) \\ &= 2(2) + t_{\text{RSum}}(n-2) \\ &= n(2) + t_{\text{RSum}}(0) \\ &= 2n + 2 \quad n \geq 0 \end{aligned} \quad \left. \begin{array}{l} \Rightarrow \text{So, runtime grows linearly} \\ \text{to } n. \\ n\text{-instance characteristic} \end{array} \right\}$$

$$\Rightarrow \text{Stepcount} = 2n + 2$$

Ex. 3

Input size is one of the instance characteristic.

The i/p size of any instance of a problem is defined to be no. of words or elts needs to describe that instance.

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];
}
```

Annotations for the first algorithm:

- ← $count = count + 1$; // for i
- ← $count = count + 1$ // for j
- ← $count = count + 1$

⇒ Algorithm Add (a, b, c, m, n)

```
{
  for i := 1 to m do
  {
    count := count + 1; // for 'i' → m
    for j := 1 to n do
    {
      count := count + 1; // for 'j' → mn
      c[i,j] := a[i,j] + b[i,j];
      count := count + 1; // for assignment → mn
    }
    count := count + 1; → m // for loop initialization
  }
  count := count + 1; → 1
}
```

Annotations for the second algorithm:

- m
- mn
- mn
- m
- 1

$$m(2n+1) + 1 + m$$

At last step count = $2mn + 2m + 1$

Method 2

Build a table & list total no. of steps contributed by each statement.

S/e - steps per execution of the stmt.

frequency - No. of times each stmt is executed

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
Total			$2n+3$

statement	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
Algorithm RSum(a, n)	0	—	—	0	0
{					
if ($n \leq 0$) 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	$2+x$

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

statement	s/e	frequency	total steps
Algo 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)	m(n+1)
c[i,j] := a[i,j] + b[i,j];	1	mn	mn
}	0	—	0
Total			2mn + 2m + 1

Statement	s/e	
	n = 0 or 1	n > 1
Algorithm Fibonacci(n) // compute nth fibonacci number	1	
{		
if (n ≤ 1) then	1	1
write(n);	1	
else		1 + 1 2 stmts
f2 = 0; f1 = 1;		n
for (i := 2 to n do)		
{		n - 1
fn = f1 + f2;		
f2 := f1; f1 := fn;		2(n-1) → 2 stmts
}		0
write(fn);		1
}		
}		
		4n + 1

Summary:-

⇒ Time complexity is no. of steps taken by algo to compute. No. of steps depend on instance characteristics, no. of i/p's, o/p's, magnitude of i/p's & o/p's.

A step is any computation unit that is independent of these characteristics.

Ex. 10 addition can be one step

100 multiplications can also be one step.

but 'n' additions cannot, nor $n/2$ additions, $p+q$ subtracting

$\leq$ soon can be one step.

In few cases, alone no. of i/p's $\leq$ o/p's are not sufficient to compute time complexity.

Ex search algo terminates immediately if elt is found in first iteration.

There are three kinds of stepcounters.

1. best case stepcount - is min no. of steps, that can be executed for given par.

2. worst-case stepcount - is max "

3. Avg " - is avg "

Uses of stepcount calculation:-

1. Helps us to compare the time complexities of two algos that compute the same func and

2. Predict the growth in runtime as instance characteristics changes.

Asymptotic Notations or Mathematical Notations

It is another method to find time complexity.

Asymptotic means study of func of parameter 'n' and 'n' becomes larger and larger without bound.

We are concerned about how the running time of an algo increases with size of the i/p.

Let f and g are non-negative functions.

1. Big-Oh Notation (O)

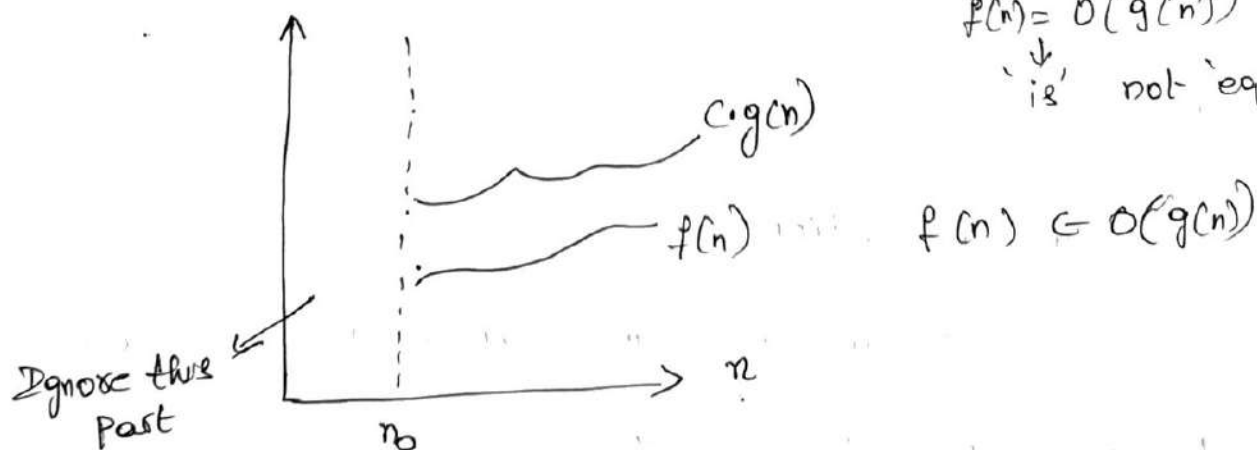
Big-oh gives an upper bound for the func $f(n)$.

i.e. The $g(n)$ will be the worst case that $f(n)$ takes to compute.

$g(n)$ is an upper bound on value $f(n)$

The function $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n) \forall n, n \geq n_0$

$f(n) = O(g(n))$
 $\downarrow$
'is' not 'equals'



Ex. 1) $f(n) = 3n+2$, Prove $f(n) = O(n)$

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

$$3n+2 \leq 4n \quad \forall n \geq 2$$

$$\downarrow$$
$$c=4$$

$$\downarrow$$
$$n_0=2$$

$$\therefore f(n) = 3n+2 = O(n)$$

n	$f(n) = 3n+2$	$\frac{f(n)}{g(n)} = 4n$
1	5	$\neq$ 4
2	8	$\leq$ 8
3	11	$\leq$ 12
4	14	$\leq$ 16
5	17	$\leq$ 20

$$2. \quad 3n+3 = O(n) \quad \left| \begin{array}{l} 3n+3 = O(n^2) \\ \text{as } 3n+3 \leq 4n \quad \forall n \geq 3. \end{array} \right. \quad (8)$$

$$3. \quad 100n+6 = O(n) \quad \left| \begin{array}{l} 3n+3 = O(n^3) \\ 3n+3 \neq O(1) \end{array} \right.$$

$$\text{as } 100n \leq 101n \quad \forall n \geq 6$$

$$4. \quad 10n^2+4n+2 = O(n^2) \quad \left| \begin{array}{l} 10n^2+4n+2 \neq O(n) \end{array} \right.$$

$$\text{as } 10n^2+4n+2 \leq 11n^2 \quad \forall n \geq 5.$$

$$5. \quad 1000n^2+100n-6 = O(n^2)$$

$$\text{as } 1000n^2+100n-6 \leq 1001n^2 \quad \forall n \geq 100$$

$$6. \quad 20n^3-3 = O(n^3)$$

$$\text{as } 20n^3-3 \leq 20n^3 \quad \forall n \geq 1$$

$$7. \quad 6 \times 2^n + n^2 = O(2^n)$$

$$\text{as } 6 \times 2^n + n^2 \leq 7 \times 2^n \quad \forall n \geq 4$$

→ Theorem :- If $f(n) = a_m n^m + \dots + a_1 n + a_0$ then $f(n) = O(n^m)$

2. Big-Omega Notation (Ω)

Ω gives lower bound behaviour of $f(n)$.

The lower bound implies below this time the algo cannot perform better, i.e. algos will take atleast this much of time,

$f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \geq c \times g(n) \quad \forall n, n \geq n_0$.

Ex: -1 $f(n) = 3n+2$, Prove that $f(n) = \Omega(n)$

$$3n+2 \geq 3n \quad \forall n \geq 1 \quad \left| \begin{array}{l} \text{also } 3n+2 = \Omega(1) \\ \downarrow \quad \downarrow \\ c=3 \quad n_0=1 \end{array} \right.$$

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

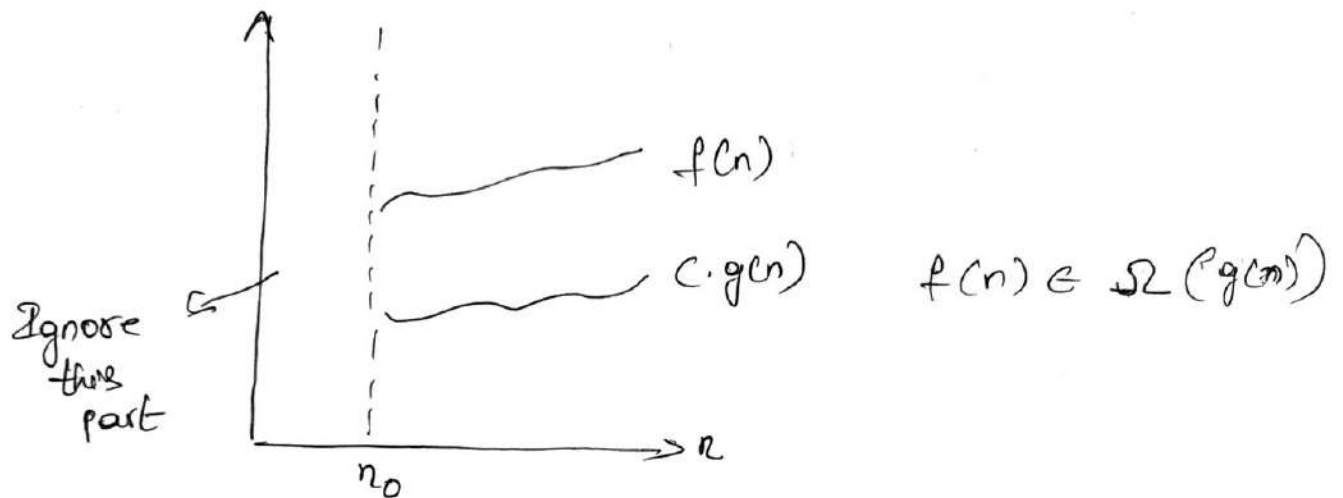
2. $100n + 6 = \Omega(n)$

as $100n + 6 \geq 100n$ for $n \geq 1$

3. $10n^2 + 4n + 2 = \Omega(n^2)$

as $10n^2 + 4n + 2 \geq n^2$ for $n \geq 1$

4. $6 * 2^n + n^2 = \Omega(2^n)$ as $6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$.



→ Function $g(n)$ is a lower bound on $f(n)$.

$g(n)$ should be as large as a func of 'n' as possible for which $f(n) = \Omega(g(n))$ is true

Theorem: -

If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then

$f(n) = \Omega(n^m)$.

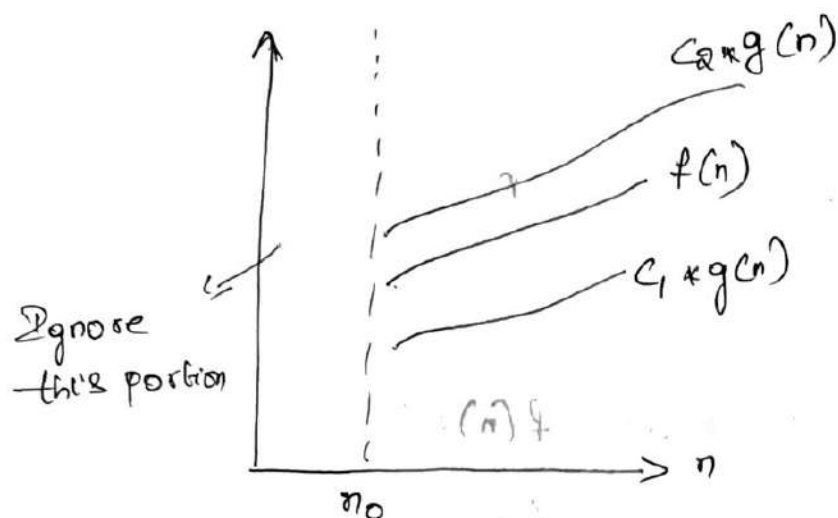
3. Theta Notation: (Θ)

For some func's the lower bound and upper bound may be same. i.e big-oh and omega will have the same func.

Ex. Find max & min elt in the given array. The computing time is $O(n)$ and $\Omega(n)$

The func $f(n) = \Theta(g(n))$ if $g(n)$ is both upper and lower bound on $f(n)$.

$f(n) = \Theta(g(n))$ iff there exists three positive constants c_1, c_2 and n_0 with the constraint that,

$$c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)| \quad \forall n \geq n_0$$


Ex. 1 Prove $3n+2 = \Theta(n)$

$$3n+2 \geq 3n \quad \forall n \geq 2 \quad \&$$

$$3n+2 \leq 4n \quad \forall n \geq 2 \quad , \quad c_1=3, \quad c_2=4 \quad \& \quad n_0=2$$

$$\Rightarrow 3n+2 = \Theta(n)$$

n	$3n$ $c_1 g(n)$		$3n+2$ $f(n)$		$4n$ $c_2 g(n)$
1	3	$\leq$	5	$\nless$	4
<u>2</u>	6	$\leq$	8	$\leq$	8
3	9	$\leq$	11	$\leq$	12

2. $3n+3 = \Theta(n)$

3. $10n^2+4n+2 = \Theta(n^2)$

4. $6 * 2^n + n^2 = \Theta(2^n)$

5. $10 * \log n + 4 = \Theta(\log n)$

6. $3n+2 \neq \Theta(1)$

→ Theta notation is more precise than both the big O & Omega notations.

Theorem: $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$

Little-Oh Notation

The func $f(n) = o(g(n))$ iff $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$

Ex 1. $3n+2 = o(n^2)$ since $\lim_{n \rightarrow \infty} \frac{3n+2}{n^2} = 0$

2. $3n+2 = o(n \log n)$ | $3n+2 \neq o(n)$

3. $6 + 2^n + n^2 = o(3^n)$

4. $f(n) = 2^n$, prove that $f(n) = o(n!)$

$f(n)$ depends on $\lim_{n \rightarrow \infty} \frac{2^n}{n!} = \lim_{n \rightarrow \infty} \frac{2^n}{n \times (n-1) \times (n-2) \dots \times 1}$
 $= 0$

Little-Omega

The fun $f(n) = \omega(g(n))$ iff $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$

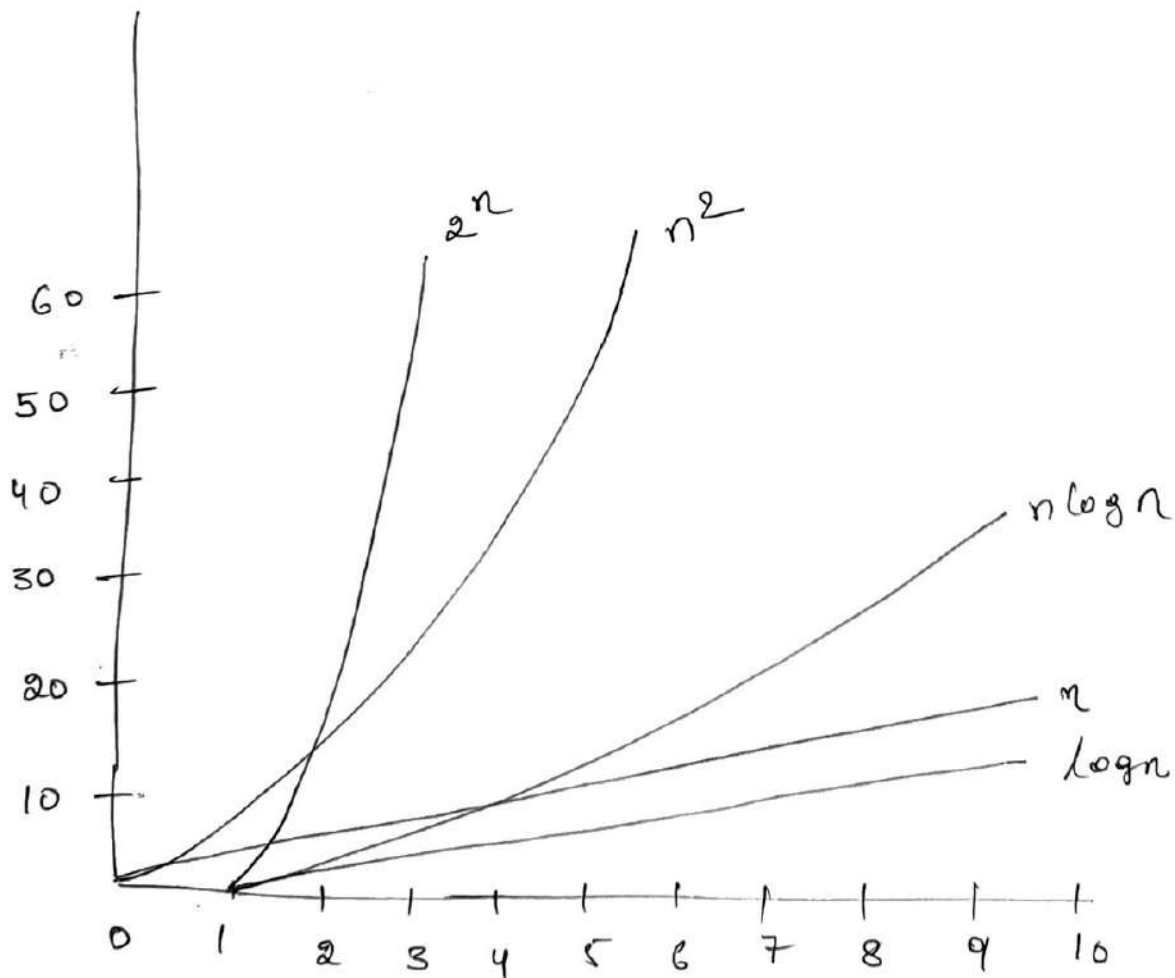
Practical complexities - or Order of growth

Suppose if algo are faster for smaller values of n and slower when n is large then we cannot say that these algo are good. This is called the order of growth of n .

Common time functions:-

Function	Name
1	constant
$\log n$	logarithmic
n	linear
$n \log n$	$n \log n$
n^2	quadratic
n^3	cubic
2^n	Exponential
$n!$	Factorial

n	$\log n$	$n \log n$	n^2	n^3	2^n	$n!$
1	0	0	1	1	2	1
2	1	2	4	8	4	2
4	2	8	16	64	16	24
8	3	24	64	512	256	40320
16	4	64	256	4096	65536	20922789888000
32	5	160	1024	32768	4.2×10^9	2.6×10^{35}
10^5	17	1.7×10^6	10^{10}	10^{15}	—	—
10^6	20	2.0×10^7	10^{12}	10^{18}	—	—



1) $\rightarrow$ If $n=40$, ϵ algo takes 2^n steps to compute, $\Rightarrow 1.1 \times 10^{12}$.

If speed of comp is one billion steps per sec, this requires about 18.3 mins.

$\rightarrow$ If $n=50$, algo runs 13 days

$\rightarrow$ If $n=60$, " 310.56 days

If $n=100$, algo runs 4×10^{13} yrs

So, algos with exponential complexity is limited to small $n \leq 40$.

2) If an algo needs n^{10} steps, then using our 1-billion-steps per sec comp, we need 10 sec when $n=10$

$n=100$, 3171 yrs

$n=1000$ 3.17×10^{13}

3) If algo complexity had been n^3 steps, then

$n=1000$, 1 sec

$n=10,000$ 110.67 mins

$n=1,00,000$ takes 11.57 days

~~$n=1000$, 110.67 mins~~

~~$n=10,000$ 11.57 days~~

$n=$

Divide-and-conquer

11

General Method

Given a function to compute on n ips, Divide-and-conquer splits the ips into k distinct subsets, $1 < k \leq n$, yielding k subproblems. The subproblems must be solved independently, & subsolns ^{are} then combined into a soln of the whole.

If the subproblems are relatively large, then D&C can be reapplied.

The subproblems resulting from a D&C are of the same type as the original problem.

D&C is recursive function.

Control abstraction:-

Algorithm DAndC(P)

§ if Small(P) then return $S(P)$;

else

§ divide P into smaller instances $P_1, P_2, \dots, P_k$, $k \geq 1$;

Apply DAndC to each of these subproblems;

return Combine($DAndC(P_1), DAndC(P_2), \dots, DAndC(P_k)$);

• $\downarrow$
 $\downarrow$

→ P - problem to be solved.

Small(P) - Boolean-valued func that determines whether the input size is small enough that the answer can be computed without splitting.

Combine() - function determines the soln to P using the solns to the k subproblems.

If the size of P is ' n ' & sizes of k subproblems are $n_1, n_2, \dots, n_k$ respectively, then the computing time of DAndC is a recurrence relation,

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \dots + T(n_k) + f(n) & \text{otherwise,} \end{cases}$$

$T(n)$ - time for DAndC on any i/p of size n .

$g(n)$ - time to compute the answer directly for small i/ps

$f(n)$ - Time for dividing P and combining the solns to subproblems.

Many DAndC ^{algs} ~~problems~~ have complexity of the form

$$T(n) = \begin{cases} T(1) & n=1 \\ aT(n/b) + f(n) & n>1 \end{cases}$$

where a and b are known constants.

Assume n is power of b .

1. Consider $a=2, b=2$. Let $T(1)=2$ and $f(n)=n$

Solve by substitution method.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + \frac{n}{2}] + n \\ &= 4[T(n/4)] + 2n \\ &= 4[2T(n/8) + \frac{n}{4}] + 2n \\ &= 8T(n/8) + 3n \end{aligned}$$

In general we see $T(n) = 2^i T(\frac{n}{2^i}) + i n$ for any $\log_2 n \geq i \geq 1$.

$$\text{for } i = \log_2 n \Rightarrow T(n) = 2^{\log_2 n} T(\frac{n}{2^{\log_2 n}}) + n \log_2 n$$

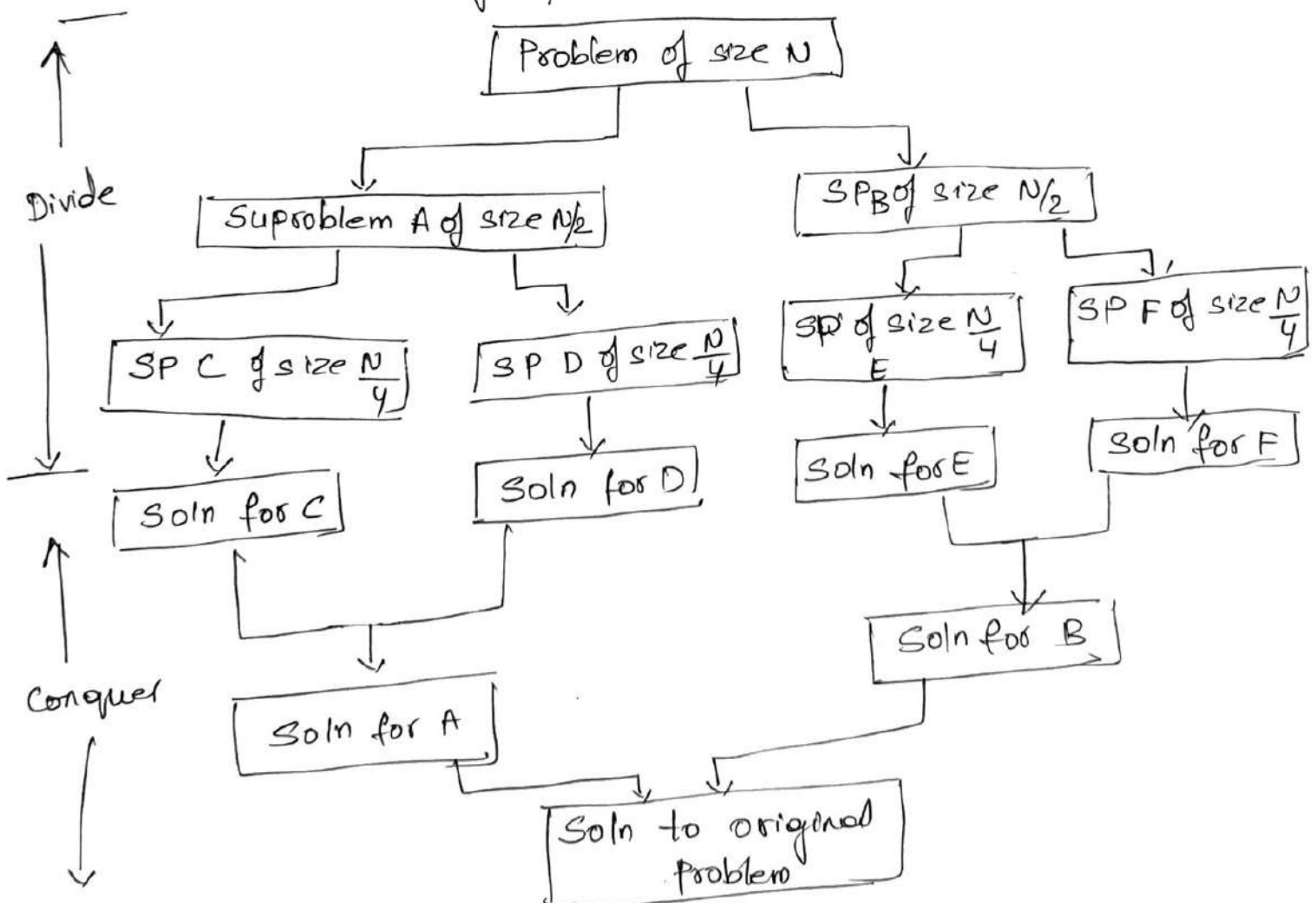
$$\Rightarrow T(n) = n T(1) + n \log_2 n \Rightarrow n \log_2 n + 2n$$

Divide & Conquer Defn:-

is a top-down technique for designing algorithms

It involves 3 steps at each level of recursion

- **Divide** - The problem is divided into a number of subproblems
- **Conquer** - The subproblems are conquered by solving them recursively. If the subproblems are smaller in size, they are solved directly.
- **Combine** - The solns of subproblems are combined to get soln for the larger problem.



Master's theorem for Dividing functions

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

$$\text{If } a \geq 1 \quad \& \quad f(n) = \Theta(n^k \log^p n) \\ b > 1$$

Find $\log_b a$

2. k

Case 1: if $\log_b a > k$ then $\Theta(n^{\log_b a})$

Case 2: if $\log_b a = k$

$$\text{if } p > -1 \quad \Theta(n^k \log^{p+1} n)$$

$$\text{if } p = -1 \quad \Theta(n^k \log \log n)$$

$$\text{if } p < -1 \quad \Theta(n^k)$$

Case 3: if $\log_b a < k$ if $p \geq 0 \quad \Theta(n^k \log^p n)$

$$\text{if } p < 0 \quad \Theta(n^k)$$

→ Solve the following recurrence relations by Master's theorem

$$\text{ii) } T(n) = 4T\left(\frac{n}{2}\right) + n \quad \text{where } n \geq 1 \text{ and is power of 2:}$$

$\downarrow \quad \quad \downarrow \quad \quad \swarrow$
 $a=4 \quad b=2 \quad \& \quad f(n)=n$

as $a \geq 1, b > 1$

$$f(n) = n^1 \Rightarrow k = 1$$

consider $\log_b a = \log_2 4 = 2$

check $\log_b a \quad k$
 $2 > 1$

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

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

$$\begin{array}{ccc} \downarrow & \downarrow & \\ a & b & f(n) = n \log n \end{array}$$

$$= n' \log' n \quad \xi \quad P = 1$$

$$\log_a b = \log_2 2 = 1 \quad = K = 1$$

$$\log_a b = K \quad \xi \quad P = 1 \text{ re } P > 1$$

$$\Rightarrow T(n) = \Theta(n^K \log^{P+1} n)$$

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

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

~~(iii)~~

Binary Search:-

Let $a_i, 1 \leq i \leq n$, be a list of elements that are stored in nondecreasing order. Problem is to determine whether a given element 'x' is present in the list. If 'x' is present, determine a value 'j' such that $a_j = x$. If 'x' is not in the list, then 'j' is to be set to zero.

Divide & conquer strategy for bin search :-

Let $\text{Small}(P)$ be true if $n=1$

If P has more than one element, then P should be divided into a new subproblems as,

Let $a(l:h)$ is the list of elts.

Pick an index $q, 1 \leq q \leq h$ and compare x with a_q .

Now 3 possibilities may occur:-

1. $x = a_q$ - P is immediately solved

2. $x < a_q$ - x has to be searched only in the sublist

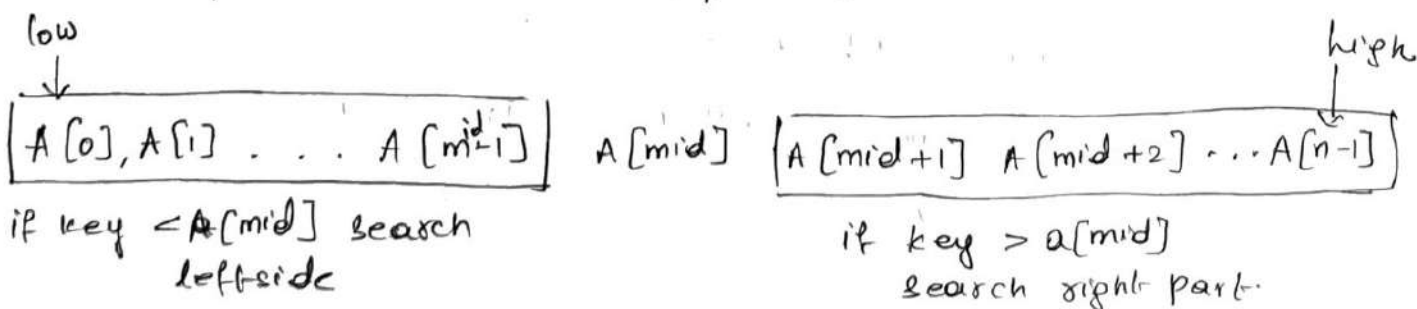
$a_l, a_{l+1} \dots a_{q-1}$

So, P reduces to $a(l:q-1)$

3. $x > a_q$ - x has to be searched only in the sublist

$a_{q+1}, a_{q+2} \dots a_h$

So, P reduces to $a(q+1:h)$



Recursive binary search:-

→ It is initially invoked as $\text{BinSrch}(a, l, n, x)$

Algorithm $\text{BinSrch}(a, l, h, x)$

/* Given an array $a[l:h]$ of elts in nondecreasing order, $1 \leq l \leq h$, determine whether x is present, and if so, return j such that $x = a[j]$; else return 0.

```
{
  if ( $h = l$ ) then // If small( $P$ )
  {
    if ( $x = a[l]$ ) then return  $l$ ;
    else return 0;
  }
```

else

```
{ // Reduce  $P$  into smaller subproblems.
```

```
  mid :=  $\lfloor (l + h) / 2 \rfloor$ ;
```

```
  if ( $x = a[mid]$ ) then return mid;
```

```
  else if ( $x < a[mid]$ ) then
```

```
    return  $\text{BinSrch}(a, l, mid-1, x)$ ;
```

```
  else
```

```
    return  $\text{BinSrch}(a, mid+1, h, x)$ ;
```

```
}
```

```
}
```

Iterative binary search:-

Algorithm $\text{BinSrch}(a, n, x)$

/* Given an array $a[1:n]$ of elts in nondecreasing order, $n \geq 0$, determine whether x is present, and if so, return j such that $x = a[j]$; else return 0.

low := 1 ; high := n ;

while (low ≤ high) do

{

mid := $\lfloor (low + high) / 2 \rfloor$;

if ($x < a[mid]$) then

high := mid - 1 ;

else if ($x > a[mid]$) then

low := mid + 1 ;

else return mid ;

}

return 0 ;

}

Ex. Let the array consists of

-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151

$x = 151$

low	high	mid	A[mid]	$x \neq A[mid]$
1	14	7	54	$151 > 54$
8	14	11	125	$151 > 125$
12	14	13	142	$151 > 142$
14	14	14	151	Found.

$x = -14$

low	high	mid
1	14	7
1	6	3
1	2	1
2	2	2
2	1	not found

$x = 9$

low	high	mid
1	14	7
1	6	3
4	6	5
		Found

→ Analysis of Comparisons

Comparison b/w x and $\text{ele of } a[]$. Assume only one comparison is needed to determine which of the three possibilities of the 'if' stmt holds.

The no: of ele comparisons needed to find each of the 14 ele .

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
a														
elements	-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons	3	4	2	4	3	4	1	4	3	4	2	4	3	4

$$\text{Avg no: of Comparisons for successful search} = \frac{3+4+2+\dots+4}{14} = \frac{45}{14} = 3.21$$

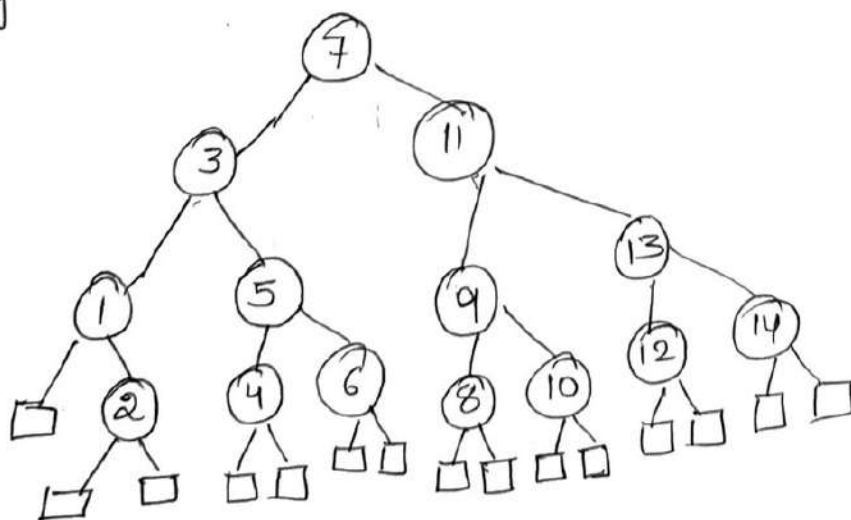
There are 15 possible positions that unsuccessful search can terminate

If $x < a[1]$, then algo requires 3 ele comparisons , to determine unsuccessfulness.

For all remaining possibilities, it requires 4 ele comparisons

$$\text{Avg no: of comparisons for unsuccessful search} = \frac{(3+14 \times 4)}{15} = \frac{59}{15} = 3.93$$

→ Binary decision tree for $n=14$



First comparison of x is with $a[7]$, if $x < a[7]$, then compares with $x[3]$...

If it is a successful search, then algo ends at one of the circular nodes.
- Internal nodes.

If it is unsuccessful search, algo terminates at one of the square nodes. - external nodes

→ If n is in the range $[2^{k-1}, 2^k]$, then BinSearch makes at most k elt comparisons for a successful search $\leq$ either $k-1$ or k

" " " " " " " " " " " "

→ Time Complexity:-

It is based on height of binary decision tree.

<u>Successful Searches</u>	<u>Unsuccessful Searches</u>
$\Theta(1)$ - best case	$\Theta(\log n)$ - best case
$\Theta(\log n)$ - average "	↳ avg "
$\Theta(\log n)$ - worst "	↳ worst "

Worst case - Every time make one comparison and divide given numbers n into $\frac{n}{2}$.

The recurrence relation is

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(\frac{n}{2}) + 1 & \text{otherwise} \end{cases}$$
$$= \lfloor T(\frac{n}{4}) + 1 \rfloor + 1$$
$$= \lfloor T(\frac{n}{8}) + 1 \rfloor + 2$$
$$= T(\frac{n}{2^3}) + 3$$
$$\vdots$$
$$= T(\frac{n}{2^k}) + k$$

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

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

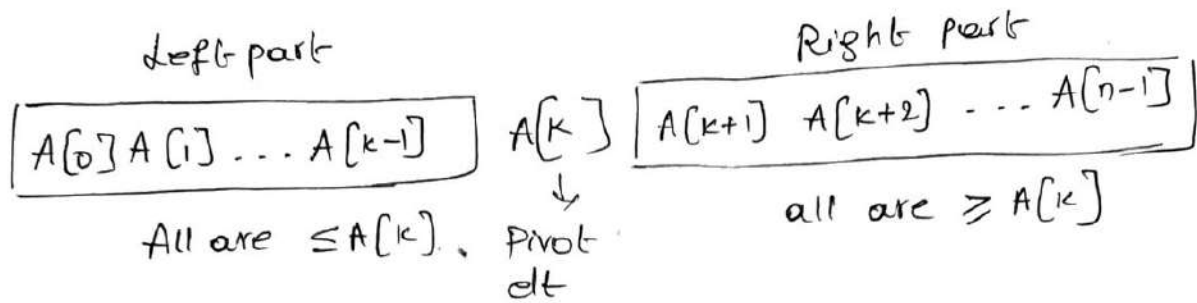
$\therefore T(n) = O(\log_2 n)$

Quick Sort (Partition exchange sort)

Works very well on large set of data.

Divide & Conquer applied to Quicksort

1. Divide the array $A[0] A[1] \dots A[n-1]$ into two subarrays.



2. Conquer: Sort the left part $A[0] \dots A[k-1]$ recursively
" right " $A[k+1] \dots A[n-1]$ "

Combine:
3. Since two subarrays are sorted in the previous step, no need to combine.

Applications: - Google uses advanced quicksort, to sort the web pages it finds as a result.

Algorithm Quicksort(p, q)

/* Sorts the elements $a[p], \dots, a[q]$ which reside in the global array $a[1:n]$ into ascending order; $a[n+1]$ is considered to be defined and must be $\geq$ all the elts in $a[1:n]$. */

{ if ($p < q$) then // If more than one elt.

{ // divide p into two subproblems

$k := \text{Partition}(a, p, q+1);$ // k is the position of partitioning elt.

// solve the subproblems.

Quicksort($p, k-1$);

Quicksort($k+1, q$);

}

}

Algorithm partition ($a, low, high$)

(15)

// Purpose: Divide the array into parts such that elements left of pivot are " $\leq$ pivot" and elts towards right of key are " $\geq$ pivot"

// low: The position of the first elt of array a .

// high: " " " "

pivot $\leftarrow a[low]$

$i \leftarrow low$

$j \leftarrow high$

while ($i \leq j$)

do $i \leftarrow i+1$ while (pivot $\geq a[i]$)

do $j \leftarrow j-1$ while (pivot $< a[j]$)

if ($i < j$) exchange ($a[i], a[j]$)

end while

exchange ($a[j], a[low]$)

return j .

→ Explain partition exchange sort- algo and trace this algo for $n=8$ elts. 24, 12, 35, 23, 45, 34, 20, 48.

quicksort (1, 8)

$4 \leftarrow k := \text{partition}(a, 1, 9);$

quicksort (1, 3)

quicksort (5, 8).

~~Ques~~ k := partition (a, 1, 9)

array: [1] 12 [2] 35 [3] 23 [4] 45 [5] 34 [6] 20 [7] 48 [8] 24 [9] 48

low = 1, high = 9, pivot = 24

i = 1, j = 9

1 ≤ 9 2 24 ≥ 12

3 24 < 35 8 24 < 48

7 24 < 20 3 < 7

exchange a[3] & a[7]

array: 12 20 23 45 34 35 48

3 ≤ 7 4 24 ≥ 23

5 24 < 45 6 24 < 34

5 24 < 45 5 & 5

5 ≤ 5 6 24 < 34 4 24 < 23 6 & 4

exchange a[4] & a[1]

array: 23 12 20 24 45 34 35 48

return 'j' = 4

Ex: 2 65 70 75 80 85 60 55 50 45

Analysis of quick sort:-

16

1. Best case :- If the key element is present exactly at the center dividing the array into two equal parts.

$$\text{Recurrence relation } T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(\frac{n}{2}) + T(\frac{n}{2}) + n & \text{otherwise} \end{cases}$$

Time reqd to sort left part of the array

Time reqd to sort right part

2) Solve by substitution method

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

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

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

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

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

$$= 2^i T(\frac{n}{2^i}) + i \times n$$

$$\text{Let } n = 2^i$$

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

$$= n \times 0 + n \log n$$

$$\Rightarrow \boxed{T(n) = \Theta(n \log n)} \quad \text{Avg Best case}$$

b) Solve by Master's theorem

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

$$a=2, b=2, f(n) = n^1 \log n \Rightarrow k=1 \Rightarrow P=0$$

$$\log_a b = \log_2 2 = 1 \Rightarrow \log_a b = k \quad \& \quad P = 0 > -1 \Rightarrow \Theta(n^1 \log^{0+1} n) \\ \Rightarrow \Theta(n \log n)$$

2. Worst case time efficiency of quicksort.

Worst case occurs when, at each invocation of the proc, the current array is partitioned into two subarrays with one of them being empty. This situation occurs if all the elts are arranged in ascending or descending order.

Ex $[22, 23, 44, 55]$ will yield following sequence of partitions

22 [33, 44, 55]

22 33 [44, 55]

22 33 44 [55]

$$\Rightarrow T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(n-1) + n & \text{otherwise} \end{cases}$$

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

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

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

$$\vdots$$
$$= T(n-n) + 1 + 2 + \dots + (n-1) + n$$

$$= 0 + 1 + 2 + \dots + n$$

$$= \frac{n(n+1)}{2} \leq n^2 \text{ for } n \geq 0$$

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

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

3. Average case time efficiency:-

Array may not be partitioned into two subarrays or may not be skewed as in worst case.

$$T(n) = \cancel{n} (n+1) + \frac{1}{n} \left(\sum_{1 \leq k \leq n} [T(k-1) + T(n-k)] \right) \quad \text{--- (1)} \quad (17)$$

↓
Partition

Multiply by 'n' on both sides

$$nT(n) = n(n+1) + \left(\sum_{1 \leq k \leq n} [T(k-1) + T(n-k)] \right)$$

$$nT(n) = n(n+1) + 2[T(0) + T(1) + \dots + T(n-1)] \quad \text{--- (2)}$$

Put $n = n-1$

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

Note $T(0) = T(1) = 0$

Perform (2) - (3)

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

Divide by $n(n+1)$

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

Substitution

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &= \frac{T(n-3)}{n-2} + \frac{2}{(n-1)} + \frac{2}{n} + \frac{2}{n+1} \end{aligned}$$

& soon.

$$\therefore \frac{T(n)}{(n+1)} = \frac{T(1)}{2} + 2 \sum_{3 \leq k \leq n+1} \frac{1}{k}$$

$$= 2 \sum_{3 \leq k \leq n+1} \frac{1}{k} \quad \text{--- (4) } [\because T(1) = 0]$$

$$\text{As } \sum_{3 \leq k \leq n+1} \frac{1}{k} \leq \int_2^{n+1} \frac{1}{x} dx = \log_e(n+1) - \log_e 2$$

$$\int \frac{1}{x} dx = \log_e x$$

Eqn ④ gives us,

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

→ Apply quicksort to sort the list E, X, A, M, P, L, E in alphabetical order.

Stable sorting algorithm:-

A sorting method is said to be stable if at the end of the method, identical elts occur in the same order as that of original list.

Quick sort is not a stable sorting method, but

Merge " is a stable sorting method.

Randomized quicksort algorithm:-

QuickSort() in previous section takes $O(n \log n)$ - Average time complexity

$O(n^2)$ - worst case " , where $n^2 > n \log n$.

so QuickSort() is improvised to RQuickSort(), which works on any input and runs in $O(n \log n)$ in all cases.

RQuickSort:-

while sorting the array $a[p:q]$, instead of picking $a[p]$, pick a random element among $a[p], \dots, a[q]$ as the partition element.

Algorithm RQuickSort(p, q)

Sorts the elements $a[p], \dots, a[q]$ which reside in the global array $a[1:n]$ into ascending order.

$a[n+1]$ is considered to be defined and must be ≥ 1 all the elements in $a[1:n]$. */

{ if ($p < q$) then

{ if ($(q-p) > 5$) then

Interchange (a , $\text{Random}() \bmod (q-p+1) + p$, p);

$j := \text{Partition}(a, p, q+1)$; // j is the position of the partitioning elem.

RQuickSort($p, j-1$);

RQuickSort($j+1, q$);

}

}

This algorithm is called Las Vegas algorithm because

- 1) it uses randomness to sort the ~~alg~~ elements and
- 2) it always gives output as correct answer.
- 3) or it informs about the failure.

We invoke the randomizer only if $(q-p) > 5$, coz if the elements are less than 5, then randomizer alone takes more time than the rest of the computation.

Let $A(n)$ be the average time of Quicksort on any input of n elements.

$$A(n) = \frac{1}{n} \sum_{1 \leq k \leq n} (A(k-1) + A(n-k)) + n + 1$$

coz the number of elements in the second part will be $0, 1, 2, \dots, n-2$ or $n-1$, ^{all} with an equal probability of $\frac{1}{n}$

This ~~$A(n)$~~ $A(n)$ eqn is same as average case time complexity of quicksort $\Rightarrow O(n \log n)$

Randomized algorithm for sorting

Algorithm RSort(a, n)

// Sort the elements $a[1:n]$

1 Randomly sample s elements from $a[]$;

Sort this sample;

Partition the i/p using the sorted sample as partition keys

2 sort each part separately;

Merge Sort:-

18

Assume the elements to be sorted in nondecreasing order.

Given a sequence of 'n' elts (also called keys) $a[1], \dots, a[n]$, the idea is to imagine them split into two sets $a[1], \dots, a[\lfloor n/2 \rfloor]$ and $a[\lfloor n/2 \rfloor + 1], \dots, a[n]$. Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of n elts.

1. Divide:- Partition array into two sublists s_1 and s_2 with $n/2$ elts each.

2. Conquer:- Recursively sort s_1 and s_2 .

3. Combine:- Merge s_1 and s_2 into a unique sorted group.

Algorithm MergeSort (low, high)

// $a[\text{low}:\text{high}]$ is a global array to be sorted.

Small(P) is true if there is only one elt. */

{ if (low < high) then // if there are more than one elt

{ // Divide P into subproblems

mid := $\lfloor (\text{low} + \text{high}) / 2 \rfloor$;

// Solve the subproblems.

MergeSort (low, mid);

MergeSort (mid+1, high);

// Combine the solutions:

Merge (low, mid, high);

}

}

Algorithm Merge (low, mid, high)

/* $a[\text{low}:\text{high}]$ is a global array containing two sorted subsets in $a[\text{low}:\text{mid}]$ and $a[\text{mid}+1:\text{high}]$. This algo merges the two sets into a single set residing in $a[\text{low}:\text{high}]$.

/* $b[]$ is an auxiliary global array */

```
{
  h := low; i := low; j := mid+1;
  while ((i ≤ mid) and (j ≤ high)) do
  {
    if (a[i] ≤ a[j]) then
```

```
    {
      b[h] := a[i]; i := i+1;
```

```
    }
```

```
  else
```

```
  {
```

```
    b[h] := a[j]; j := j+1;
```

```
  }
```

```
  h := h+1;
```

```
}
```

```
if (i > mid) then
```

```
  for k := j to high do
```

```
  {
```

```
    b[k] := a[k]; k := k+1;
```

```
  }
```

```
else
```

```
  for k := i to mid do
```

```
  {
```

```
    b[k] := a[k]; k := k+1;
```

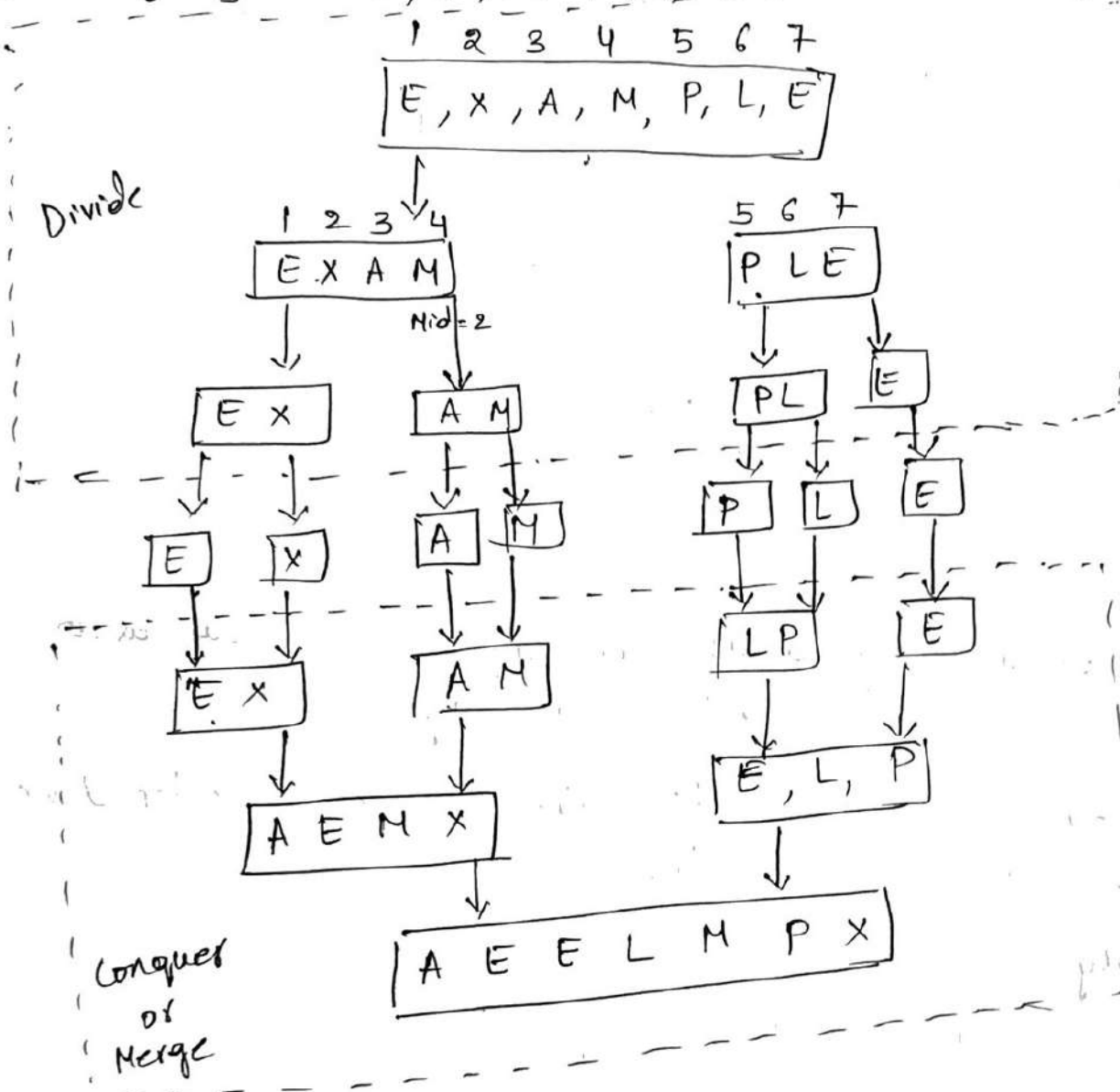
```
  }
```

```
  for k := low to high do a[k] := b[k];
```

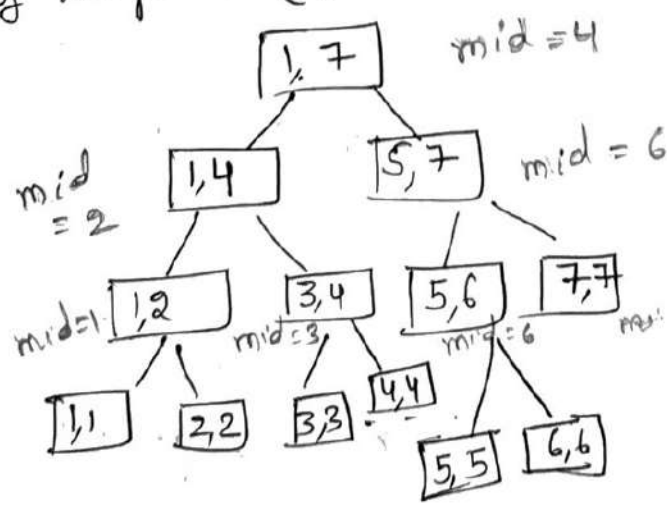
```
}
```

Ex-1 $a[1:10] = (310, 285, 179, 662, 351, 423, 861, 254, 450, 520)$ (19)

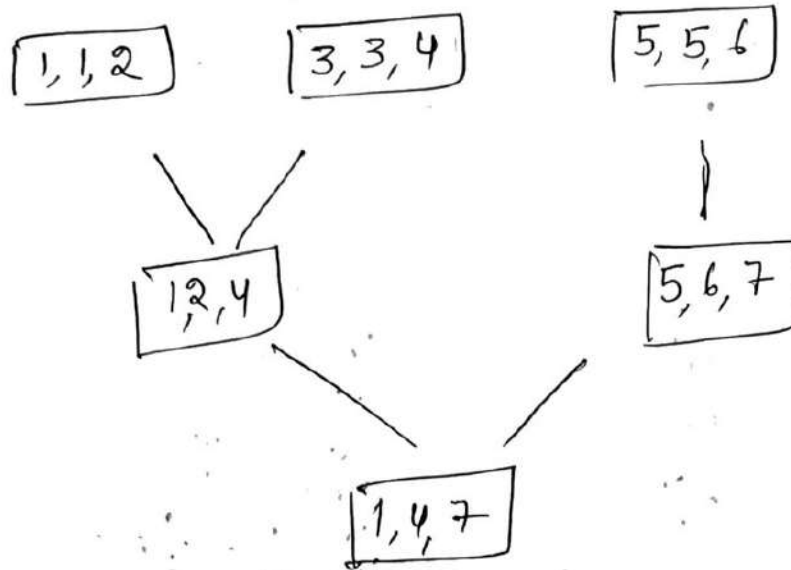
Ex-2 $a[1:7] = \text{"E", "X", "A", "M", "P", "L", "E"}$



Tree of calls of MergeSort (1, 7)



Tree of calls of Merge



Limitations:-

1. It needs an auxiliary array, same amount of storage as i/p array. So, this algo is not in-place algo.
2. Stack space for recursion; b/c2 algo is designed in top-down fashion. Stack takes $O(\log_2 n)$ space.

Time complexity:-

$$T(n) = \begin{cases} 1 & \text{when } n=1 \\ 2T\left(\frac{n}{2}\right) + cn & \end{cases}$$

When $n = 2^k$

$$\begin{aligned} T(n) &= 2 \left[2T\left(\frac{n}{4}\right) + \frac{cn}{2} \right] + cn \\ &= 2^2 T\left(\frac{n}{2^2}\right) + 2cn \end{aligned}$$

$$= 2^k T\left(\frac{n}{2^k}\right) + k \cdot cn$$

$$= n + \log_2 n \cdot cn$$

$$\Rightarrow T(n) = O(n \log_2 n)$$

Space

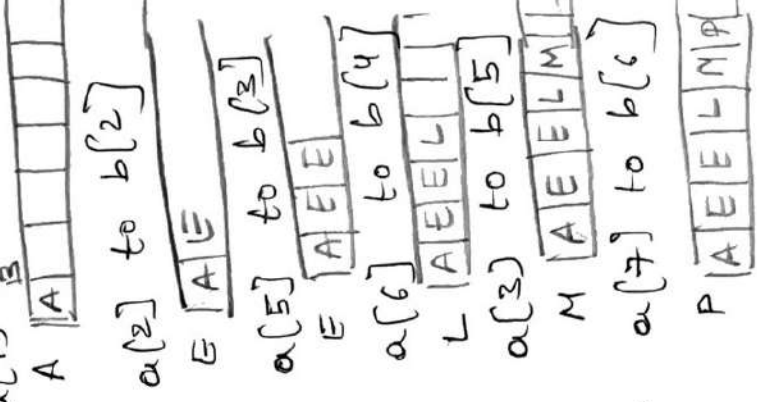
	[1]	[2]	[3]	[4]	[5]	[6]	[7]
a →	A	E	M	X	E	L	P
b →	A	E	E	L	M	P	X

k	i	j
1	1	5
2	2	
3	3	
4	4	6
5	5	7
6	6	
7	7	8

Tracing of merge (A, E, M, X) to (E, L, P)
 merges to (A, E, E, L, M, P, X)
 low-1, mid-4, high-7

$a[i] \leq a[j]$
 $a[1] \leq a[5]$ → copy a[1] to b[1]
 $A \leq E$
 $a[2] \leq a[5]$ → " a[2] to b[2]
 $E \leq E$
 $a[3] \leq a[5]$ → " a[3] to b[3]
 $M \neq E$
 $a[3] \neq a[5]$ → " a[6] to b[4]
 $M \neq L$
 $a[3] \neq a[5]$ → " a[3] to b[5]
 $M \leq P$
 $a[3] \leq a[7]$ → " a[7] to b[6]
 $a[4] \neq a[7]$
 $X \neq P$

$i > mid$
 $4 \neq 4$
 $k = 4$ to 4 copy a[4] to b[7]
~~X~~



Strassen's Matrix Multiplication

Let A and B be two $n \times n$ matrices. The product matrix $C = AB$ is also an $n \times n$ matrix, whose i, j th element is formed by taking the elements ⁱⁿ of the i th row of A and j th column of B .

$$C(i, j) = \sum_{1 \leq k \leq n} A(i, k) \cdot B(k, j) \quad \forall i, j \text{ between } 1 \text{ and } n.$$

To compute $C(i, j)$ we need n multiplications.

$\Rightarrow$ Conventional matrix multiplication takes $\Theta(n^3)$

Divide-&-conquer strategy:-

Let n is power of 2, $n = 2^k$. If n is not power of 2, then append enough rows & columns of zeros to both A & B , so that resulting dimensions are a power of two.

Imagine A & B are partitioned into 4 square submatrices, each submatrix having dim $\frac{n}{2} \times \frac{n}{2}$.

Then product AB of 2×2 matrices is:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\text{then } C_{11} = A_{11} B_{11} + A_{12} B_{21}$$

$$C_{12} = A_{11} B_{12} + A_{12} B_{22}$$

$$C_{21} = A_{21} B_{11} + A_{22} B_{21}$$

$$C_{22} = A_{21} B_{12} + A_{22} B_{22}$$

Since n is a power of 2, these matrix products can be recursively computed by the same algo, until submatrices become small with $n=2$.

To compute AB , we need 8 multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and 4 additions of $\frac{n}{2} \times \frac{n}{2}$ matrices

$$T(n) = \begin{cases} b & n \leq 2 \\ 8T(\frac{n}{2}) + cn^2 & n > 2 \end{cases}$$

$\hookrightarrow$ addition time

b & c are constants.

$$\Rightarrow T(n) = O(n^3)$$

Hence, no improvement than traditional method.

Strauss discovered a way to compute c_{ij} 's using 7 multiplications and 18 additions or subtractions.

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22})B_{11}$$

$$R = A_{11}(B_{12} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12})B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

Resulting recurrence relation for

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(\frac{n}{2}) + an^2 & n > 2 \end{cases}$$

$$T(n) = an^2 \left[1 + \frac{7}{4} + \left(\frac{7}{4}\right)^2 + \dots + \left(\frac{7}{4}\right)^{k-1} \right] + 7^k T(1)$$

$$\leq cn^2 \left(\frac{7}{4}\right)^{\log_2 n} + 7^{\log_2 n}, \quad c \text{ a constant}$$

$$= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7}$$

$$= O(n^{\log_2 7}) \approx O(n^{2.81})$$

$$T(n) = 7T\left(\frac{n}{2}\right) + cn^2$$

$$= 7\left(7T\left(\frac{n}{4}\right) + c\left(\frac{n^2}{4}\right)\right) + cn^2$$

$$= 7^2\left(7T\left(\frac{n}{8}\right) + c\left(\frac{n^2}{16}\right)\right) + 7c\left(\frac{n^2}{4}\right) + cn^2$$

$$= 7^3 T\left(\frac{n}{2^3}\right) + cn^2\left(\frac{7}{4}\right)^2 + cn^2\left(\frac{7}{4}\right) + cn^2$$

$$\vdots$$

$$= 7^k T\left(\frac{n}{2^k}\right) + cn^2 \left[\left(\frac{7}{4}\right)^{k-1} + \left(\frac{7}{4}\right)^{k-2} + \dots + \left(\frac{7}{4}\right) + 1 \right]$$

$$= 7^k T\left(\frac{n}{2^k}\right) + cn^2 \left(\frac{7}{4}\right)^k$$

$$= 7^k T\left(\frac{n}{2^k}\right) + cn^2 \left(\frac{7^k}{4^k}\right) \quad \text{Let } n = 2^k, \quad k = \log_2 n$$

$$= 7^{\log_2 n} T(1) + cn^2 \frac{7^{\log_2 n}}{4^{\log_2 n}}$$

$$= 7^{\log_2 n} T(1) + cn^2 \frac{n^{\log_2 7}}{n^{\log_2 4}}$$

$$= n^{\log_2 7} + c \cdot \cancel{n^2} \frac{n^{\log_2 7}}{\cancel{n^2}}$$

$$= (1+c) n^{\log_2 7}$$

$$\therefore \text{Time complexity is } O\left(n^{\log_2 7}\right) \text{ or } O\left(n^{2.81}\right)$$