29/4/21

Unit-1

* Algorithm is a step by step procedure or design process to accomplish a particular task.
* We want to design algorithms which are fast means which give answer very quickly.

Analytical Approach:

1. Build a mathematical model of computer.
2. In this model we will design algos and study the properties of algos.
3. Reasoning about the algorithms.
   (prove facts about the time taken and space taken by algo correctness).

PBM: Given m & n find their greatest common divisor.

Algorithm 1: Simple Factorising Algorithm

Input: two integers m, n.

Output: largest integer that divides both m & n without leaving a remainder.

Algorithm:

Step 1: find the factor of m (prime factor)
Step 2: find the factor of n (prime factor)
Step 3: Identify the common factor (CF).
Step 4: Multiply CF & return result.

ex: 48 36

m = 36 $2 \times 2 \times 3 \times 3$

n = 48 $2 \times 2 \times 2 \times 2 \times 3$

CF 2 2 3

12 (GCD)

Total 9 divisions (4+5).

Algorithm 2: Euclid Algorithm.

Input : two integers m, n.

Output : largest integer that divides both m & n without leaving a remainder.

Algorithm

```
Euclid(m,n)
begin while(m does not divide n)
    r = n mod m
    n = m
    m = r
end while
return m.
```

T.C :

$\therefore O(\log_{3/2}(m+n))$

T.C

1) $\frac{m+n}{3/2}$

2) $\frac{m+n}{(3/2)^2}$

:

k) $\frac{m+n}{(3/2)^k}$

sum reduces by min of 3/2 in each iteration

min is $\frac{m+n}{(3/2)^k} = 1$

$\Rightarrow k = \frac{\log(m+n)}{\log(3/2)}$

2 divisions.

$\Rightarrow \boxed{k = \log_{3/2}(m+n)}$

ex: n = 48, m = 36.

```
36 | 48 ( 1
     36
    ---
     12
```

n = 36
m = 12

m = 12 ✓

Hence Euclid's Algorithm is more efficient & fast.

Let's consider another example.

Algorithm 1: Simple Factorising Algorithm.

m = 434 → 2 × 7 × 31

n = 966 → 2 × 3 × 7 × 23

2 × 7 = 14 (GCD).

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31

$^{31}(11) + (9)^{23} + 1$

20 divisions.

Algorithm 2: Euclid Algos.

```
      2
434 | 966
      868
     ---
      r = 98
```

n = 434
m = r = 98

```
      4
98 | 434
     392
    ---
     r = 42
```

n = m = 98
m = r = 42

```
      2
42 | 98
     84
    ---
     r = 14
```

n = 42
m = 14 GCD.

4 divisions ←

Hence Euclid's Algo. is very fast & efficient.

Algorithm :-

It is an abstract computation procedure which takes some value(s) as input and produce a value(s) as output

Program	Algorithm
* Concrete (exact implementation of idea)	* Abstract implementation of ~~data~~ idea.
* Write program @ implementation time	* Write algorithm @ design time.
* Depends on platform (hardware + OS)	* Independent of platform.
* Domain knowledge independent.	* Domain knowledge dependent.
* We test the program. — watch time.	* We analyze the algorithm — function of time (growth of the time function)
"Posteriori testing."	"Priori analysis" Will measure growth of problem size (input size).

Characteristics of an algorithm:-

1. Input — an algo must have 0 or more inputs.
2. Output — it must have atleast 1 o/p.
3. Finiteness — it must terminate.
4. Definiteness — each step of algorithm must be unambiguous (precisely define).
5. Effectiveness — each step of algorithm must be correct and it should happen in finite amount of time.

How to express an algorithm?

1. Natural Language
   - → doesn't have a structure.
   - → Reaches wide audience (advantage).
   - → It is ambiguous and vague because the sense of words may be diff. in diff. lang. (dis. adv.).
2. Pseudo Code
   - → it uses programming constructs to write a algo.
   - → there is no standard of writing it, we cannot visualise the algorithm (dis. adv.).
   - → language independent (advantage).
     - any language programmer can come along, read the pseudo code and translate it into its own programming language.
3. Flow Chart
   - ① square : steps of algorithm.
   - ② circle : start and end of algorithms.
   - ③ Rhombus : condition/selection step.

advantage: visualize the algorithm

ex: Find num is even or odd.

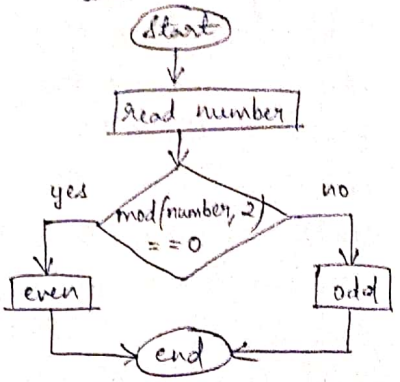
Natural Language:

1. Store the number
2. Divide the number by 2 and find the remainder
   - if remainder is zero
     - even
   - otherwise
     - odd.

Pseudo Code:

```
input the number
if mod(number, 2) == 0
even
else
odd.
```

Flow Chart:



How to do Analysis? (TC & SC)

1. Time complexity.
   - ↳ how much time the algo has taken on RAM (Random Access Machine)
   - ① this time will be $T(n)$, where this indicates the Maximum time taken by algo on RAM.
   - * ② We are not interested in exact (precise) value of $T(n)$ but we are interested in the shape of $T(n)$ or order of $T(n)$ or Growth of $T(n)$.
     - Ex: $T(n) = an^2 + bn$
       the time will be order of n^2 i.e $O(n^2)$.
   - ③ Actually is the measure of Goodness for the algorithms.

ex: A and B both are solving the same problem.
but A is taking $O(n)$ and b is taking $O(n \log n)$
So A is better as it takes less time.

(a) Large n is important.
n is the input size
as n goes infinity or n is very large then which algo is good.

Based on the growth of the $T(n)$, time complexity can be of many types:

$T(n)$	Type of $T(n)$	Example Algo.
$O(1)$	constant T.C	find given no. is even/odd
$O(n)$	linear T.C	finding max/min from array
$O(n^2)$	quadratic T.C	Bubble sort
$O(n^3)$	cubic T.C	Matrix multiplication.
$O(\log n)$	logarithmic T.C	Binary Search.
$O(n \log n)$	linear arithmatic T.C	Merge Sort
$O(2^n)$	exponential T.C	finding all subset of set
$O(!n)$	Factorial T.C	finding permutation of numbers

The base of log is 2 is DAA

Methods to compute T.C :-
1. Frequency method
2. Tree/Graph method
3. Substitution method
4. Master method.

* just find the total no. of iterations to find T.C.

ex: (1) swap(a,b)
temp = a : 1
a = b : 1
b = temp : 1
3 $O(1)$ [∵ constant]

(2) for i 1---n i++ : n+1
statement : n
2n+1
$O(n)$.

(3) for i 1---n i+2
statement (n/2)
$O(n)$

(4) for i 1---n i++
for j 1---n j++
statement
explanation:
i=1 j=1---n
i=2 j=1---n
⋮
i=n j=1---n
n * n = n^2
∴ $O(n^2)$.

(5) for i 1---n i++
for j 1---i j++
explanation: ↓

		iterations
i=1	j=1	= 1
i=2	j=1,2	= 2
i=3	j=1,2,3	= 3
⋮		
i=n	j=1,2,3,...n	= n
		1+2+...n
		= n(n+1)/2

∴ $O(n^2)$.

(6) n = 100
p = 0
for(i=1; p<=n; i++)
p = p+i
explanation:

i	p
1	0+1=1
2	1+2=3
3	3+3=6
4	6+4=10
⋮	⋮
k	1+2+...+k
	k(k+1)/2

Stopping condition is p>n
k(k+1)/2 > n
$k^2 > n$
k = root(n).
∴ $O(\sqrt{n})$.

(7) for i=1---n i++ : n+1
for j=1---n j++ : n+1
c[i,j] = 0 : n
for k=1---n k++ : n+1
c[i,j] = c[i,j] + a[i,k] * b[k,j] : n
$O(n^3)$
(2n+1) * (2n) * (n+1)
∴ $O(n^3)$

(8) for i=1..n i*2

i	no. of iteration
1	2^0
2	2^1
4	2^2
8	2^3
⋮	
i=xyz	$2^k > n$

take log on b.s.

$k > \log_2 n$.

$k > \text{ceil}(\log_2 n)$.

n=10 ↳ $\log_2 10 = 3.2$ then k should be k=4 log. So ceil of it.

Mention floor/ceil of obtained floating point no.

(9) for(i=n; i>=1; i=i/2)

i	no. of iteration
n	
n/2	
n/4	
n/8	
⋮	
$n/2^k < 1$	
$2^k > n$	
$k > \log_2(n)$	

if n = 10, $\frac{10}{2} = \frac{5}{2} = \frac{2.5}{2}$ (3) $= \frac{1.5}{2} < 1$

but $\log_2(10) = 3.2$ So k = floor($\log_2 n$)

$\therefore k = \text{floor}(\log_2 n)$

(10) for(i 1--n i=i*2) = log n

p++

for(j=1--p j=j*2) = log p

$\therefore O(\log p)$

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

i.e ceil($\log_2$(ceil($\log_2 n$)))

(11) for i=0..n i++ = n

for j=1..n j*2 = log n

$\therefore O(n \log n)$.

(12)

a=1

while(a<b)

statement

a = a*2

$\therefore O(\log_2 b)$

(13)

while(m!=n)

if(m>n)

m = m-n

else

n = n-m

if m=5 n=5 no. of iterations = 0 (best case).

if m=16, n=2 no. of iterations

14	2	8 = n/2
12	2	O(n/2)
10	2	= O(n)
8	2	
6	2	
4	2	
2	2	

B.C O(1)

W.C O(n)

(14)

i=1

j=1

while(j<n)

statement

j = j+i

i++

→ O(root(n))

2. Space Complexity.

↳ total space occupied during execution including the input values.

Space complexity = no. of words * word size

Space complexity = auxiliary space + space used by input values.

(temporary space)

→ the lesser the space, the faster the algo.

ex: ① int a,b,c 1,1,1

c = a+b

S.C = O(1) ↻ 3

② int i, n, sum = 0 1,1,1

int a[n] n

for i=0 n i++

sum = sum + a[i]

n+3

$\therefore$ S.C = O(n)

③ for i=0 n i++

for j=0 n j++

c-ij = a-ij + b-ij

→

i	j	a	b	c	n
1	1	n*n	n*n	n*n	1

$= 3n^2 + 3$

$\therefore O(n^2)$.

Asymptotic Notations :-

* Asymptotic notations is a formal way to speak about the functions and classify them.
* Asymptotic analysis refer to the classification of functi

In the Asymptotic Notations, we have ...,

θ	"theta class"	Average bound
O	"Big Oh"	Upper bound
Ω	"Omega (little)"	Lower bound

→ When we classify the functions in classes, then we aspect two properties from this classes,

1. Should be worried about the shape, behaviours of the function when is sufficiently large? give more

importance to the behaviour of function when n is large.

2. Ignore the constant multipliers.

$2n+3$, $3n+3$ } both belong to same class.

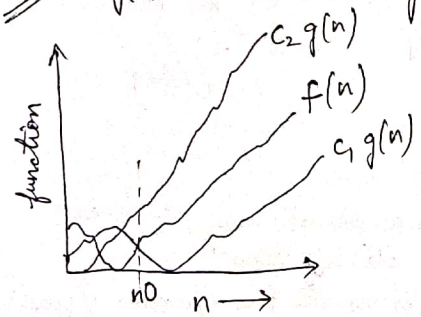
→ Measure the growth function

1. If two functions have same growth then they belong to same class and can be related with Theta notation.

If two functions do not have same growth then theta notation is not useful instead we will use big Oh or Omega notation.

ex: Let $f(n)$, $g(n)$ be two functions.

Theta:- $f(n)$, $g(n)$ } non-negative functions i.e. non-negative output functions.



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

$\Updownarrow$

$\boxed{f = \Theta(g)}$

$\Theta(g)$: { f | f is a non-negative function and there exist constants c1, c2, n0 such that,

$c1 * g(n) \leq f(n) \leq c2 * g(n)$ for $n \geq n0$.

↑ Theta Notation

ex: $10n^3 + \log n = f$

$10n^3 \leq f \leq 11n^3$

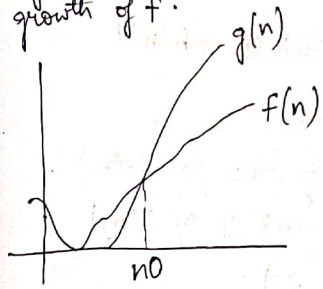
c_1 c_2

$\therefore \Theta(n^3)$.

O "Big Oh":-

Theta class pose a upper and lower bound on the growth of f.

Big O class only pose upper bound on the growth of f.



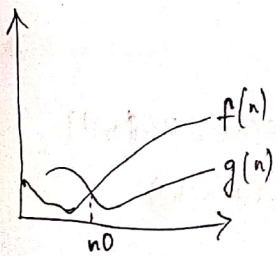
$F = O(g)$

$O(g) = \{ f$ where f is non-negative there exist c_1, n0 such that,

$f(n) \leq c_1 O(g(n))$

$\forall n \geq n0$.

Little Omega:-



$f = \Omega(g)$

$f(n) \geq c_1 (\Omega(g(n)))$

$\forall n \geq n0$.

Important properties of Asymptotic Notations :-

① General Property :

If $f(n) = \Theta(g(n))$ then we can conclude that $f(n) = \Omega\, g(n)$ and $f(n) = O\, g(n)$

ex: $f(n) = 2n^2+5 \rightarrow O(n^2)$

$g(n) = n^2$

$\frac{2n^2+5}{2} \leq 2n^2+5 \leq 7(2n^2+5)$

② Transitive Property :

If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n) = O(h(n))$.

③ Symmetric Property : (only true for theta).

If $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$ (will not apply for O, Ω).

④ Transpose Symmetric :

If $f(n) = O(g(n))$ then $g(n)$ is $\Omega(f(n))$.

⑤ (Imp. for gate).

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

$d(n) = O(e(n))$

then $f(n)+d(n) = \max(O(f(n)), O(d(n)))$ (or) $O(g(n)+e(n))$.

ex: $\max(n^2, n^3)$ (or) $O(n^2+n^3)$

$O(n^3)$ $O(n^3)$.

⑥ $f(n) * d(n) = O(g*e) \Rightarrow f*d = O(g*e)$.

Best, Average, Worst Case Complexities :-

→ Linear Search, Binary Search.

9	10	11	12	13	2	5

Key = 12

Comparing each and every number.

Algorithm :- Linear Search

Input : List / array of elements and a search key.

Output : Index of search key if found.

```
Start
 Linear_Search (A[], key)
  i = 0
  for( i=0; i< length(A); i++)
    if (key == A[i])
      then return i
End
```

If search key = 9, then we require only 1 comparison. So best case is $O(1)$ here.

If search element is 5, in worst case, then we have n comparisons.

So here worst case is $O(n)$.

Average case,

elements	comparisons
9	1
10	2
11	3
12	4
13	5
2	6
5	7

Avg Complexity = $\frac{\text{Complexities of all possible cases}}{\text{total no. of cases}}$

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

Avg Case $O\left(\frac{n}{2}\right)$.

Best case Complexity :-

* It is the case that causes minimum no. of operations that an algo has to do.
* This calculates the lower bound on the time complexity of algorithm.

Worst case Complexity :-

* It is the case that causes maximum no. of operations that an algo has to do.
* Places upper bound time complexity of algorithm.

→ Best case algo. will not always be $O(1)$. It changes with algo.

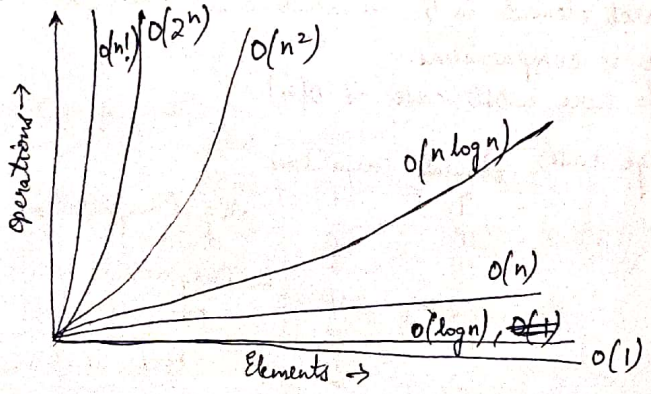
ex: Merge Sort

Best case : $O(n \log n)$
Avg case : $O(n \log n)$
Worst case : $O(n \log n)$

Bubble Sort

Best case : $O(n)$
Avg. case : $O(n^2)$
Worst case : $O(n^2)$

Complexity Growth Illustration :-



$1 < \log n < \text{root}(n) < n < n^2 < n^3 \ldots$
$< 2^n < 3^n \ldots < \ldots < n! < n^n$

(a) $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$
$< O(n^3) < \ldots < O(2^n) < O(3^n) < \ldots < O(n!) < O(n^n)$

ex: ① $f_1 = 10^n$, $f_2 = n^{\log n}$, $f_3 = n^{\sqrt{n}}$

Apply log

$\log 10^n$	$\log n \log n$	$\sqrt{n} \log n$
$n \log 10$	$(\log n)^2$	
$O(n)$	$O(\log n)$	$O(\sqrt{n})$

$\therefore f_1 > f_3 > f_2$.

② $10, \sqrt{n}, n, \log n, \frac{100}{n}$.

$\Rightarrow \frac{100}{n} < 10 < \log n < \sqrt{n} < n$.

Algorithm :- Binary Search

Input : list/array of numbers and search key.
Output : index of search key

```
Start
Binary_Search (A[], key)
    i = 0, low = 0, high = length(A) - 1
    while (low <= high)
    {
      mid = (low + high) / 2
      if( a[mid] == key)
        return mid
      else if (a[mid] > key)
          high = mid - 1
      else
          low = mid + 1
End.
```

Best case $O(1)$
Worst case $O(\log n)$
Average case $O(\log n)$
↳ (Calc. using formula).

1	n	n
2	$n/2$	$n/2$
3	$n/4$	$n/2^2$
4	$n/8$	$n/2^3$
k	$\frac{n}{2^{k-1}} = 1$	

$n = 2^{k-1}$
$\log n = k-1$
$k = \log n + 1$
$\Rightarrow O(\log n)$. $\Rightarrow$ W.C

Recurrence relation (equation):-

When an algorithm contain a recursive call to itself then its time complexity can be described by recurrence relation.

ex: Merge sort using recursive calls

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

① void fun(int n)

```
if n>0        = 1
  print n     = 1
  fun(n-1)    = T(n-1)   → since it is recursive fn.
```

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

② A(n)

```
if n>0              = 1
  for i=0..n i++    = n+1
    print n         = n
  A(n-1)            = T(n-1)
```

o/p:
55555
4444
333
22
1

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

③ fib(n)

```
if(n<=1)                    = 1
  return n                  = 1
else
  return fib(n-1)+fib(n-2)  = T(n-1)+T(n-2)
```

$\therefore T(n) = T(n-1) + T(n-2) + 2 \rightarrow$ W.C
$T(n) = 1$ when $n = 0 \rightarrow$ B.C

How to solve Recurrence relation? - (3 methods).

① Substitution method (forward and backward substitution)

Backward substitution:

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

$T(n) = T(n-1) + 1 \rightarrow$ W.C
$T(n) = T(n-2) + 1 + 1$
$T(n) = T(n-3) + 1 + 1 + 1$
$\boxed{T(n) = T(n-k) + k}$

$n - k = 0 \Rightarrow T(n) = T(0) + n$
(stopping condition) $= 1 + n$
$\therefore O(n)$

Forward Substitution:

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

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

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

$= T(n-k) + (n-k+1) + \dots + n$

$n-k=0 \Rightarrow T(0) + 1 + 2 + 3 + \dots + n$

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

$\therefore O(n^2)$

ex: ①.

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

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

$T(n) = 2\left[2T(n/4) + n/2\right] + n$

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

$\vdots$

$2^k T\left(\frac{n}{2^k}\right) + kn$

$\frac{n}{2^k} = 1 \Rightarrow 2^k T(1) + kn$

$\Rightarrow k = \log_2 n \quad = 2^{\log_2 n} + n\log n$

$= O(n\log n)$

②.

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

$T(n) = \sqrt{2}\,T(n/2) + \sqrt{n}$

$= \sqrt{2}\left[\sqrt{2}\,T(n/4) + \sqrt{\frac{n}{2}}\right] + \sqrt{n}$

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

$\vdots$

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

$\frac{n}{2^k} = 1 \Rightarrow \sqrt{2}^{\log n}\,T(1) + \sqrt{n}\log n$

$k = \log n \quad = \sqrt{n} + \sqrt{n}\log n$

$\therefore O(\sqrt{n}\log n)$.

Q. What explains best the T.C of binary search.

a) $O(n)$ b) $O(n^2)$ c) $O(n^2)$ d) None.

O means not exactly T.C it is upper bound. So $O(n)$ is upper bound of $O(\log n)$ → T.C of Binary Search. Hence $O(n)$ best explains T.C of binary search.

$\theta(n)$ gives exact T.C but $O(n)$ gives upper bound of T.C only not exact.

ex③ Substitution Method:

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

$T(n) = T(\sqrt{n}) + 1$

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

$T(n) = T(n^{1/4}) + 2$

$= T(n^{1/8}) + 3$

$\vdots$

$T(n) = T(n^{\frac{1}{2^k}}) + k \quad = T(1) + \log(\log n)$

$n^{1/2^k} = 2 \Rightarrow \frac{1}{2^k}\log n = 1 \Rightarrow 2^k = \log n$

$\Rightarrow k = \log(\log n)$

$\therefore O(\log(\log n))$

(2). Recurrence Tree Method.

Step 1: Draw a recursion tree based on given R.R.

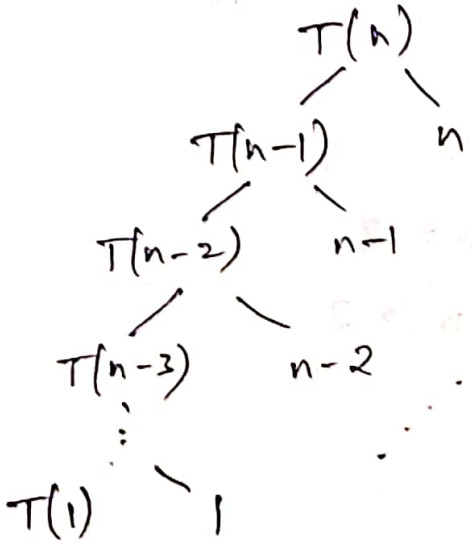
Step 2: Determine — • Cost of each level
• Total no. of levels in recursion tree.
Adds constant so not necessary { • No. of nodes in last level
• Cost of last level.

Step 3: Add cost of all levels of recursion tree and simplify the expression so obtained in terms of asymptotic notation.

ex: ①. $T(n) = \begin{cases} 1 & , n = 1 \\ T(n-1) + n & , n > 1 \end{cases}$

$T(n) = \boxed{T(n-1)} + \boxed{n}$

↓ operation ↓ cost of operation.



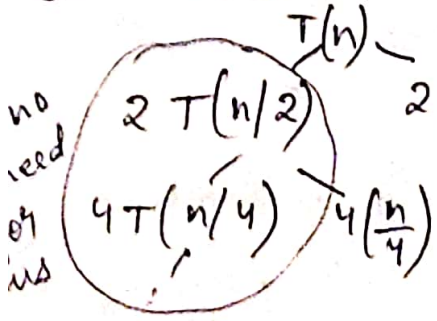
$T(1) + 2$

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

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

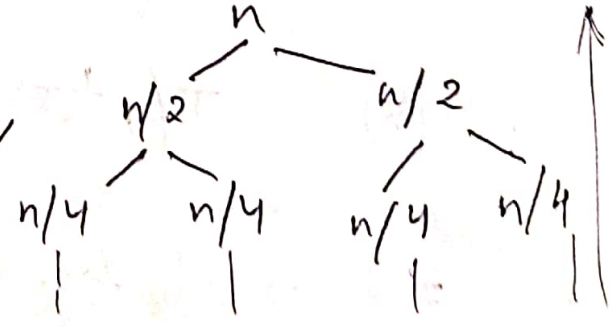
$\therefore O(n^2)$

② $T(n) = 2T(n/2) + n$



no need of ins

→ So,



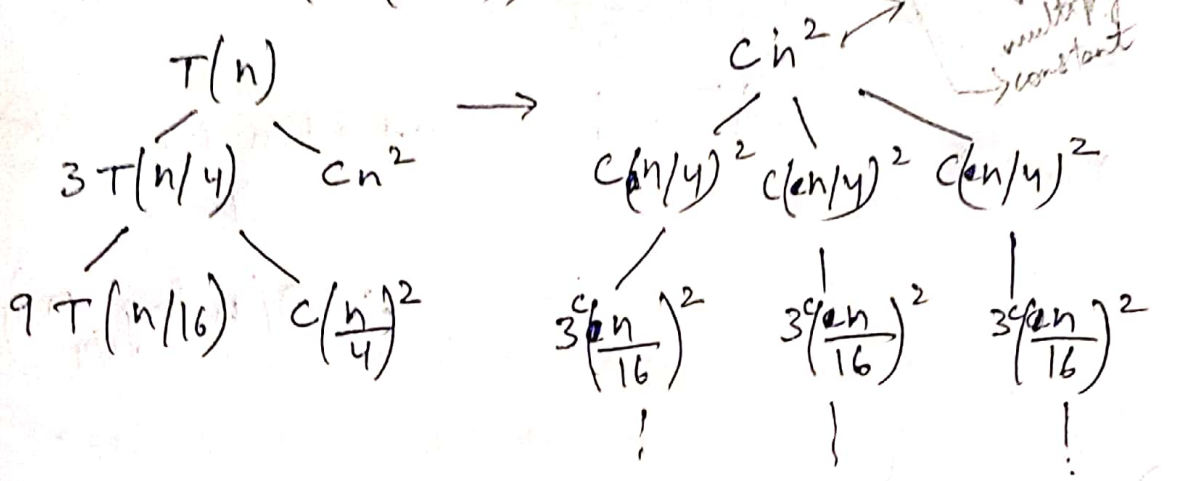
If k is the height, then

$$\frac{n}{2^k} = 1 \Rightarrow K = \log n$$

$$\therefore T(1) + kn = 1 + n\log n$$

$$\therefore O(n \log n)$$

③ $T(n) = 3(T(n/4)) + cn^2$



$$= cn^2 + 3\left(\frac{cn}{4}\right)^2 + 9\left(\frac{cn}{16}\right)^2 + \dots$$

$$= cn^2\left(1 + \frac{3}{16} + \frac{9}{256} + \dots\right)$$

It is G.P, $a = 1$, $r = \frac{3}{16}$ $\left[\text{sum} = \frac{a}{1-r}\right]$

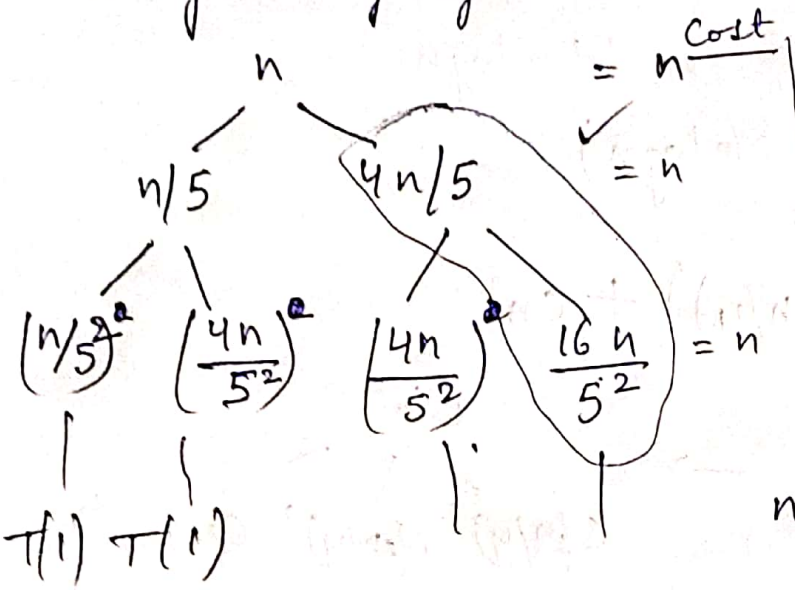
$$= cn^2\left(\frac{1}{1-\frac{3}{16}}\right)$$

$$= O(n^2)$$

If there is series directly find sum of series. otherwise take height as K and then solve.

④ $T(n) = T(n/5) + T\left(\frac{4n}{5}\right) + n$.

Taking directly right tree we have,



$\left(\frac{4}{5}\right)^k n = 1$

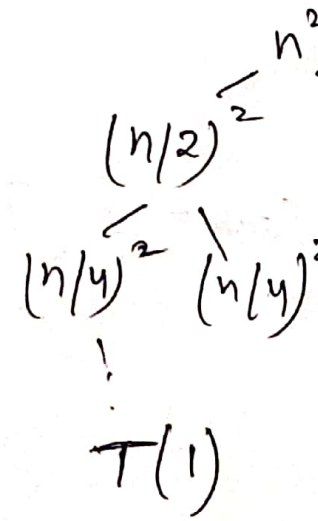
$k = \log_{5/4} n$

$n + n + n + \ldots$ (k)

nk

$\therefore O(n \log_{5/4} n)$.

⑤ $T(n) = 2T(n/2) + n^2$



$= n^2$

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

$= 4\left(\frac{n}{4}\right)^2$ (G.P)

$= n^2\left(1 + \frac{2}{2^2} + \frac{4}{4^2} + \ldots\right)$

$= O(n^2)$.

(3) · Master Method ·

- Master method for reducing (subtraction function)

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

In recurrence relation $T(n)$

$T(n) = T(n-1) + 1$	$O(n)$
$T(n) = T(n-1) + n$	$O(n^2)$
$T(n) = T(n-1) + \log n$	$O(n \log n)$
$T(n) = 2T(n-1) + 1$	$O(2^n)$
$T(n) = 2T(n-1) + n$	$O(n 2^n)$.
$T(n) = 3T(n-1) + 1$	$O(3^n)$.
⋮	

from these we can conclude,

$$T(n) = aT(n-b) + f(n)$$

$a>0,\ b>0,\ f(n) = O(n^k)$.

if $a=1$ $\quad T(n) = O(n \times f(n))$

$a>1$ $\quad T(n) = O(a^n f(n))$

$a<1$ $\quad T(n) = O(f(n))$.

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

then $= O(2^n \cdot n \log n)$ $[\because a>1]$.

$= T(\quad) \quad \alpha \; T(\quad) + n \log n$

then $= O(2^n \log n \log n)$ [$\because a > 1$].

ex: $T(n) = T(\sqrt{n} - 2) + n \log n$

Take $\sqrt{n} = k$

$\Rightarrow \quad = T(k-2) + k^2 \log k^2$

Now use master method, $a = 1$

$= O(k \cdot k^2 \log k^2)$

$= O(k^3 \log k^2) = O(n^{3/2} \log n)$.

- Master method/theorem for dividing function.

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

where $a >= 1$, $b > 1$, $f(n) = O(n^k \log^p n)$

If the recurrence relation is of the above form, then,

Find $\log_b^a$ and k.

Case 1:

$\log_b^a > k$

$O(n^{\log_b^a})$

Case 2:

$\log_b^a = k$

1) $p > -1$, $O(n^k \log^{p+1} n)$
2) $p = -1$, $O(n^k \log\log n)$
3) $p < -1$, $O(n^k)$

Case 3:

$\log_b^a < k$

1) if $p >= 0$ then $O(n^k \log^p n)$
2) if $p <= 0$ then $O(n^k)$.

ex:

① $T(n) = 2T\left(\frac{n}{2}\right) + 1$, n^0

$\log_2^2 = 1 > k = 0$.

$\therefore O(n^1) = O(n)$.

② $T(n) = 4\left(T\left(\frac{n}{2}\right)\right) + 1$

$\log_2^4 = 2 > k = 0$

$\therefore O(n^2)$

③ $T(n) = 8T\left(\frac{n}{2}\right) + n$

$\log_2^{2^3} = 3 > k = 1$

$\therefore O(n^3)$

④ $T(n) = 9T\left(\frac{n}{3}\right) + 1$

$\log_b^a = \log_3^{3^2} = 2 > k = 0$

$\therefore O(n^2)$.

⑤ $T(n) = 8T\left(\frac{n}{2}\right) + n^3 \log^0 n$

$\log_2^8 = 3 = k = 3$, $p = 0$

$\therefore O(n^3 \log n)$

⑥ $T(n) = 2T\left(\frac{n}{2}\right) + n \log^2 n$

$\log_2^2 = 1 = k = 1$, $p = 0$

$\therefore O(n \log n)$

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

$\log_2^4 = 2 = k = 2$, $p = 1$

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

⑧ $T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^2}{\log n}$

$\log_2^4 = 2 = k = 2$, $p = -1$

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

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

$\log_2^2 = 1 < k = 2$, $p = 1$

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

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

$\log_2^2 = 1 < k = 2$, $p = -2$

$\therefore \not{O}(n^2)$.

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

$\log_2^2 = 1 < k = 2, \; p = 1$

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

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

$\log_2^2 = 1 < k = 2, \; p = -2$

$\therefore \not\in O(n^2)$.

Unit-2.

Design Techniques :-

1. Divide and Conquer
2. Greedy Method
3. Dynamic Programming
4. Back Tracking
5. Branch and bound.

- Max-Min PBM
- Recursive Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix Multiplication

How to find Max of a list/array?

10	21	9	19	43	101	2	99	45	1012

Input : Array of elements
Output : Index of Max Element / or the max element itself

Algo:
Take a temp variable and assign the first element of array.
Now scan the list against the temp variable.
if (temp < A[i]) ↖ loop.
temp = A[i].
in the last the max will be in the temp.

complexity of algorithm = $n-1 = O(n)$.

What is the time complexity for finding min of list.
$n-1$
$O(n)$.

What is the complexity of an algorithm that finds max-min of list.
$2n-2$ ← can we reduce these steps/ complexity?
$O(n)$

For this, suppose there are x_1, x_2, x_3, x_4.

L: $x_1 < x_2$ → Min L, Max L
R: $x_3 < x_4$ → Min R, Max R

So $T(n) = 2T(n/2) + 2$ → again comparing L & R twice.
↓ Min & max; ↓ array divided into 2

$= O(n)$

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

$= 2^{k+1}$

$= \frac{3n}{2} - 2$ — ?

Now compare $2n-2 > \frac{3n}{2} - 2$

$O(n) = O(n)$

Hence the no. of operations are reduced using the divide & conquer.

Recursive Algorithm:-

Input : array of element
Output : Max of array.

→ This is not divide & conquer algo. Just a R.Algo.

R.Algo:

```
Rmax (A, n)
  if (n==1)
    return A[0]
  else
    return max(A(n-1), Rmax(A, n-1))
```

Divide and Conquer:-

- Divide a large pbm into subproblems.
- Solve the subproblems recursively.
- Combine the solutions of subproblems to get the solution for the original pbm.
  - The nature of subproblem must be same as large pbm. You can't convert the pbm into another pbm.

Recursive Binary Search algorithm:-

Input : sorted array, key.
Output : index of key.

```
Rbinary Search (low, high, key)
    if low == high
      if A[low] == key
        then return low
      else
        return -1 .
    else
      mid = (low + high)/2
      if (a[mid] == key)
        return mid
      elseif (a[mid] > key)
        Rbinarysearch (low, mid-1, key)
      else
        Rbinarysearch (mid+1, high, key) .
```

Bubble Sort:-

```
for i 0--n-1 i++
  for j i--n-1 j++
    if a[i] < a[j]
      then swap(a[i], a[j])
```

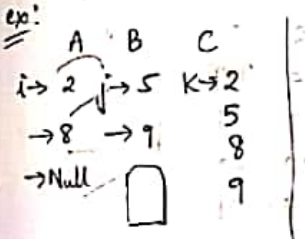
Refer in-order sorting, stable & unstable sorting techniques w.r.t. GATE.

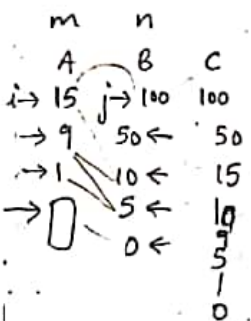
Merge Sort:-

For sorting, merge sort uses merging procedure.

Merging procedure — is the process of combining two sorted list into a single list

ex:





For merge sort

$O(n \log n)$

↓ merging procedure, ↓ dividing

time complexity of merging $O(m+n)$.

2-way merging procedure algorithm:

Input : 2 sorted lists.
Output : 1 sorted list.

Algorithm:

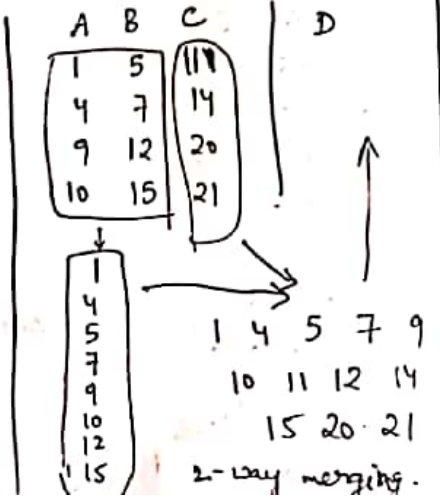
```
merge(A, B, m, n)
{
  while (i<=m && j<=n)
  { if a[i] < b[j]
      c[k++] = A[i++]
    else
      c[k] = b[j]
      k++   j++
  }
  for(; i<=m; i++)
    c[k++] = a[i++]
  for(; j<=n; j++)
    c[k++] = b[j++]
```

* 3-way merging and M-way merging for M-lists is very complicated & hence we use 2-way merging procedure.

ex:

A	B	C	D
1✓	5✓	11	1 ←l
→4✓	→7	14←	4
→9✓	→12	20←	5
→10	→15	21	7
→	→		9
i	j	k	10
			11
			12
			14
			15
			20
			21

3-way merging (complicated.



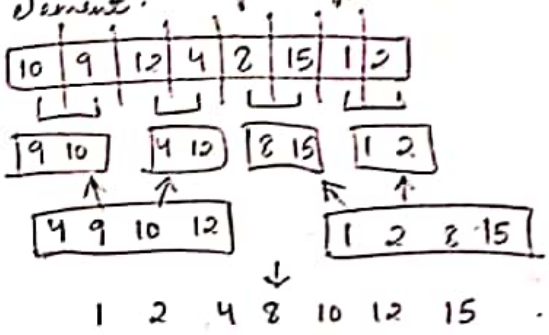
- Two way merge sort → Bearable Merge.
- Merge sort → Permissive Algo.

2-way merging procedure:-

We need 2 lists. To use follow a assumption.

Assumption:
Assume each element of array/list as list of one element.

ex:



But the implementation is not at all easy.

Input : Unsorted list/array
Output : Sorted list/array

RMerge sort Algorithm:

```
ms(l,h)
{ if(l<h)
  {
    mid = (l+h)/2
    ms(l, mid)
    ms(mid+1, high)
    merge(l, mid, high) → have to write merge algo also
  }
}
```

Quick sort :-

* Elements arrange themselves & hence is quicker than other algorithms.
* Best case $O(n \log n)$ same goes for avg case.
* Worst case is $O(n^2)$.
* The main logic is — in quick sort, we select an element in the right place & then sort other elements.

ex: Let's see the procedure through an example.

10	16	8	12	15	6	3	9	5
l	i							h
								j

Larger than pivot → ← smaller than pivot and keep swapping i & j

pivot = a[l]

10	5	8	12	15	6	3	9	16
			i				j	

10	5	8	9	15	6	3	12	16
				i		j		

10	5	8	9	3	6	15	12	16
					j	i		

Once i & j cross each other swap j & pivot

6	5	8	9	3	10	15	12	16

< >

pivot in its right place.

i.e elements to right are greater & elements to left are lesser.

⇒ Partition Algorithm :-

```
partition(l,h)
{ p = a[l]
  i = l , j = h
  while(i<j){
  do
  { i++
  }while(a[i]≤p)
  do
  { j++
  }while(a[j]≥p)
  swap[a[i],a[j]]
  }
  swap(a[p],a[j])
} return j
```

R. Quicksort Algorithm :- → have to write partition algo. also

```
quicksort(l,h)
if(l<h)
{ j = partition(l,h)
  quicksort(l,j)
  quicksort(j+1,h)
}
```

Complexity :

Best Case : $T(n) = 2T(n/2) + n$ [in best case, pivot is in middle & we compare with all n elements]

$\therefore O(n \log n)$

Worst Case: When elements/list are already sorted.
(or) when 1st or last element is chosen as pivot.

Suppose 1 2 4 5 6 9 10 elements.

$n \quad n-1 \quad n-2 \quad n-3 \quad \ldots + 1$ comparisons

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

$\therefore O(n^2)$

[we make comparisons or efforts to place the element in its own place only i.e right place itself].

How to avoid the Worst case in Quick Sort?

1. Always choose the middle element as a pivot.
2. Choose a random element as pivot (chance of W.C).

Even when the list is sorted, we perform sorting because humans can recognize sorted lists but computer machines cannot.

Strassen's Matrix Multiplication :-

A_n*n B_n*n C_n*n — Classical method of Mat-mul.

```
C_ij = A_iK * B_Kj
for i 0..n i++
  for j 0..n j++
    C_ij = 0
    for K 0..n k++
      C_ij = C_ij + A_ik * B_kj
```

Time Complexity $O(n^3)$

Worst case: When elements/list are already sorted. (or) when 1st or last element is chosen as pivot.

Suppose 1 2 4 5 6 9 10 elements

$n \quad n-1 \quad n-2 \quad n-3 \quad \ldots + 1$ comparisons

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

$\therefore O(n^2)$

[we make comparisons or efforts to place the element in its own place only i.e right place itself].

How to avoid the Worst case in Quick Sort?

1. Always choose the middle element as a pivot.
2. Choose a random element as pivot (chance of W.C).

Even when the list is sorted, we perform sorting because humans can recognize sorted lists but computer machines cannot.

Strassen's Matrix Multiplication:-

— Classical method of Mat-mul.

```
A_n*n  B_n*n
C_n*n
C_ij = A_iK * B_Kj
for i 0..n i++
  for j 0..n j++
    C_ij = 0
    for K 0..n k++
      C_ij = C_ij + A_ik * B_kj
```

Time Complexity $O(n^3)$.

This is the classical method for Matrix mult.

— Divide and Conquer method for Mat. Mul.

$X = [\ ]_{2^i \times 2^i}$

The size of largest matrix will be 2^i, only then we will be able to divide into 2 and so on.

ex:

1 2 | 2 1	1 2 | 2 1	6 12 12 6
1 2 | 2 1	1 2 | 2 1	6 12 12 6
1 2 | 2 1	1 2 | 2 1	6 12 12 6
1 2 | 2 1	1 2 | 2 1	6 12 12 6
A	B	C

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}_{2^i \times 2^i} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}_{2^i \times 2^i}$$

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

$$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$$
$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$$
$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$$
$$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$$

Now verify:

$$\begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix}\begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix}\begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 6 \\ 3 & 6 \end{bmatrix} + \begin{bmatrix} 3 & 6 \\ 3 & 6 \end{bmatrix}, \begin{bmatrix} 6 & 3 \\ 6 & 3 \end{bmatrix} + \begin{bmatrix} 6 & 3 \\ 6 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 6 \\ 3 & 6 \end{bmatrix} + \begin{bmatrix} 3 & 6 \\ 3 & 6 \end{bmatrix}, \begin{bmatrix} 6 & 3 \\ 6 & 3 \end{bmatrix} + \begin{bmatrix} 6 & 3 \\ 6 & 3 \end{bmatrix}$$

$$= \begin{bmatrix} 6 & 12 & 12 & 6 \\ 6 & 12 & 12 & 6 \\ 6 & 12 & 12 & 6 \\ 6 & 12 & 12 & 6 \end{bmatrix}$$

In classical method,
1 element 4 mul 3 addition.
For 16 elements, 16*4 mul 16*3 addition.
64 ~~mul~~ M 48 A

In divide & conquer,
1 element 2 M and 1 A
4*2 M 4*1 A $\begin{bmatrix} 3 & 6 \\ 3 & 6 \end{bmatrix}$.

8*4*2 M 8*4 + 4*4
8 such $\begin{bmatrix} 3 & 6 \\ 3 & 6 \end{bmatrix}$
64 M 48 A

$\begin{bmatrix} 3 & 6 \\ 3 & 6 \end{bmatrix} + \begin{bmatrix} 3 & 6 \\ 3 & 6 \end{bmatrix}$
2 + 2 = 4
4 such mat. of 4*4

no change.

Let's do algorithmically,

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

Apply Master method, $\log_b^a = 3 > K = 2$

$$\therefore O(n^{\log_b^a}) = O(n^3)$$

Algorithm:- (Divide & conquer).

MM(A, B, n)
{
if n == 2
{ $c_{11} = a_{11}b_{11} + a_{12}b_{21}$ → elements
$c_{12} = a_{11}b_{12} + a_{12}b_{22}$
$c_{21} = a_{21}b_{11} + a_{22}b_{21}$
$c_{22} = a_{21}b_{12} + a_{22}b_{22}$
}
if (n > 2)
{ mid = n/2 → matrices
MM(A11, B11, n/2) + MM(A12, B21, n/2)
MM(A11, B12, n/2) + MM(A12, B22, n/2)
MM(A21, B11, n/2) + MM(A22, B21, n/2)
MM(A21, B12, n/2) + MM(A22, B22, n/2)
}

$n^2 + n^2 = 2n^2$

and hence we write $T(n) = 8T(n/2) + n^2$

Strassen proposed a few constants from research,

P = (A11 + A22) * (B11 + B22)
q = (A21 + A22) * B11
r = A11 * (B12 − B22)
s = A22 * (B21 − B11)
t = (A11 + A12) * B22
u = (A21 − A11) * (B11 + B12)
v = (A12 − A22) * (B21 + B22)

Strassen's Mat-mul Algo:-
if (n <= 2)
then A * B
else
{ find p q r s t u v
C11 = p + s − t + v
C12 = r + t
C21 = q + s
C22 = p − q + r + u
}

$$T(n) = 7T(n/2) + n^2$$

By Master theorem, $O(n^{\log_2 7})$

$= O(n^{2.8073})$

$= O(n^{2.81})$.

So classical & divide & conquer $O(n^3)$.

But Strassen's Mat·mul $O(n^{2.81})$ → T.C reduced.

But nobody uses Strassen's mat·mul due to the constants.

* Whereas for Sparse matrices where, almost more than half elements are 0's, the complexity for mat·mult· will be $O(n^2)$.

Amortized analysis :-

└ Amortization

Depreciation.

Whenever we need to reflect the cost of an asset in Balance Sheet we divide the cost into its useful life. (intangible assests)

loans

An algorithm is performing sequence of operations(s)

Some operations are very faster.

but few operation(s) or occasional operations are slow.

| | | | n | | | | | n

O(n)

1 + 1 + + 1 + 1 + n + 1 + + + + 1 + 1 / total operations

if one operation affecting the cost of on another operations in the sequence then we have to use amortized analysis instead of worst case analysis.

ex: Like in a dynamic array where the array size keeps changing when we add elements.

3 ways to do Amortized analysis:

1. Aggregate method ✓ (We discuss only this)
2. Accounting method
3. Potential method

ex: Consider we need to do N insertion in dynamic array.

Worst case complexity/analysis:

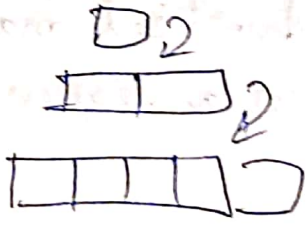
all N insertion is worst time.
i.e each element inserted at the end.
For 1 element W.C. complexity is $O(N)$.
So for N elements it is $O(N^2)$.

Amortized analysis (Aggregate method):-

Same array can't be doubled-(∵ array needs continuous memory allocation)
Need to create & new array with double size

ex:

Elements	Cost of insertion
1	1
1	2
1	3
1	1
1	5
1	1
1	1
1	1
1	9
1	



$1+1+3+1+5+1+1+1+9$

$1+1 \quad 1+2 \quad 1+4 \quad 1+8$

$1+1+1+\ldots+1+2+4+8+\ldots 2n$

$n+$

$\hookrightarrow \frac{a(r^n-1)}{r-1} = \frac{2^{\log_2 n}-1}{1}$

$= n-1$

So $n + n - 1$

$= 2n - 1$

Removing constants $= n$.

So cost for 1 insertion $= \frac{n}{n} = 1$.

∴ The amortized cost of insertion on dynamic array is $2n-1$

amortized cost of 1 insertion is $O(1)$.

Aggregate method says:

cost of n operations = cost(fast operations) + cost(slow operations)

total operations (n)

* Amortized analysis is also Worst case analysis but for a sequence of operations.

total operations (n)

* Amortized analysis is also Worst case analysis but for a sequence of operations.

Probabilistic Analysis of Algorithm:-

We make use of probability theory to analyse the running time of algo. → distribution (PDF/PMF, Expected value/mean, S.D, variance).

We use it to analyze avg. case complexity.

↳ To find exact complexity because not all (Why) algo's go till the extent of using W.C comp.

↳ (How)? Using Gaussian distribution

We use this distribution, because it tells us how frequently we get the data

i.e. we need to understand the distribution of input or atleast make some assumption about it

↓ like uniform random generator

ex: Max of list:

hiring pbm

input: list

output: index of max / max itself

Algo:

```
max = element 1
for i 1 to n
  if max < element i
  then max = element i
```

⟶ time complexity = cost of comparison (cc) + cost of assignment (ca)

= cc + ca

= n(cc + ca) Worst case complexity

= n*cc + ca Best case complexity

Since we can't change comparisons, n*cc in all 3 cases

We take only assignment, n*ca in W.C

ca in B.C

What about avg case?

Avg. value = $\frac{\text{mean}}{\text{expected value}}$

Avg. assignment = $ca * \log n$

$= O(\log n)$ — Avg. comp. for assignment only.

Total avg. comp = $n + \log n = O(n)$.

→ How many assigment in avg-case

x_1, x_2, x_3, x_4 (4

1A, 2A, 3A, 4A

1 element $P(P = Max) = 1$

$P(P = Max) = \frac{1}{2}$

$P(P = Max) = \frac{1}{4}$

$= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$

This is H.P series so $O(\log n)$?

$P(P = max) = \frac{1}{4}$

$= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$

This is H.P series so $O(\log n)$ - ?

Max of list.

input : list
output : index of max / max itself

Algo:

```
max = element 1
for i 1 to n
    if max < element i
        then max = element i
```

Hiring Process

input : n candidates
output : office assistant

Algo:

```
assistant(n)
best = 0
for i 1 to n
    if candidate[i] > best
        best = candidate[i]
    hire i
```

* Here assignment cost does not matter.

* Comparison cost does not matter (interviewing

* Comparison cost matters.

cost).
* Assignment cost matters. (hiring cost).
* Hiring cost = pay salary to previous assistant and pay commision to agency.

Salary by skill set or capability of person.

* Worst case not cost effective (i.e sending persons in inc. order of capability)
Best case you do not know (you have to interview all then only you know best).
So go for the average case.
* In avg. case, the company randomly assigns the numbers for candidates & shuffles them & then interviews & hires.

Greedy Method :-

Q① Cashier in bank
you have to return $6 to customer
1$ 5$ 7$ coins available
you will give the change — (Minimisation).
5$ + 1$ (optimal solution)
$ + $ + $ + $ + $ + $ (non-optimal solution).

② there are 5 places to visit.
you want to go from place 1 to 5
1 → 2 or 3
2 → 3 or 5
3 → 1 or 4
4 → 2 or 5

(minimize the steps)

Humans sol^n : 1 → 2 → 5

Greedy method sol^n : 1 → 3 → 4 → 5 (it is not optimal)

∴ Greedy does not guarantee optimal solution, it will only give optimum solution when local optima and global optima is same.

* Greedy method is used for maximation or minimisation problems.
  - → objective function
  - → constraints.
* Out of all the algorithmic design techniques, greedy is simplest and straight forward.
* It finds a solution to the pbm in steps. And in each step it chooses a subsolution that gives a immediate benefit.

→ greedy methods ← (then don't have to interview many.)
filter
10 th class less than 60% remove them
↳ This only gives immediate benefit.
No guarantee that it's the best.

* In greedy method decision is taken based on the current available information without worrying about the effect of the current decision in future.
* Always makes the choice that seems to be the best at that moment.
* It makes locally optimal choice in the hope that this choice will lead to the globally optimal solution.
* Applicability of greedy method — wherever the local optimal (is global optimal) ↓ to global optimal lead the greedy is the solution.

Note: the greedy have only one shot to solve the pbm that means greedy method cannot go back (back track) in reverse direction.

Elements of the greedy approach:-

* Candidate set = is set from where solutions has to be considered / A solution is created from this set
* Selection function = chooses the best candidate to be added to the solution.
* Feasibility function = whether the selected candidate is satisfying constraint / whether a candidate can be used to contribute to the solution.
* Objective function = this defines the objective of the optimization ~~set~~ pbm.
* Solution = it represents complete or partial sol^n

General Method :-

```
Greedy (C, n)
{
  solution = 0
  for i 1 to n
  {
    s = select(C)
    if (feasible (solution, s))
      then solution = union (solution, s)
  }
}
```

Knapsack Problem (Transporter problem) :-

You have been given a bag with its capacity.

Objects	A	B	C	D	E	F	G	H
Profits	10	5	15	7	6	18	3	45
Weights	2	3	5	7	1	4	1	16
Profit/weight	5	1.3	3	1	6	4.5	3	

(selection metric: profit per kg)

Objective: To maximise profits by charging more.

0/1 knapsack pbm

- Objects are indivisible
- Greedy does not give optimal solution (We use dynamic sol^n).

Fractional Knapsack pbm:

- Objects are divisible.
- Greedy will give the optimal solution.

(Capacity of knapsack is 15 kg)

Candidate set is (A B C D E F G)

Selection function max (P/W).

Feasibility function:

$$\sum W_{c_i} < 15$$

Objective function:

$$\max \sum W_i \times P_i$$

(Profit maximisation).

Object	Weight	Constraints	Profit
E	1	15-1 = 14	6
A	2	14-2 = 12	10
F	4	12-4 = 8	18
C	5	8-5 = 3	15
G	1	3-1 = 2	3
B	2	2-2 = 0	$\frac{2}{3}(5)$
			55.33

This is the maximum profit which can be gained.

* For fractional knapsack pbm, greedy method gives the optimal sol^n.

0/1 sol^n:

F	18	4	= 4
C	15	5	= 9
A	10	2	= 11
	7	7	= X
E	6	1	= 12
B	5	3	= 15

Selection fn: Max (P_i)

Objective fn: $\max \sum P_i$

But the greedy might not give the optimal sol^n

Take another example:

O:	A	B	C	D
P:	24	18	18	10
W:	24	10	10	7

Capacity = 25

0/1 Knapsack.

A 24 24

B C

18 + 18 = 36 > 24

10 + 10 = 20 ≤ 25

So Greedy can not give optimal sol^n.

Job sequencing with deadline :-

example ①

Jobs	J1	J2	J3	J4	J5
Profit	20	15	10	5	1
deadlines	2	2	1	3	3

↓

is not completion time (i.e job J1 has to be completed in 2 hrs but it doesn't mean that J1 takes 2 hrs to complete)

Assumption that each job needs same time to complete and it is 1 hr.

0 — 1 — 2 — 3 → (Max 3 hrs they can wait)

J_2 J_1 J_4

20
15
5
40 → Max profit

Choose the jobs by profit and arrange them by deadlines if they can wait or not!

Consider another example (2).

Jobs	:	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profit	:	35	30	25	20	15	12	20
deadlines	:	2	2	2	1	3	1	3

0 — 1 — 2 — 3

J_2 J_1 J_7

35
30
20
85 → Max profit which can be earned.

Hence greedy is the only soln to this (because of our assumption that each job takes same time).

For example (1)

Job to be considered next	Slot assigned	Slot available	Solution	Profit
—	—	[0-1] [1-2] [2-3]	ϕ	zero
J1	[1-2]	[0-1] [2-3]	J1	20
J2	[0-1]	[2-3]	J1 J2	15
J3	reject	[2-3]	J1 J2	—
J4	[2-3]	ϕ	J_1 J_2 J_4	5

40

Note: If each job does not take the same amount of time to complete then greedy is not the solution.

Choose the jobs by profit and arrange them by deadlines if they can wait or not!

Consider another example ②.

Jobs	:	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profit	:	35	30	25	20	15	12	20
deadlines	:	2	2	2	1	3	1	3

0 — 1 — 2 — 3

J_2 J_1 J_7

35
30
20
85 → Max profit which can be earned.

Hence greedy is the only soln to this (because of our assumption that each job takes same time).

For example ①

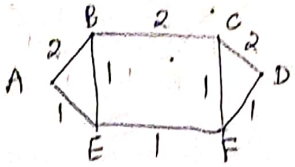
Job to be considered next	Slot assigning	Slot available	Solution	Profit
—	—	[0-1][1-2] [2-3]	∅	zero
J1	[1-2]	[0-1][2-3]	J1	20
J2	[0-1]	[2-3]	J1 J2	15
J3	reject	[2-3]	J1 J2	—
J4	[2-3]	∅	J1 J2 J4	5
				40

Note: If each job does not take the same amount of time to complete then greedy is not the solution.

Minimum (Cost) Spanning Tree :-

Spanning visit all vertices once with no cycle in it and the cost of visiting should be minimum.

ex ①:



ABEFCD	2 1 1 1 2	= 7
AEBCFD	1 1 2 1 1	= 6 ✓ minimum
ABCDFE	2 2 2 1 1	= 8

Greedy Method is applicable (Minimisation).
For this we have 2 algo.s Prim's and Kruskal's.

Prim's Algorithm :-

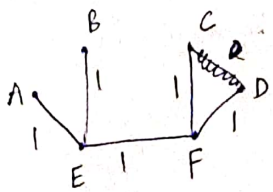
1) Choose a starting vertex and remaining vertex fringe list.
2) For each vertex in the fringe list do the following.
   a) select an edge 'A' connecting a fringe vertex with minimum weight.
   b) add the edge and the vertex in the MST.
3) Exit.

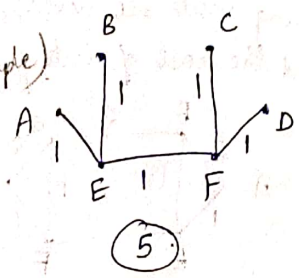
Kruskal's Algorithm :-

1) Sort all the edges in non-decreasing order of weight.
2) Pick the edge with smallest weight (with previous subgraph)
   a) check whether it is forming cycle or not.
      if not add it to the MST.
3) Repeat step 2 until there are (V-1) edges in spanning tree.
4) Exit.

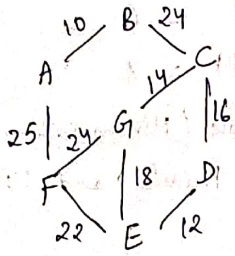
Kruskal's

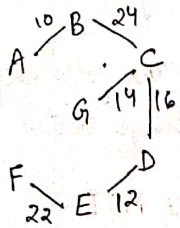
ex (1) : Prim's (check this example)



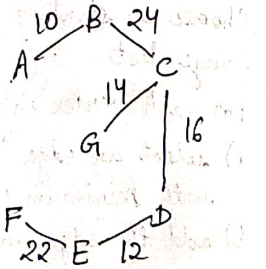


ex 2:





Prim's (starting from G)



Kruskal's (starting from AB edge 10 wt)

Prim's Algo complexity :-

Implement prim's algo. with binary heap ($\log n$).

$\therefore O(E \log V)$.

Kruskal's Algo complexity :-

Sorting $n \log n$ $O(E \log E)$

union find Algo $O(n \log V)$

$\hookrightarrow O(E \log E + E \log V)$

$O(E \log E)$

Disjoint set data structure :-

It is a data structure that keep tracks of a set of elements partitioned into a no. of disjoint sets.

We can perform two operations on it.

1. Find
2. Union

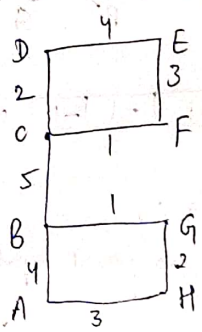
* Find :— Given an element Find will find to which subset the element belongs to.

* Union :— if two elements does not belong to same subset then union them and create new subset.

Union Find Algo :-

It uses Disjoint set data structure to find cycle in a subgraph / graph.

It is limited to undirected graph.



$U = \{A, B, C, D, E, F, G, H\}$

e(D, C) → $S_1 = \{D, C\}$ (so delete this)

e(C, B) → $S_4 = \{C, D, B\}$

e(E, F) → $S_2 = \{E, F\}$

e(C, F) → $S_5 = \{C, D, B, E, F\}$

e(G, H) → $S_3 = \{G, H\}$

e(D, E) ↑ belong to same set. So forms cycle.

Implementation:-

DSDS →

A	B	C	D	E	F	G	H
-1	-1	-1	-1	-1	-1	-1	-1

e(A, B)

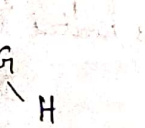
A	B	C	D	E	F	G	H
-2	0	-1	-1	-1	-1	-1	-1

A is parent, B is child. → -2 says no. of elements and 0 says that element in 0th index is its parent.

e(G, H)

A	B	C	D	E	F	G	H
-2	0	-1	-1	-1	-1	-2	6

e(A, H) = {0, 6} ⇒ {A, B, G, H}.

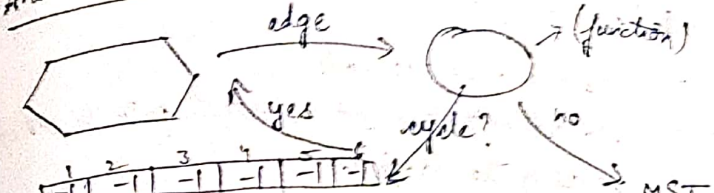


-4	0	-1	-1	-1	-1	0	6

e(B, G) = {0, 0} = {A, A}.

same parent, so forms cycle.

Another example:



1	2	3	4	5	6
-1	-1	-1	-1	-1	-1

e(6, 5)

1	2	3	4	5	6
-1	-1	-1	-1	5	-2

e(2, 3)

1	2	3	4	5	6
-1	2	-2	-1	5	-2

e(3, 4)

1	2	3	4	5	6
-1	2	-3	2	5	-2

e(4, 5)

-1	2	-5	2	5	2

e(1, 6)

-1	2	-6	2	5	2

e(1, 2)

2	2	-6	2	5	2

cycle. So stop.

Find and Union algo. complexity is

$O(n)$ — actually $O(n \log V)$.

↳ $O(E \log E + E \log V)$.

Single source shortest path problem (Shortest path tree set):

Dijsktra's Algorithm:-

The problem is minimization pbm and therefore greedy can be solution.

It works on undirected graph as well as directed graph (correct solution).

Input : Graph and source node

Output : shortest path tree set (shortest path to other nodes of the graph).

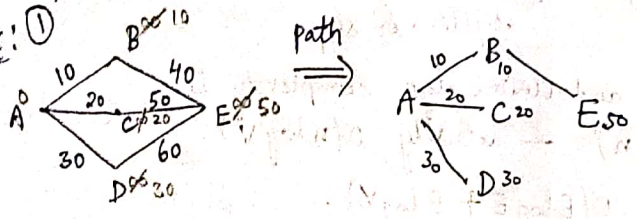
Algo:

1. Initially all nodes will be marked unvisited and call it unvisited set. $O(V)$
2. Set distance to source vertex 0 and all other vertex infinity and set source vertex as current vertex. $O(1)$
3. Relax all the adjacent vertex of current vertex
   e.g. assume u and v are adjacent

$$\text{if } d[u] + c(u,v) < d[v]$$
$$d[v] = d[u] + c(u,v)$$

4. Update current vertex. $\log V \rightarrow O(V)$
   choose the shortest vertex from unvisited set whose distance is minimum from current vertex.
5. repeat 3 and 4 until unvisited set is empty.

ex: ①

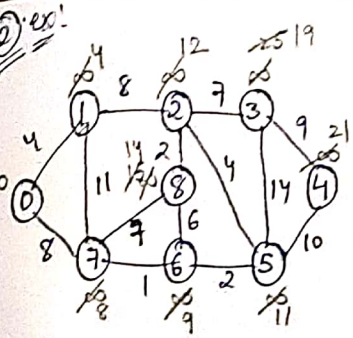


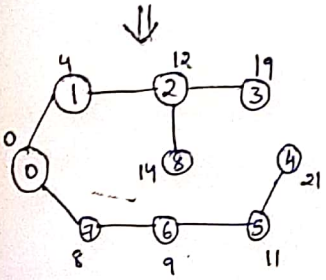
The time complexity of this algo is $O(V) + E \log V$

(We can relax a node until it is unvisited).

for min in W.C how many vertex to relax ie E

② ex:





Visited vertex

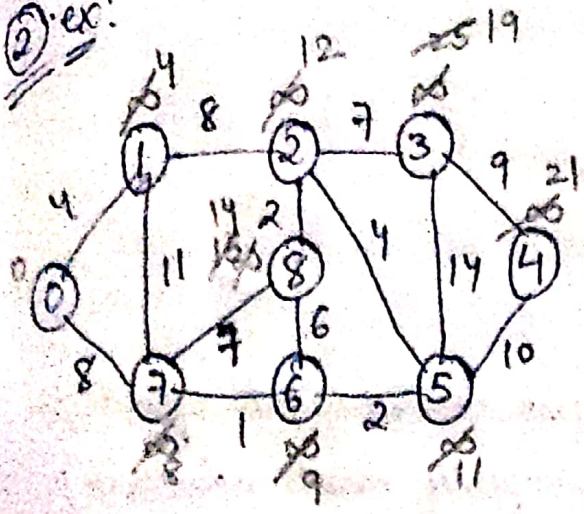
VV

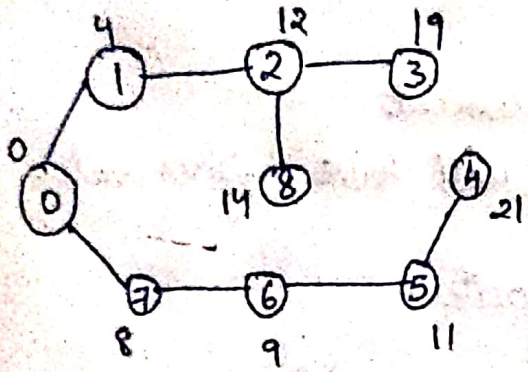
Vertex

	0	1	2	3	4	5	6	7	8
0	0	4	∞	∞	∞	∞	∞	∞	∞
1	0	4	12	∞	∞	∞	∞	8	∞
7	0	4	12	∞	∞	∞	9	8	15
6	0	4	12	∞	∞	11	9	8	15
5	0	4	12	25	21	11	9	8	15
2	0	4	12	19	21	11	9	8	14
8	0	4	12	19	21	11	9	8	14
3	0	4	12	19	21	11	9	8	14
4	0	4	12	19	21	11	9	8	14

Dijkstra Algo. does not work with negative weighted edges. (dis. adv.) → (may or may not).

(2) ex:





↳ Visited vertex

VV ↳ Vertex

	0	1	2	3	4	5	6	7	8
0	0	4	∞	∞	∞	∞	∞	∞	∞
1	0	4	12	∞	∞	∞	∞	8	∞
7	0	4	12	∞	∞	∞	9	8	15
6	0	4	12	∞	∞	11	9	8	15
5	0	4	12	25	21	11	9	8	15
2	0	4	12	19	21	11	9	8	14
8	0	4	12	19	21	11	9	8	14
3	0	4	12	19	21	11	9	8	14
4	0	4	12	19	21	11	9	8	14

↳ (may or may not).

Dijkstra Algo. does not work with negative weighted edges. (dis. adv.).

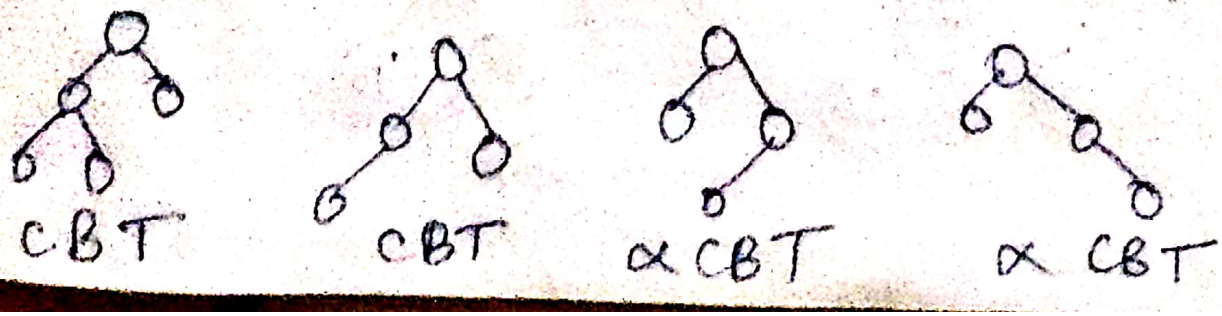
Heap Sort:-

Binary Tree — 0, 1, 2 descendents.

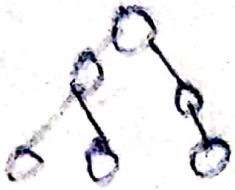
Full Binary Tree — 0 or 2 descendents

Complete Binary Tree — All levels of tree are filled entirely except last level. & last level is filled from left to right.

Almost complete B.T — is same as complete B.T (do not have any application)



∝ CBT



∝ CBT

Heap is a complete B.T. (no matter what operation you perform on Heap, this property must maintain)

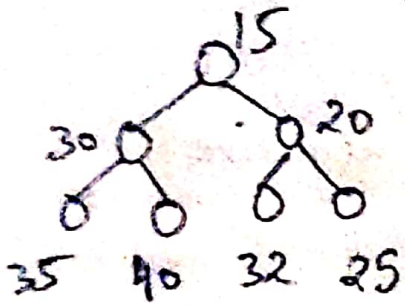
There will be Min heap and Max heap.

Min heap :-

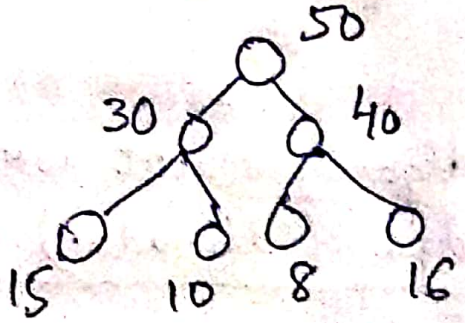
Root will will be minimum element.

All descendants of a parent will have higher value than parent.

Max heap — is just the opposite.



Min heap



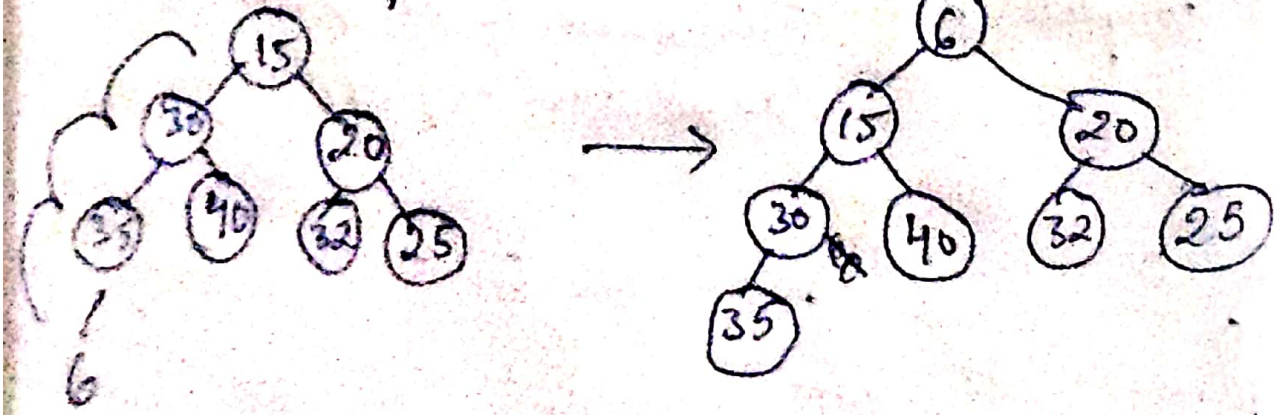
Max heap.

Build heap

Insertion

Deletion (only from root)

→ in min heap



Time complexity on insertion = log n (cost of heapify)

cost of n insertion = n log n (complexity of building a heap)

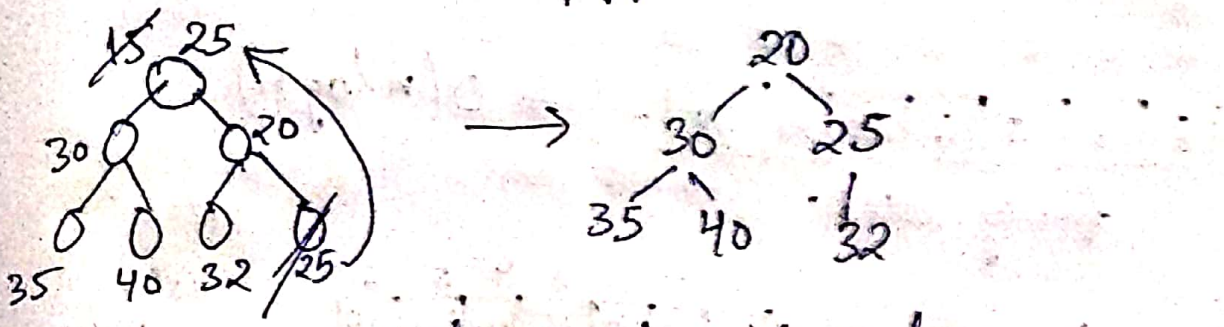
Heapify is putting node in right position.

Deletion:

In the above ex. we can delete 15.

First delete 15 and replace with last element i.e 25 (rule).

And then heapify.



The cost of deleting 1 element is log n

If we delete all elements then O(n log n)

* Heap sort does not follow divide & conquer algo.

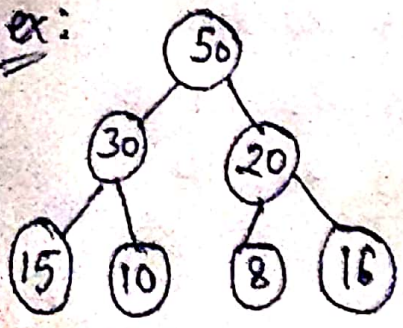
Heap Sort Algorithm:-

1. Create a min/max heap.
2. Delete all elements one by one.
3. The elements will be in the sorted order.

10 9 8 7 6 5 → Min heap

5 6 7 8 9 10 → Max heap

ex:



implementation →

1	2	3	4	5	6	7
50	30	20	15	10	8	16

(if indexing start from 1)

Parent is i

then left child is 2i

right child will be 2i+1

if (indexing start from 0)

parent i
then left child will be $2i+1$
right child will be on $2i+2$

* If there are gaps in array then it is not C.B.T.

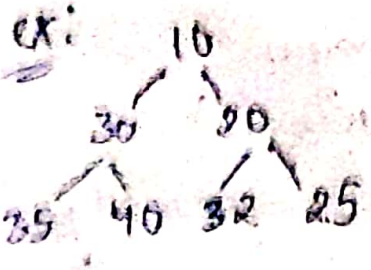
While deletion of 50

finally sorted. → $O(n \log n)$.

→ not part of heap, just using space

using $2i+1$, $2i+2$

ex:



On deleting 10,

25 30 20 35 40 32 | 10 |

Deleting 20, ⇒ 25 30 32 35 40 | 20 10 |

Deleting 25 ⇒ 40 30 32 35 | 25 20 10 |

Deleting 30 ⇒ 35 40 32 | 30 25 20 10 |

Deleting 32 ⇒ 40 35 | 32 30 25 20 10 |

. . . . 35 40 32 30 25 20 10

40 35 | 32 30 25 20 10 |

↳ Hence sorted.

∴ Time complexity is $O(n \log n)$, Space complexity $O(n)$

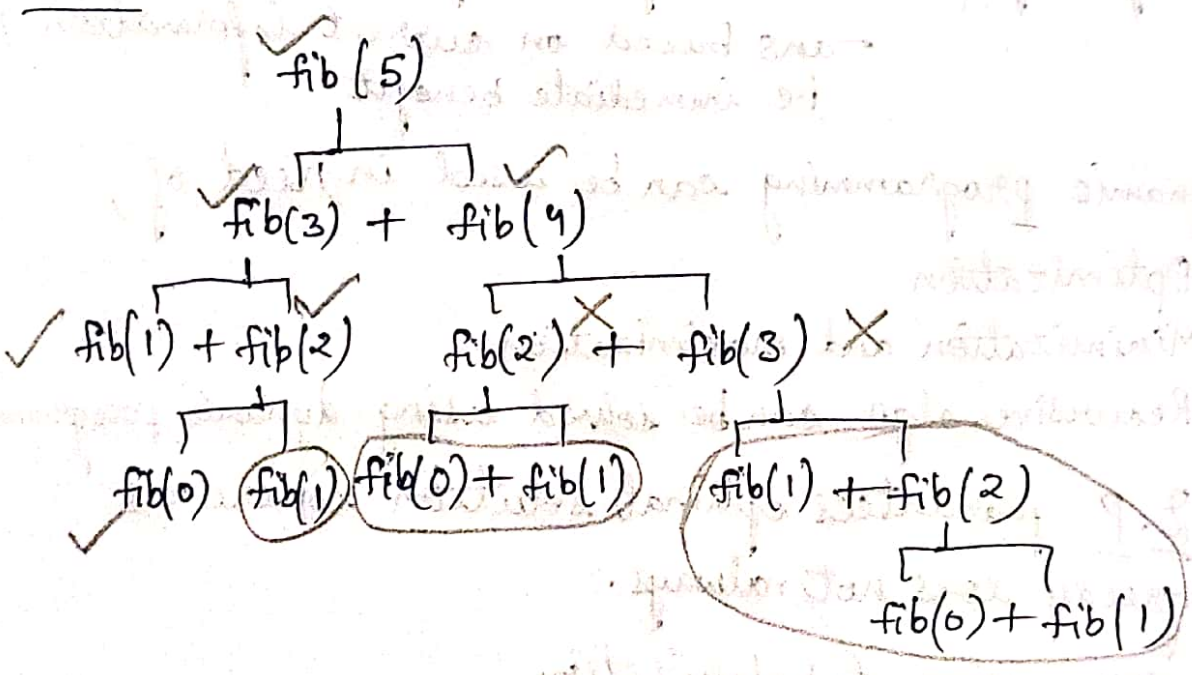
Dynamic Programming :-

* Remember your past.
* There will be overlapping subproblems, so store the result for future reference.
* Follows principle of optimality : optimal solutions to the sub problems contribute to the optimal solution of the original pbm.

ex: Consider Fibonacci Search

```
fib(n)
if n<=1
    return n
return fib(n-2) + fib(n-1).
```

} using memoization

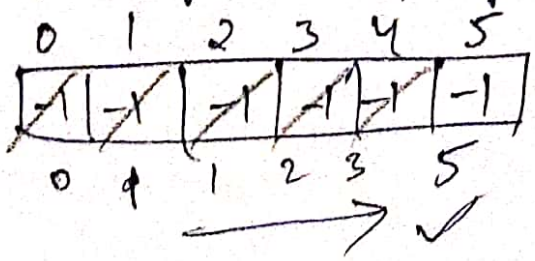


Time complexity $\Rightarrow$ $a(T(n-b)) + 1$

$a^{n-1} = 2^{n-1}$.

Calls $= 2^{16-1} = 2^{15}$

So, now in dynamic programming, the calls are reduced



This is example of memoization.

6 calls.

$n+1 = 6$

$\therefore O(n)$ Hence complexity reduced.

* Dynamic programming uses, memoization (top to bottom) and tabularization (bottom to top)

→ diff. btw dynamic vs divide & conquer
(no overlapping subproblems in D&C) → independent of each other

→ diff. btw dynamic vs greedy
dynamic (look for all possible solution & choose the best one)
greedy (do not look for all possible solution - ans based on current information i.e immediate benefit)

Dynamic programming can be used in need of,
* Optimization
* Minimization and maximization.
* Recursive algo. can be solved using dynamic programming

→ D.P guarantees optimal solution where as greedy does not always.

Same ex: using tabularization,

```
f[0] = 0;
f[1] = 1;
for (i=2; i<=n; i++)
{
  f[i] = f[i-1]+f[i-2];
}
return f[n];
```

0	1	1	2	3	5
F 0	1	2	3	4	5

It is bottom to up because we want f[5) and start from 0.

But in memoization we started from 5 only (in the tree).

Matrix chain multiplication :-

Problem is not to find the solution.

$A*B*C*D$; $(A*B)*(C*D)$ $A*(B*(C*D))$ $((A*B)*C)*D$

$A*((B*C)*D)$ $(A*(B*C))*D$

With n internal nodes, how many binary trees are possible? $\frac{\lfloor 2n}{\lfloor n \lfloor n+1}$

in this ex: $= \frac{6!}{3!\,4!} = 5$

questions: out of all the possible multiplying sequences which sequence we should use?

ans: minimum cost

pbm: minimisation pbm.

applicability of DP: Yes

Cost of matrix multiplication:

Then $O(M+A)$

$= O(M)$ [$\because A < M$]

M → multiplication cost
A → addition cost.

If $[\ ]_{5\times4}\ [\ ]_{4\times3}$

$= [\ ]_{5\times3}$

ex: $A_{5\times4}$ $B_{4\times6}$ $C_{6\times2}$ $D_{2\times7}$

(AB)(CD)

5 4 4 6 6 2 2 7 = 5x4x6 + 5x6x7 + 6x2x7

5 6 6 7

5 7

= 120 + 210 + 84

= 414

(A(BC))D

(5 4 (4 6 6 2)) 2x7 = 4x6x2 + 5x4x2 + 5x2x7

5 4 4 2 2x7

5 2 2 7

5 7

= 158

((AB)C)D = 120 + 60 + 70 = 250

(A(B(CD))) = 392

A((BC)D) = 244

∴ (A(BC))D = 158 ✓ is minimum.

1|23|4

Tabularization (Bottom-Up) [filling diagonally]

M

	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

S (Split)

	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

$M_{11} = M_{22} = M_{33} = M_{44} = 0$ [∵ we didn't multiply AA, BB, CC, DD]

$M_{12} = AB = 5\times4\times6 = 120$.

$M_{23} = BC = 4\times6\times2 = 48$

$M_{34} = CD = 6\times2\times7 = 84$

$M_{13} = \min\left[{}_{M_{22}+}^{(AB)C} M_{12} + 5\times6\times2 \;,\; {}_{M_{11}+}^{A(BC)} M_{23} + 5\times4\times2\right]$

ABC

1|23

0+120 + 60 , 0+48 + 40

88 ✓

$M_{24} = \min\left[{}_{M_{44}+}^{(BC)D} M_{23}+4\times2\times7 \;,\; {}_{M_{34}+}^{B(CD)} M_{22}+4\times6\times7\right]$

BC D

2 3 4

0 + 48 + 56 , 168 + 84 + 0

104 → splitting from 3 (BC)D, 2 3 4

so write 3.

$M_{14} = ABCD$

↳ A(BCD) || (ABC)D || (AB)(CD)

$= \min\left[M_{24}+M_{11}+5\times4\times7,\; M_{13}+M_{44}+5\times2\times7,\; M_{12}+M_{34}+5\times6\times7\right]$.

$= \min[104+0+140,\; 88+0+70,\; 120+84+210]$

$= \min[244, 158, 414]$.

→ splitting from C so 3.

∴ A|BC|D is the split (from S matrix)

Formula :-

$A_{d_0 \times d_1}$ $B_{d_1 \times d_2}$ $C_{d_2 \times d_3}$

(AB)C A(BC)

$\min[m12 + m33 + d_0 d_2 d_3 \;,\; m11 + m23 + d_0 d_1 d_3]$

$\min[m_{\underline{i}\,\underline{k}} + m_{\underline{k+1}\,\underline{j}} + d_0 d_1 d_3 \;,\; m_{\underline{i}\,\underline{k}} + m_{\underline{k+1}\,\underline{j}} + d_0 d_2 d_3]$

$d_{i-1} d_k d_j$ $d_{i-1} d_k d_j$

$Cost[i,j] =$

↳ $\min_{i<=K<j} \left(m[ik] + m[k+1, j] + d_{i-1} d_k d_j \right)$.

Suppose to find Cost[1,4].

$$Cost[1,4] = \begin{cases} k=1 & \min(m[11] + m[2,4] + d_0 d_1 d_4) \\ k=2 & \min(m[1,2] + m[3,4] + d_0 d_2 d_4) \\ k=3 & \min(m[13] + m[44] + d_0 d_3 d_4) \end{cases}$$

↳ again using same formula find its value recursively.

Time complexity :

$n \;\; n-1 \;\; n-2 \;\; n-3 \;\; \ldots \;\; 1$

$[n(n+1)/2] * n$

$= n^2 * n$

$= O(n^3)$

How many ways you can multiply the chain of matrices? i.e parenthesis pbm.

= total no. of binary tree we can generate having n internal nodes.

= $^{2n}C_n/(n+1)$ → (This is Catalan number).

where n is = no. of matrices − 1.

$= \frac{(2n)!}{n!\,n!\,(n+1)} = \frac{(2n)!}{n!\,(n+1)!}$ → This is a Catalan number

where n is internal nodes.

0/1 Knapsack Problem :-

Item	:	0	1	2	3
Weight	:	2	2	4	5
Benefit	:	3	7	2	9

Max capacity of knapsack is 10.

at wt 2, we place the profit

Item	P	W	0	1	2	3	4	5	6	7	8	9	10
0-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	3	2	0	0	3	3	3	3	3	3	3	3	3
1 (0,1)	7	2	0	0	7	7	10	10	10	10	10	10	10
2 (0,1,2)	2	4	0	0	7	7	10	10	10	10	12	12	12
3 (0,1,2,3)	9	5	0	0	7	7	10	10	10	16	16	19	19

2+2=4 profit

7+3=10 profit

max(7,3)

if we take W=5, P=9 only so don't change 10.

W=5+2

(W = 2+2+5)

How to fill rows by max.

formula = max(P[i-1,w], P[i-1, w-w_i] + p_i)

19-9 = 10-7 = 3-3 = 0 stop

item 3 1 0

∴ Hence Max profit 19 obtained on taking 0, 1, 3 items.

0/1 Knapsack Problem:-

o	1	2	3	4	5	6	7
p	10	5	15	7	6	18	3
w	2	3	5	7	1	4	1

Max. capacity 15.

item	p	w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	10	2	0	0	10	10	10	10	10	10	10	10	10	10	10	10	10	10
2	5	3	0	0	10	10	10	15	15	15	15	15	15	15	15	15	15	15
3	15	5	0	0	10	10	10	15	15	25	25	25	30	30	30	30	30	30
4	7	7	0	0	10	10	10	15	15	25	25	25	30	30	30	30	32	32
5	6	1	0	6	10	16	16	16	21	25	31	31	31	36	36	36	36	38
6	18	4	0	6	10	16	18	24	28	34	34	34	39	43	49	49	49	54
7	3	1	0	6	10	16	19	24	28	34	37	37	39	43	49	52	52	54

	1	2	3	4	5	6	7
{	1	1	1	0	1	1	0 }

First 54 is highest which is already in 6 item.
So 7 doesn't contribute put 0.

Ny, 54 - 18 = 36 - 6 = 30 - 15 = 15 - 5 = 10 - 10 = 0 stop

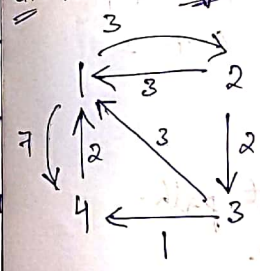
item 6 5 3 2 1

∴ Hence max profit 54 obtained on taking 1, 2, 3, 5, 6 items.

All pair shortest path:- (Floyd-Warshall Algo):

* For single source shortest path, dijkstra's algo has $O(V^2)$ or $O(n^2)$.
* If we run dijkstra's algo n time, every time with different source vertex will get answer to all pair shortest path.
∴ $O(n^3)$ → Greedy also works here.

ex: For F-W Algo



Step 1: Create adjacency matrix

	1	2	3	4
1	0	3	∞	7
2	3	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

Step 2: Update adjacency matrix by introducing a new intermediate node.

1st take 1 is intermidatory node.

$\min(A^k[ij], A^k[ik]\ A^k[kj])$

	1	2	3	4
1	0	3	∞	7
2	3	0	2	10
3	5	8	0	1
4	2	5	∞	0

suppose 2 3 , 2 1 1 3

2 ✓ 3 + ∞

Hy...

✓ => as it is

Now take 2 as intermediatory node.

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

1 1 , 1 2 2 1

0 , 3 + 3

Hy...

Now take 3 as intermediatory node.

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

2 4 , 2 3 3 4

10 , 2 + 1

3 ✓

Hy...

Now take 4 as intermediatory node.

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

1 3 , 1 4 4 1

5 , 6 + 2

Hy...

↳ Hence this is the shortest path.
We have used greedy.

Algo → Floyd-Warshall Algo:-

```
for K = 1, n
    for i = 1, n
        for j = 1, n
            a[i, j] = min( a[i, j] , a[ik] + a[kj] ).
```

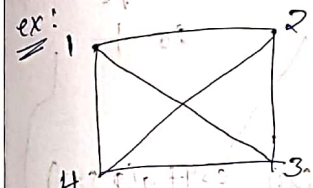
Travelling Salesman Problem:-

1. Travelling Salesman optimization pbm.
2. Travelling Salesman search pbm.

We deal with 2nd pbm only.

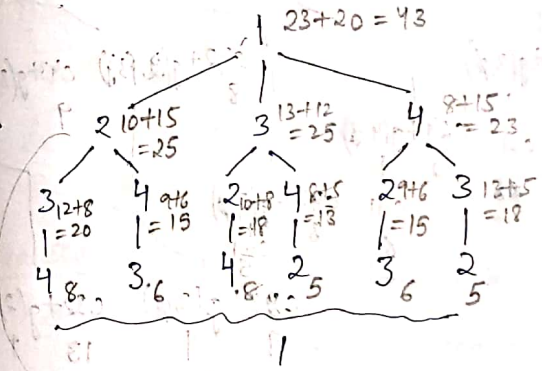
* We need to find a cycle with minimum cost.
(It means we have to visit all vertices at once).

ex:



Cost matrix

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0



Selecting 2-4 because it is < 2-3
&, 2-4 cost = 10 + 15 = 25

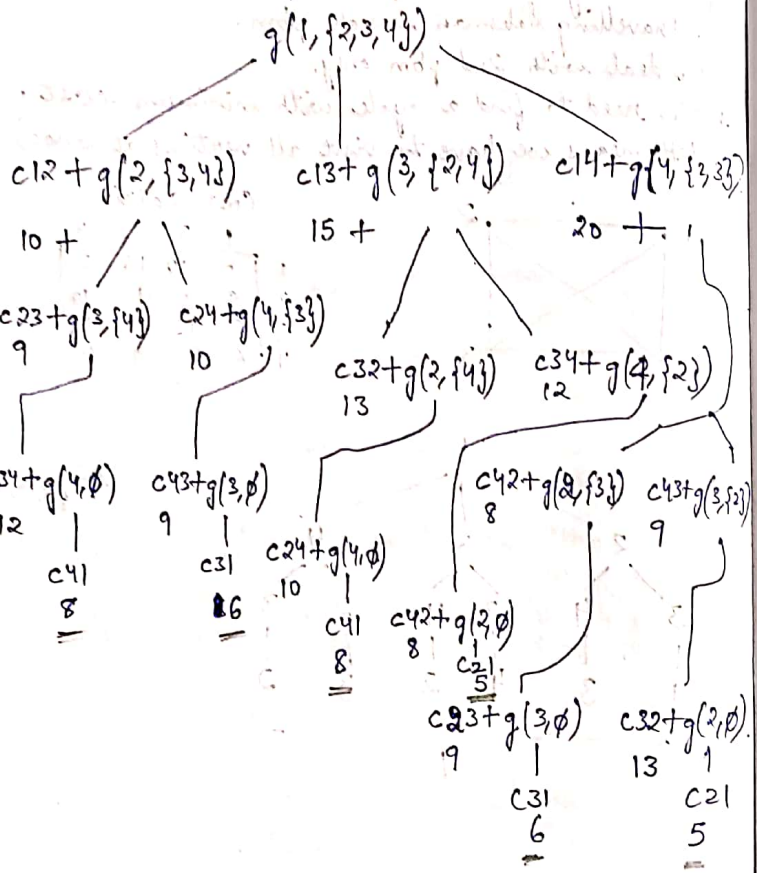
$\{1,2,3,4\}$

$g(1,\{2,3,4\}) = \min\{c1K + g(K,\{2,3,4\}-\{K\})\}$

$\Rightarrow \boxed{g(i,R) = \min\{c_{iK} + g(K, R-\{K\})\}}, K \in R.$

↳ formula using dynamic programming, to go from i to remaining vertices.

Now let's solve using formula.



Time complexity $O(n^2 2^n)$

↳ $(n-1)\,2^n$ (dividing into subsets)

↳ $n \times (n-1)\,2^n$

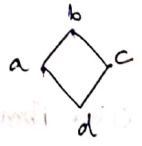
Hence it is $O(n^2 2^n)$.

(NP complete theory) → Gallery 29/6/21

↳ Non-deterministic Polynomial

Unit - 5.

HP ↳ HC ↳



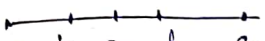
Exactly once every vertex except source vertex.

$n-1$.

HC ab bc cd da

HP bc cd da (HC − 1 edge) = HP

Suppose:



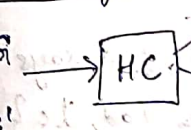
HP ✓

HC ✗

So HP doesn't guarantee HC.

But HC guarantees HP. ✓

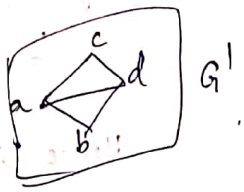
Reduction Algo:-



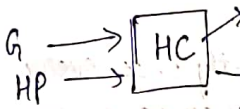
Yes ✓ then G has HP

No → then create G' i.e modify G then again send and check.

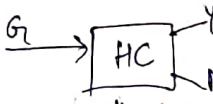
So max 2 times we are using this HC algo.



* G will have HP iff G' have HC.



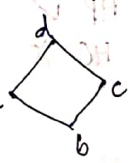
Decision Pbm.



Given decision algo for HC?

Search version sol^n is given as follows.

suppose G →



⇒ 1st we send G to HC algo then it o/p s Yes.

Then if we remove ad edge & send it says no.
So add it to {ad,}.

Next if we remove dc edge & send it says no.
So it imp. for HC hence add to {ad, dc}.

Next bc, ab also works similarly.

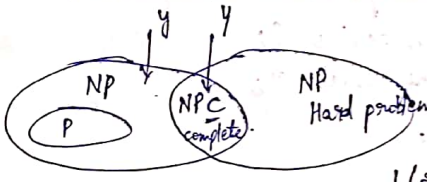
So {ad, dc, bc, ab}.

∴ {ad, dc, bc, ab} forms the HC.

i.e n times we check.

$O(n^{c+1})$ → which is polynomial.

Prover (all the power)	Verifier (limited resources)
If prover says "Yes" there is HC.	→ why should I believe you?
We will take all the edges which forms HC and supply it to verifier.	←
If prover says "No". Solving pbm is not P bounded but given a sol^n it can be verified polynomially.	→ ~~Not possible to verify~~ (NP) ← So, ↓ non-deterministic polynomial



$X \propto_P Y$

$X \notin NP$ → X is not NP.

Y is polynomially reducible to X.

then X is NP hard.

(or) ↴

$\forall\, y \in NP$ if $X \propto_P y$ $X \notin NP$.

So X is NP hard.

NP hard :-

① all problems of NP are p-reducible to x but x does not belong to NP.
then x is NP hard.

② y is NPC.
y is p-reducible to x.
but x does not belong to NP.
then x is NP hard.

NPC :-

All the problems of NP-C.
is p-reducible to x.
and x belong to NP and x is also NP hard.
then x is NPC.

— X — Unit -5 — X —

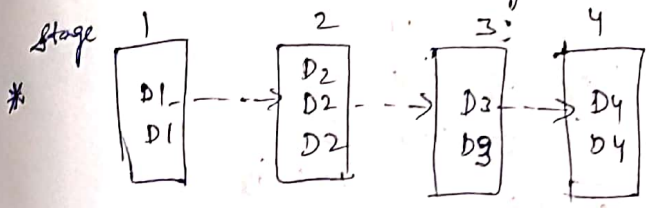
Reliability Design :-

We have to setup a system that is composed of devices d_i for $i = 1, 2, \ldots n$ & they are serially connected.
We know two things about the devices:
cost to purchase device C_i
reliability of the device r_i

d_1 d_2 d_3 d_4
c_1 c_2 c_3 c_4
r_1 r_2 r_3 r_4 and suppose reliability is 0.9 for each device.

Then reliability of the system $= \prod_{i=1..n} r_i$

$= (0.9)^4 = 0.6561$
i.e. 35% chances that system can fell.



$r1 = 0.9$
$1 - (1 - 0.9)^2 = 0.99$

* We have given a cost, with that cost we have to design a system.
* I have to setup a system by buying this devices and connecting them.
* How many copies of each device should I buy within the given cost such that the reliability of overall system is maximized.
* Suppose reliability of each stage is given by a function, $f_i(m_i)$, $1 \le i < n$.

m_i is the no. of copies of devices
reliability is the function of no of devices.

$$\max \left(\prod f_i(m_i) \right)$$

subject to $\sum_{i=1..n} c_i * m_i <= C$ (total cost which can be spent in designing system).

Solution:

Suppose

D_i	C_i	r_i	m_i
D_1	30	0.9	2
D_2	15	0.8	3
D_3	20	0.5	3

$D_1+D_2+D_3$

$C = 105$

Max no. of copies I can buy for each device

$$= \left\lfloor \frac{C - \Sigma c_i}{c_i} \right\rfloor + 1$$

$c_1 + c_2 + c_3 = \Sigma c_i = 65$

$\Rightarrow C - 65 = 40/30 = 1 \quad D_1$

$C - 65 = 40/15 = 2 \quad D_2$

$C - 65 = 40/20 = 2 \quad D_3$

Set method

Set is order pair of (R, C).

$S_0 = (1, 0)$

→ **D1**

$S_1^1 = \{(0.9, 30)\}$ → Reliability.

subscript = device no.

superscript = no. of copy.

$S_1^2 = \{(0.99, 60)\}$ → 2 copies. So $30+30 = 60$.

$1-(1-0.9)^2$

$\therefore S_1 = \{(0.9, 30), (0.99, 60)\}$ rule of dominance.

→ **D_2**

$S_2^1 = \{(0.72, 45), (0.792, 75)\}$ $1-(1-r)^2 = 0.96$

$S_2^2 = \{(0.864, 60), (0.9504, 90)\}$

have to cancel this pair bez we need to take atleast 1 from D_3 then $90+20 = 110 > 105$. So remove it.

$S_2^3 = \{(0.89, 75), (\ldots 105)\}$ $1-(1-r)^3 = 0.992$

0.9×0.992

↳ we cannot accomodate D_3, & atleast one copy of D_3 should be accomadated.

$\therefore S_2 = \{(0.72, 45), (0.792, 75), (0.986, 60), (0.89, 75)\}$

if reliability increases cost should increase if that is not the case then use the rule of dominance i.e. remove one with higher cost.

$\therefore S_2 = \{(0.72, 45), (0.86, 60), (0.89, 75)\}$.

→ **D_3**

$S_3^1 = \{(0.36, 65), (0.432, 80), (0.4664, 95)\}$

$S_3^2 = \{(0.54, 85), (0.64, 100), (\ldots 115)\}$ $1-(1-r)^2 = 0.75$

$S_3^3 = \{(0.63, 105), \ldots\}$ $1-(1-r)^3 = 0.875$

$\therefore S_3 = \{(0.36, 65), (0.432, 80), (0.54, 85), (0.64, 100)\}$.

$$\text{Max}\left(\pi_i \, \phi_i(m_i)\right).$$

$= (0.64, 100)$ this is the reliability.

But what device and count of it?

d_1	d_2	d_3
1	2	2

$$30 + 15 * 2 + 20 * 2$$

$$= 30 + 30 + 40 = 100$$