Module 1

Algorithm

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

- Input : Zero or more quantities are externally supplied.
- output : Atleast one quantity is produced.
- Definiteness : Each instruction is clear & unambiguous
- finiteness : If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite no. of steps.
- Effectiveness : Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Space / Time Complexity

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

Performance Evaluation can be loosely divided into two major phases:

1) A priori estimates (Performance analysis)
2) A posteriori testing (Performance measurement)

→ Space Complexity

* The space needed by each algorithms is seen to

be the sum of the following components

1) A fixed part that is independent of the characteristics of the I/P & O/P. The part typically includes the instruction space (i.e., space for the code), space for simple variables & fixed-size component variables, space for constants & so on.

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

3) The space requirement s(p) of any algorithm p may therefore be written as

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

where c is a constant.

How to Analyse algorithm:

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e, to estimate the complexity function for arbitrarily large I/p. The term "analysis of algorithms" was coined by Donald Knuth.

- Algorithm analysis provides theoretical estimation for the required resources of an algorithm.
- It is the determination of the amount of time

& Space resources required to execute it

Performance Evaluation consist of

* Priory estimate or Performance or asymptotic analysis
* posteriori testing or performance measurement

→ A priori analysis

- It is an absolute analysis
- Independent of language of compiler & types of h/w
- It will give approximate answer.
- Uses asymptotic notations to represent how much time algorithm will take in order to complete its execution.

→ A Posteriori analysis

* It is a relative analysis.
* dependent on language of compiler & types of h/w
* give exact answer.
* don't use asymptotic notations to represent the time complexity.

Time efficiency is analysed by determining the number of repetitions of the basic operation as a function of input size.

↓

It contribute towards the running time of the algorithm.

$$T(n) \approx C_{op} C(n)$$

RUNNING TIME — EXE TIME — NUMBER OF TIMES BASIC OPERATION IS EXECUTED

where n is the input size

Best-Case, Average Case, Worst-Case

For some algorithms, efficiency depends on form of input:

- Worst-case
- Best case
- average case

Worst-case

* $C_{worst}(n)$
* maximum over inputs of size n

Best-case

* $C_{best}(n)$
* minimum over inputs of size n

Average-case

* $C_{avg}(n)$
* "average" over inputs
* Number of times the basic operation will be executed on typical input.
* NOT the average of worst & best case.
* Expected no. of basic operations considered as a random variable under some assumption about the probability distribution of all possible i/p.

Mathematical Analysis (Frequency Count Method)

Time → Time Complexity → • as a func. of i/p. size
• running time of algorithm.

SPACE → Space Complexity → • Space needed by an algorithm
• represented as func. of i/p size 'n'

Q) Algorithm Sum(A, n)

	Cost	Frequency
{		
s=0	c_1	1
for(i=0; i<n; i++)	c_2	n+1
{		
s = s + A[i];	c_3	n
}		
return s;	c_4	1
}		

$T(n) = c_1 + c_2(n+1) + c_3(n) + c_4$
$= c_1 + c_2 \cdot n + c_2 + c_3 n + c_4$
$= c_1 + n(c_2 + c_3) + c_2 + c_4$
$= n(c_2 + c_3)$
$= O(n)$ (Linear Complexity)

$S(n)$ List out the variables

A ⟶ n (array)
s ⟶ 1
i ⟶ 1
n ⟶ 1

$n+3 \approx O(n)$

Q. Writing algo. for adding nxn matrices. Find its time & space complexity.

Algorithm Add(A, B, n)	Cost	Frequency
{		
for(i=0; i<n; i++)	c_1	n+1 / (2n+2)
{ for(j=0; j<n; j++)	c_2	n(2n+2)

```
    c[i,j] = A[i][j] + B[i][j]        c3        n x n
  }
}
```

$T(n) = c_1(2n+2) + c_2(2n+2)n + c_3 n^2$

$= c_1 \cdot 2n + c_1 2 + c_2 \cdot 2n^2 + c_2 \cdot 2n + c_3 n^2$

$= n(2c_1 + 2c_2) + 2c_1 + n^2(2c_2 + c_3)$

$= n + n^2$

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

$S(n)$

A $\longrightarrow n^2$
B $\longrightarrow n^2$
C $\longrightarrow n^2$
i $\longrightarrow 1$
j $\longrightarrow 1$
n $\longrightarrow 1$ $\Rightarrow 3n^2 + 3 = n^2 \approx O(n^2)$

Q. Multiplying two square matrix

```
Algorithm multiply (A, B, n)          Cost        Freq.
{
  for(i=0; i<n; i++)                  c1          2n+2
    for(j=0; j<n; j++)                c2          (2n+2)n
      c[i,j] = 0                      c3          n x n
      for(k=0; k<n; k++)              c4          (2n+2)n^2
      c[i,j] = c[i,j] + A[i,k] + B[k,j]   c5      n^3
}
```

$T(n) = c_1(2n+2) + c_2(2n+2)n + c_3 n^2 + c_4(2n+2)n^2 + c_5 n^3$

$= 2c_1 n + 2c_1 + 2c_2 n^2 + nc_2 + c_3 n^2 + 2c_4 n^3 + 2c_4 n^2 + c_5 n^3$

$= n(2c_1 + c_2) + 2c_1 + n^2(2c_2 + c_3 + 2c_4) + n^3(2c_4 + c_5)$

$= n + n^2 + n^3 = n^3 \approx O(n^3)$

S(n)

A → n^2
B → n^2
i → 1
j → 1
K → 1
C → n^2
n → 1

$= 3n^2 + 4$
$= n^2$
$\approx O(n^2)$

Q. 1)
```
for (i=0; i<n; i++)
  {
   stmt 1
  }
```
$= O(n)$

a)
```
for (i=n; i>0; i--)
  {
   stmt 2
  }
```
$= n+1 = O(n)$

3)
```
for (i=1; i<n; i=i+2)
  {
   stmt 3
  }
```
$= n/2 = O(n)$

4)
```
for (i=1; i<n; i=i+20)
  {
   stmt 4
  }
```
$= n/20 = O(n)$

Q.
```
for (i=1; i<n; i=i*2)
  {
   stmt;
  }
```

Suppose n = 8

$n = 2^k$

1 → 2
2 → 4
4 → 8
8 → X

3 = k

$n = 2^k$

$k = \log_2 n$

$\approx O(\log_2 n)$

$n = 10$

$n = k/2 = 10$

10 b/w 2^3 & $2^4 = n = 2^4$

take ceil → $k = \log_2 10 = 4$

$\log_2 10 = 4$

$i = i * 3 \longrightarrow O(\log_3 n)$

$i = i * 16 \longrightarrow O(\log_{16} n)$

Q.
```
for (i=0; i<n; i++)
for (j=0; j<i + j++)
  { stmt;
  }
```

$= 1+2+3+\cdots+n$

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

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

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

Worst, Best & Average case analysis of algorithm

Here we assume that element can be at any position & is equally likely.

Let 'p' be the probability that for Successful Search, '1-p' be the probability that for unsuccessful Search.

$p \longrightarrow$ Successful Search

$1-p \longrightarrow$ Unsuccessful Search

$\boxed{Pi = P/n}$ = successful search for every possible equal.

$P_i = P/n$

$T_{avg}(n) = \sum P_i \, x_i$

$= \left[1 \times \frac{P}{n} + 2 \times \frac{P}{n} + 3 \times \frac{P}{n} + \cdots\right]$

$= \frac{P}{n}\left[1 + 2 + 3 + \cdots + n\right]$

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

Successful Search $(P=1)$, $= \frac{1(n+1)}{2} = \frac{n+1}{2}$ //

Unsuccessful Search. $T_{avg}(n) = (1-P)n$

$(P=0) = (1-0)n = n$ //

n	n^2	2^n
1	1	2
2	4	4
4	16	16
8	64	256
16	256	65536
100	10000	2^{100}

ASYMPTOTIC NOTATIONS

To enable us to make meaningful statements about time & space complexities of an algorithm, asymptotic notations are used.

Execution time of an algorithm depends on the

1) Instruction set
2) Processor Speed
3) disk I/O Speed

Hence, we estimate the efficiency of an algorithm asymptotically.

• Asymptotic notations are mathematical tool to represent the time complexity of algorithms for asymptotic analysis.

↓

To measure the efficiency of algorithms that
- don't depend on machine specific constants
- don't require algorithms to be implemented
- time taken by programs to be compared.

• Important asymptotic notations include

1) Big Oh (O)
2) Omega (Ω)
3) Theta (Θ)

* Big Oh Notations

• Let f(n) & g(n) be two functions mapping integers to real numbers.

• A function f(n) = O(g(n)) iff $f(n) \leq c \cdot g(n)$ for some $n \geq n_0$ & c.

where n_0 & c are constants which is greater than 0.

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

We know for O(n), $f(n) \leq c \cdot g(n)$

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

$2n + 3 \leq 5n \Rightarrow n = 1$ & $c = 5$

→ $f(n) = 2n^2 + 3n + 5$

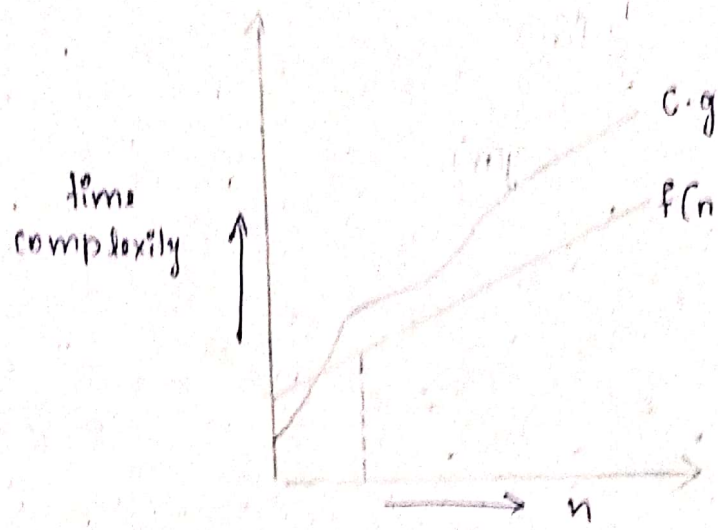
$2n^2 + 3n + 5 \leq c \cdot g(n)$

$2n^2 + 3n + 5 \leq 10n^2$

n = 1

$10 \leq 10$

Then $n_0 = 1$ & $c = 10$.



- function $g(n)$ is an upper bound for function $f(n)$, as $g(n)$ grows faster than $f(n)$ for all $n >= n_0$.

(max time) if n=5
(worst case) time = 5

* Big-Omega (Ω) notation

- The function $f(n) = \Omega(g(n))$ iff $f(n) \geq c \cdot g(n)$, for all $n >= n_0$.

- Ω notation represents the lower bound of the running time of the algorithm. Thus, it provides the best case complexity of an algorithm.

- for Ω notation,

$f(n) \geq c \cdot g(n)$ is a function $f(n)$ belongs to the set $\Omega(g(n))$ if there exist a +ve constant c such that it lies above $c \cdot g(n)$ for sufficiently large n.

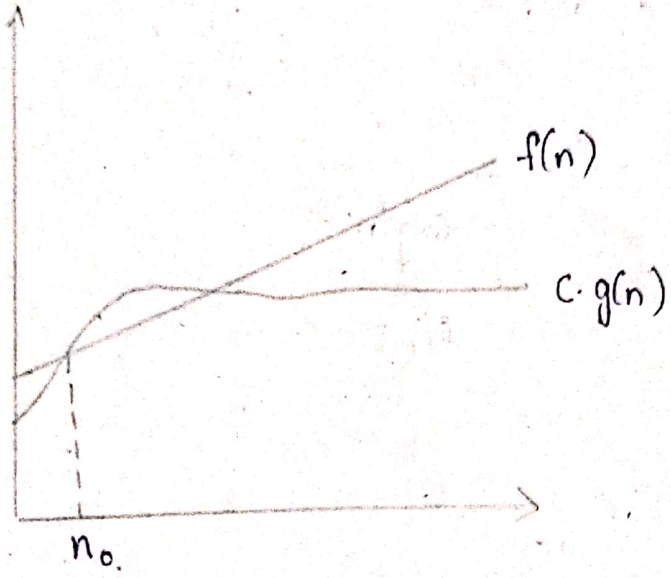
- For any value of n, the minimum time required by the algorithm is given by $\Omega(g(n))$.

(best case)

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

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

$3n + 2 \geq 3n$ for all $n >= 1$, $c = 3$ $g(n) = n$ $n_0 = 1$

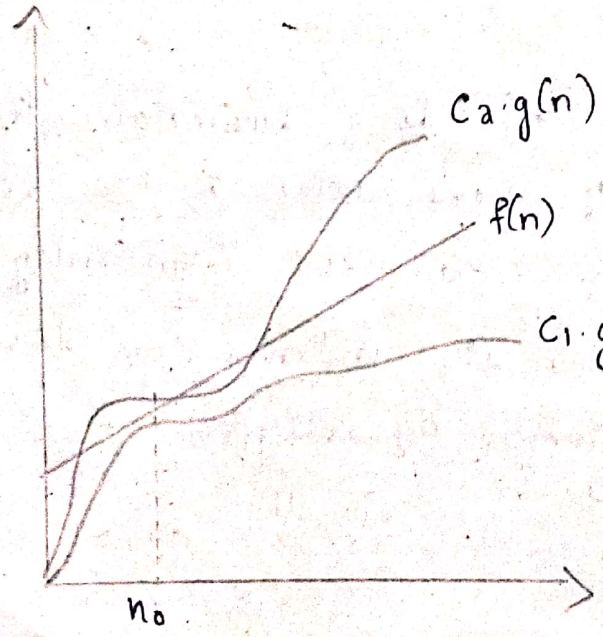


* Big-Theta (Θ) Notation

- Θ notation encloses the func. from above & below.
- Used for analyzing the average case complexity.
- Function $f(n) = \Theta(g(n))$ iff there exist a positive constants c_1, c_2 & n_0 such that

$$c_1 g(n) <= f(n) <= c_2 g(n) \quad \text{for all } n >= n_0$$

- $f(n) = \Theta(g(n))$ states that $g(n)$ is both upper & lower bound on the value of $f(n)$ for all $n >= n_0$.



Q. Suppose $f(n) = \frac{1}{2}n^2 - 3n$ .. S.T $f(n) = \Theta(n^2)$

- We know, for Θ notation,

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

Here $f(n) = \frac{1}{2}n^2 - 3n$ $\quad g(n) = n^2$

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

$$f(n) \leq c_2 \cdot g(n)$$

$$\frac{1}{2}n^2 - 3n \leq c_2 \cdot n^2$$

Suppose $n = 1$, $\quad \frac{1}{2} - 3 \leq c_2$

$$-\frac{5}{2} \leq c_2 \times$$

$n = 2$, $\quad \frac{1}{2} \times 4 - 3 \times 2 \leq c_2 \cdot 4$

$$2 - 6 \leq c_2 \cdot 4$$

$$-1 \leq c_2 \times$$

$n = 7$, $\quad \frac{1}{2} \times 7^2 - 21 \leq c_2 \cdot 49$

$$\frac{\cdot 1}{14} \leq c_2 \checkmark$$

Q. $f(n) = 3n^2 + 4n - 2$ S.T $f(n) = \Theta(n^2)$

- $f(n) \leq c_2 \cdot g(n)$ & $f(n) \geq c_1 \cdot g(n)$

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

$$3n^2 + 4n - 2 \geq c_1 (n^2)$$

$n = 1$, $\quad 3 + 4 - 2 \geq c_1$

$$3n^2 + 4n - 2 \leq c_2 (n^2)$$

$$5 \leq c_2$$

3/12

Mathematical Analysis of Recursive algorithm.

All recursive algorithms must obey three laws :

i) A recursive algo. must have a base case

ii) It must change its state & move towards base case

iii) It must call itself, recursively.

Stopping condition or base case occur when :

> there has been no improvement in the popula for x iteration.

> we reach an absolute no. of generations.

> objective func. value has reached a certain pre-defin value.

General plan for analyzing time efficiency of recursive algorithm

1) decide on a parameter indicating input size
2) Identify algorithm's basic operation.
3) Check whether no. of times basic operation is executed
4) Set up a recurrence relation.
5) Solve the recurrence

General algorithm should ~~contain~~

* Identify the base ~~else~~ case
* Write base case in 'IF' condition
* else part in recursive case.

Eg : n!

```
Algorithm fact(int n)
  if (n==0)
    return 1;
```

else
return n x fact(n-1)

Recurrence relation / Recurrence Equation.

- A recurrence is an equation or inequality that describes a function in terms of its values on smaller i/p.
- Useful for expressing the running time of recursive algorithms.

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

- General form:

$T(n) = 0$ if $n = n_0$ → Base Case

$T(n) = a\,T(n) + g(n)$ for $n > n_0$

- a: no. of times recursive func. is called.
- $g(n)$: time required for doing other operation

Recurrence eqn for factorial

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

$T(n) = 1 \cdot T(n-1) + b$ for $n > 0$

(b: constant)

Three different methods for Solving recurrences:

> Substitution method
> Iteration method
> Master method.

(1) Iteration method :

Here, we expand the recurrence by backward substitution & express it as summation of terms dependent only on 'n' & initial conditions.

Steps

① Expand the recurrence k times
② Work some algebra to express as a summation
③ Evaluate the summation.

$$\begin{cases} T(n) = c & \text{for } n = 0 \quad \text{---------- (1)} \\ T(n) = T(n-1) + b & \text{for } n > 0 \quad \text{---------- (2)} \end{cases}$$

* We use method of backward substitution in this method to solve the problem efficiently.

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

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

Sub. the value of $T(n-1)$ in this eqn

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

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

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

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

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

General formula : $\boxed{T(n) = T(n-k) + kb}$ ---------- (3)

In the above Equation, when base condition is reached, put $n-k = 0$

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

Substitute the value of k in (3)

$T(n) = T(n-n) + nb = T(0) = nb$ ——— (4)

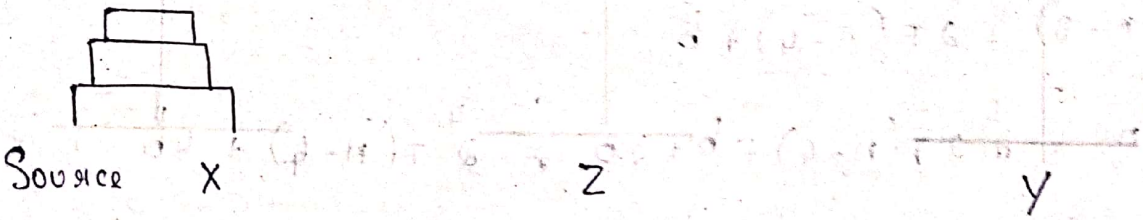
From (1), $T(0) = c$

Substitute in (4)

$T(n) = c + nb$, where c & b are constants

$\boxed{T(n) = O(n)}$

Recursive soln for Towers of Hanoi



Hanoi (x, y, z)

x → source, y → dest^n, z → temporary dest^n

(1) Move (N-1) disk from Source (x) to temp (z)

(2) Move N^{th} disk from Source (x) to dest^n (z)

(3) Move (N-1) disk from temp (z) to dest^n (y)

```
Algorithm hanoi (n, x, y, z)
                 ↓ no. of disk
{
 if(n=1)
   print(" move disk 1 from x to y)
 else
   hanoi(n-1, x, z, y)
   print(" move disk n from x to y)
   hanoi(n-1, z, y, x)
}
```

Recurrence Eq^n :

$T(n) = c$ for $n = 1$ ——— (1)

$T(n) = aT(n-1) + b$ for $n > 1$ ——— (2)

↳ hence (2) is used 2 times

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

$T(n-1) = aT(n-a) + b$

$T(n) = a \cdot aT(n-a) + b + b = a^2T(n-a) + ab$

$T(n-a) = aT(n-3) + b$

$T(n) = a^2 \cdot aT(n-3) + b + ab = a^3T(n-3) + 3b$

$T(n-3) = aT(n-4) + b$

$T(n) = a^3 \cdot aT(n-4) + b + 3b = a^4 \cdot T(n-4) + 4b$

General form:

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

✓ $T(n) = aT(n-1) + b$

$T(n-1) = aT(n-a) + b$

✓ $T(n) = a(aT(n-2) + b) + b = 4T(n-2) + ab + b$

$T(n-2) = aT(n-3) + b$

✓ $T(n) = a(a(aT(n-3) + b) + b)) + b =$

$= a(a(4(aT(n-3) + b) + ab + b$

$= 8T(n-3) + 4b + ab + b$

$T(n-3) = aT(n-4) + b$

✓ $T(n) = 8(aT(n-4) + b) + 4b + ab + b$

$= 16T(n-4) + 8b + 4b + ab + b = a^4T(n-4) + b[1+2+4+8]$

General form:

$$2^k T(n-k) + b(2^k - 1) \longrightarrow (3)$$

when base case is reached, $n = 1$

$$n - k = 1$$

$$k = n - 1 \rightarrow \text{sub. in } (3)$$

$T(1) = c$

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

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

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

$$= c \cdot 2^{n-1} + b \cdot 2^{n-1} - b$$

$-1 = 1/2$

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

$$= \frac{2^n}{2}(b+c) - b$$

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

Q. Algorithm Bin(n)

HW

```
{
  if(n = 1)
   return 1
  else
   return bin((n/2) + 1)
}
```

- Recurrence eqn:

$$T(n) = c \quad \text{for } n = 1 \longrightarrow (1)$$

$$T(n) = T(n/2) + b \quad \text{for } n > 1 \longrightarrow (2)$$

The presence of $(n/2)$ in the function's argument makes backward substitution stumble on values of n that are not powers of 2.

Therefore, the standard approach to solve such recurrence is to solve by taking,

$$n = 2^k$$

& use Smoothness rule.

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

$$T(2^k) = T(2^{k-1}) + 1 \quad \text{for } k > 0 \quad \text{---(1)}$$

$$T(2^0) = 0 \quad \text{for } k = 0 \quad \text{---(2)}$$

By backward Substitution :

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

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

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

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

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

general form, :

$$T(2^n) = T(2^{n-k}) + k \quad \text{---(3)}$$

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

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

$$T(2^n) = n$$

we know $n = 2^k$

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

$$k = \log_2 n \approx O(\log n)$$

(2) Master's Method

Theorem : Let 'a' be a positive integer, let 'b' be an integer greater than 1, & let f be a real-valued func. defined on perfect powers of b. for all perfect powers of n of b, define T(n) by the recurrence.

$$T(n) = aT(n/b) + f(n) \quad \text{---------- (1)}$$

with a non-negative initial value T(1).

Master method is used to solve the recurrence of the form $T(n) = aT(n/b) + f(n)$. $a \geq 1$ & $b \geq 1$.

NOTE : Compare f(n) with $n^{\log_b a}$

f(n) is asymptotically smaller or larger than $n^{\log_b a}$ by a polynomial factor n^2

$$f(n) = n^{\log_b a}$$

Three cases :

(1) Case 1 :

If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \theta(n^{\log_b a})$

(2) case 2 :

If $f(n) = \theta(n^{\log_b a})$ then $T(n) = \theta(n^{\log_b a} \log n)$

(3) case 3 :

If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if

REGULARITY CONDITION ← $af(n/b) = c \cdot f(n)$ for $c < 1$ and all sufficiently large n,

then $T(n) = \theta f(n)$

Q. Given that, $T(n) = 9T(n/3) + n$ using master's theorem

- We know, $T(n) = aT(n/b) + f(n)$

$a = 9 \quad b = 3 \quad f(n) = n$

$n^{\log_b a} = n^{\log_3 9} = n^{\log_3 (3)^2} = n^{2 \log_3 3} = \underline{n^2}$

$f(n) = n^{\log_b a}$

$n = n^2$ (case 1)

$\theta(n^{\log_b a - \epsilon}) = \theta(n^{\log_b a})$

$\Rightarrow \theta(n^2)$

Q. $T(n) = T(2n/3) + 1$

$a = 1 \quad b = \frac{3}{2} \quad f(n) = 1$

$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$

$f(n) = n^{\log_b a}$

$1 = 1$

case 2:

$= \theta(\log^{ba}) \cdot \log n = \theta(1 \cdot \log n) = \underline{\theta(\log n)}$

Q. $T(n) = 3T(n/4) + n \log n$

$a = 3 \quad b = 4 \quad f(n) = n \log n$

$n^{\log_b a} = n^{\log_4 3} = n^{0.793}$

$f(n) = n^{\log_b a}$

$n \log n \geq n^{0.793}$

<u>case 3</u> $f(n) = \Omega(n^{\log_b a + \omega})$

Regularity condition: $af(n/b) = c \cdot f(n)$ for $c < 1$

$f(n) = n \log n \Rightarrow f(n/b) = (n/b) \log(n/4)$

3. $\left(\frac{n}{4} \log(n/4)\right) = c(n \log n)$

$3\left(\frac{n}{4} \log\left(\frac{n}{4}\right)\right) \leq \frac{3}{4}(n \log n)$ $\left(c = \frac{3}{4}\right)$ ✗

$\boxed{\frac{a}{b^k} < 1}$ $= \frac{3}{4^1} < 1$

$= \frac{3}{4} < 1$

K = power of log

$f(n) = n \log^{k} n$

$= \theta(f(n)) = \theta(n \log n)$

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

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

$n^{\log_b a} = n^{\log_2 4} = n^{\log_2 2^2} = n^{2 \log_2 2} = n^2$

$f(n) = n^{\log_b a}$

$n < n^2$

case 1 $\theta(n^{\log_b a - \omega}) = \theta(n^{\log_b a}) = \theta(n^2)$

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

$a = 4 \quad b = 2 \quad f(n) = n^2$

$f(n) = n \log^{b} a$

$n^2 = n^2$

case 2 $= f(n) = \omega(\log_b^a \cdot \log n) = \theta(n^2 \log n)$

Q) $T(n) = 4T(n/2) + n^3$

$a = 4 \quad b = 2 \quad f(n) = n^3$

$f(n) = n \log_b a$

$n^3 > n^2$

case 3 $f(n) = \Omega(n^{\log_b a + \omega})$

$a \cdot f(n/b) = c \cdot f(n)$

$4 f(\frac{n}{2}) = c \cdot n^3$

$4 \cdot (\frac{n}{2})^3 = c \cdot n^3$

$\Rightarrow c = 1/2$

$T(n) = \cancel{\Theta f \log}\ \Theta(f(n)) = \underline{\Theta(n^3)}$

(HW)

1) $T(n) = 5T(n/2) + \Theta(n^3)$

2) $T(n) = 27\, T(n/3) + \Theta(n^3/\log n)$

Q1 $T(n) = 5T(n/2) + \Theta(n^3)$

$a = 5 \quad b = 2 \quad f(n) = n^3$

$n^{\log_b a} = n^{\log_2 5} = n^{2.322}$

$f(n) = n^{\log_b a}$

$n^3 = n^{2.322} \Rightarrow n^3 > n^{2.322}$

case 3 $f(n) = \Omega(n^{\log_b a + \omega}) \Rightarrow (n^3 = n^{2.322 + \omega})$

$\Rightarrow (\omega > 0)$

$a \cdot f(n/b) = c \cdot f(n)$

$5 \cdot f(n) = n^3$

$f(n/2) = (\frac{n}{2})^3$

$5 \cdot (\frac{n}{2})^3 = c \cdot (n)^3$

$\frac{5}{8} = c$ $\quad c = \frac{5}{8} \Rightarrow c$

$\frac{5}{8} < 1$ ✓

$T(n) = \theta(f(n)) = \underline{\theta(n^3)}$

Q.2 $T(n) = 27\, T(n/3) + \theta(n^3/\log n)$

$a = 27 \quad b = 3 \quad f(n) = \frac{n^3}{\log n}$

$n^{\log_b a} = n^{\log_3 27} = n^{\log_3 (3)^3} = n^{3\log_3 3} = \underline{n^3}$

$f(n) = n^{\log_b a}$

$\frac{n^3}{\log n} \leq n^3$

Case 1 $\quad f(n) = O(n^{\log_b a - w})$

$T(n) = \theta(n^{\log_b a}) = \underline{\theta(n^3)}$

Solution

Divide And Conquer

CONTROL ABSTRACTION

→ Divide and Conquer

- It is a top-down approach for designing algorithms that consist of dividing the problem into smaller sub problems hoping that the solutions of sub problems are easier to find.
- Then composing the partial solutions into

the solutions of the original problem.

→ 3 parts are

- divide
- conquer
- combine

① Divide :

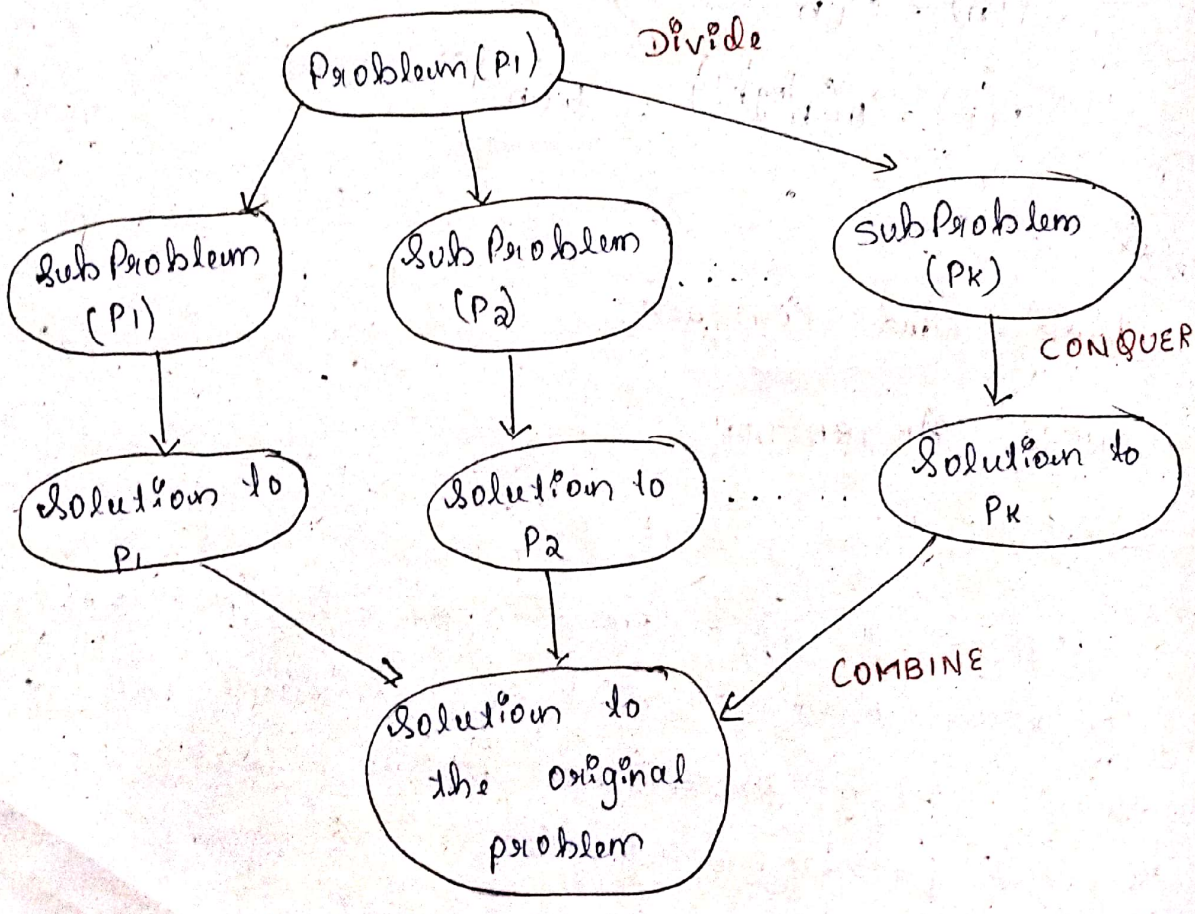
Dividing the problem into number of sub problems that are smaller instances of same problem.

② Conquer :

Conquer (solve) the sub problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.

③ combine :

Combining the solutions of subproblems into solution of the original problem.



control abstraction for recursive divide & conquer method with problem instance 'P'

```
divide-and-conquer(P)
{
  if small(P)  // P is very small, sol^n is trivial
    return solution(s);
  else
    divide the problem P into k instances P1, P2 ... Pk
    return (combine (divide-and-conquer(P1),
                     divide-and-conquer(P2),
                     divide-and-conquer(Pk)))
}
```

The solution of the above problem is ~~divided~~ described by recurrence, assuming size of P denoted by n.

$$T(n) = \begin{cases} g(n) \\ T_{n_1} + T_{n_2} + T_{n_3} + \cdots T_{n_k} + f(n) \end{cases}$$

where $f(n)$ ⇒ time to divide elements & combine the solutions.

Run time is determined by size & no. of sub problems to be solved in addition to the time required for decomposition.

General complexity competition:

$$T(n) = \begin{cases} T_1, & n = 1 \\ aT_{n/b} + f(n), & n > 1 \end{cases}$$

where a & b are constants

[a ⟶ Number of sub-Problems
  n/b ⟶ Size of each sub-Problems]

Eg: MERGE SORT RECURRENCE

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

Solution for merge sort recurrence = $\Theta(n \log n)$

Algorithms that uses divide and conquer :

1) Merge Sort
2) Quick Sort
3) Matrix Multiplication

MERGE SORT

* Uses divide and conquer strategy.
* The basic idea is to > divide the list into sublist,
  > sort each sublist and
  > finally merge them to get a single sorted list.

The recursive implementation of 2-way merge sort divides the array into 2 parts of sublists, then merges them into a single sorted list.

To sort : A[lb ... ub]

divide into 2 array

A[lb ... mid] B[mid+1 .... ub]

mid : $\frac{up + lb}{2}$

floor

algorithm :

MERGE (A, P, q, r)

{ $n_1 = q - p + 1$

p = lb
q = mid
r = ub

$n_2 = r - q$

let $L[1 \ldots n_1+1]$ and $R[1 \ldots n_2+1]$ be new arrays

```
for(i=1 to n1)
    L[i] = A[p+i-1]
for(j=1 to n2)
    R[j] = A[q+j]
L[n1+1] = ∞
R[n2+1] = ∞
i=1, j=1
for(k=p to r)
    if(L[i] ≤ R[j])
        A[k] = L[i];
        i = i+1;
    else
        A[k] = R[j]
        j = j+1;
}
```

Eg:

	1	2	3	4	5	6	7	8
A	1	5	7	8	2	4	6	9
								r

⇒ Here left half of the array is sorted and right half of the array is also sorted i.e, Merging algorithm works on two sorted arrays.

(i) firstly, it copies the 2 sorted halves into arrays L & R. (adding ∞ at the end)

L

1	5	7	8	∞
i				

R

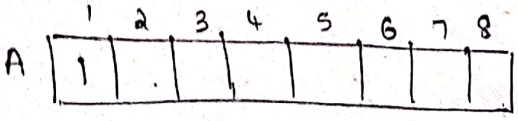
2	4	6	9	∞
j				

if i pointing to the first element of L & R.

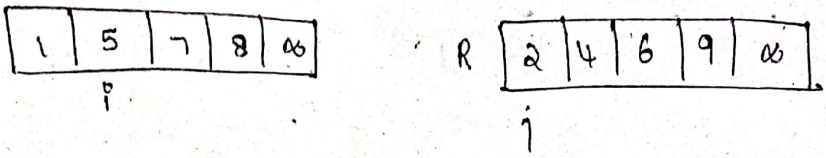
(ii) comparing if $L[i] \leq R[j]$

i.e, 1 < 2

if so L[i], i.e. 1 is copied to the initial array A



then, increment i ⟶ i = i+1

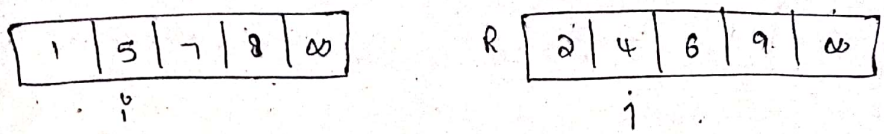


→ if $L[i] \leq R[j]$ is false

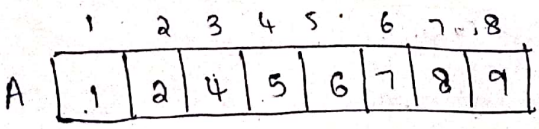
R[j] is copied to the initial array A & j is incremented)

compare 5 ≤ 2 ⟶ false

j = j+1 & 2 is copied to A



IIIly do all the steps.



algorithm for Merge Sort:

MERGE_SORT (algorithm)

MERGE_SORT (A, P, r)
First Last

{ if (P < r)

middle q = $\lfloor (P+r)/2 \rfloor$

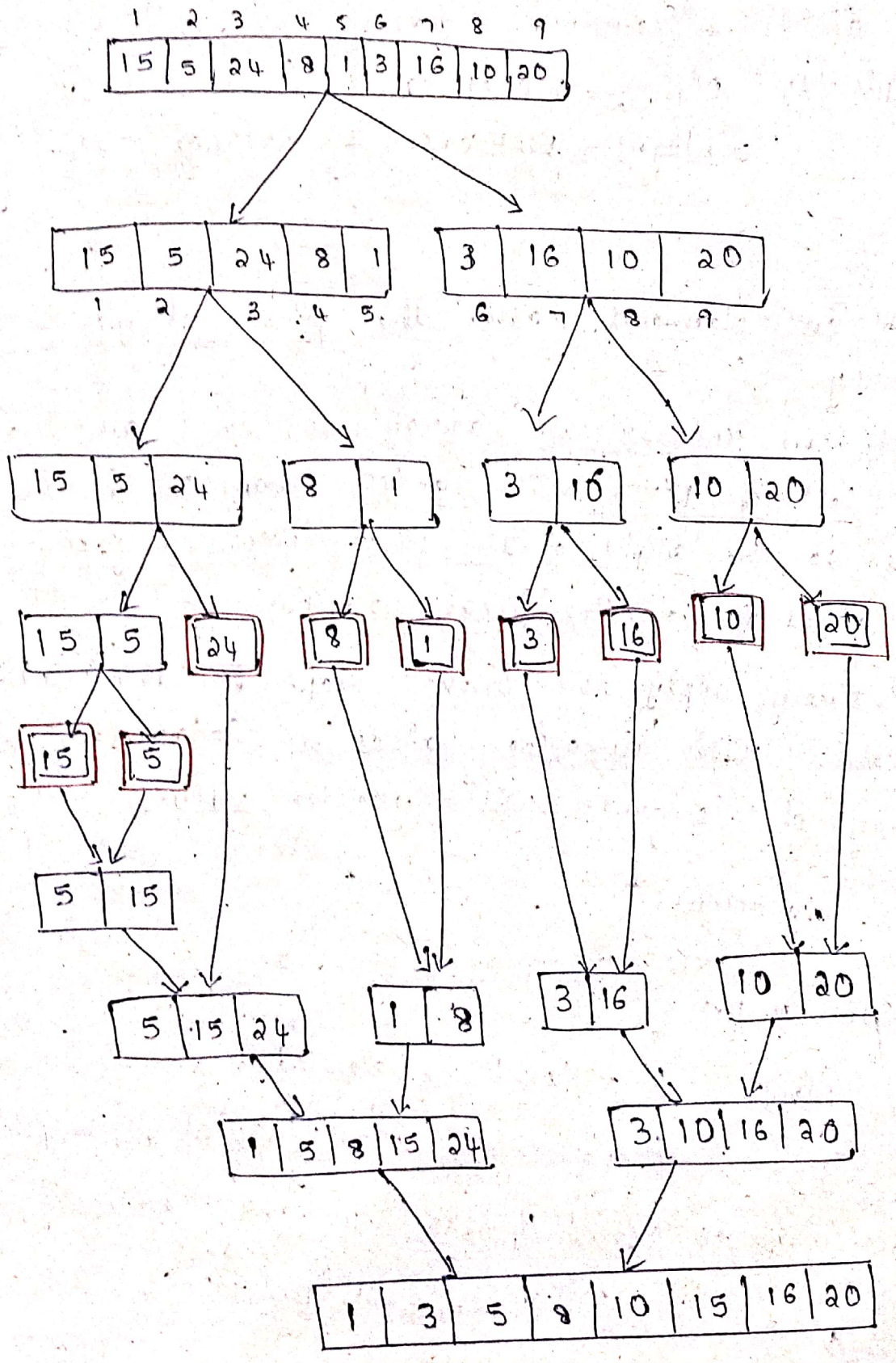
MERGE_SORT(A, p, q)

MERGE_SORT(A, q+1, r)

MERGE(A, p, q, r)

}

Eg:



Quick Sort

* In-place comparison (no extra array)
* not stable
* fast
* Use divide & conquer strategy
* Complexity $O(n^2)$ — worst case
  $O(n \log n)$ — Best case & Average case

Steps

→ Pick an element called the pivot element from the array.

→ Partition reorder the array. so that all the elements to the left of pivot will be less than it & all the elements to the right of the pivot will be greater than it. Now pivot is in its final position.

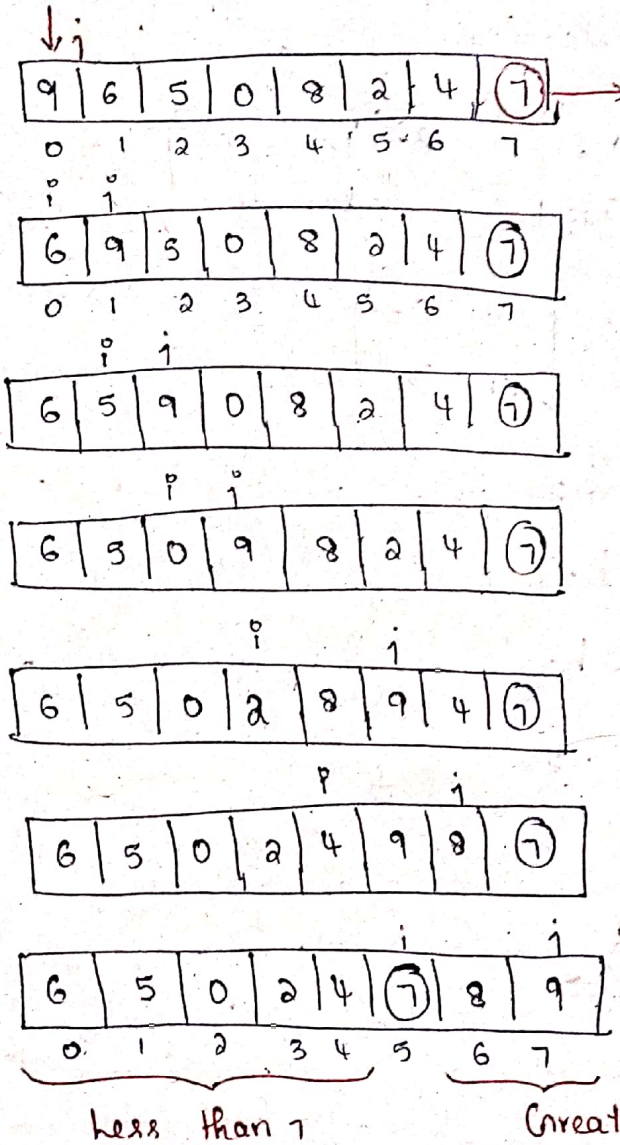
→ Recursively apply the above steps to the sub-array of elements with smaller values & separately to the sub array of elements with greater values.

Partition algorithm

```
PARTITION ( A, p, r)
{
    x = A[r]        // x is Pivot element.
    i = p-1         // i Initially doesn't point to any element
    for (j=p to r-1) // scan upto last element
    {
        if (A[j] ≤ x)  // element ≤ Pivot
        {
            i = i+1
            exchange A[i] with A[j]
```

}
}
exchange A[i+1] with A[r]
return i+1
}

Eg:



And 7 is in its correct position.

QUICK SORT ALGORITHM

```
QUICK_SORT(A, p, r)
{
  if (p<r)
  { q = PARTITION(A, p, r) // Position of Pivot element
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)
} }
```

STRASSEN'S MATRIX MULTIPLICATION

It is an algorithm for matrix multiplication which is faster than the standard matrix multiplication algorithm.

① Naive matrix multiplication

Two matrices $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ $B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$

$$C = A \times B = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$c_{11} = a_{11} * b_{11} + a_{12} * b_{21}$

$c_{12} = a_{11} * b_{12} + a_{12} * b_{22}$

$c_{13} = a_{21} * b_{11} + a_{22} * b_{21}$

$c_{14} = a_{21} * b_{21} + a_{22} * b_{22}$

$$c_{ij} = \sum_{i=1}^{n} A_{ik} * B_{kj}$$

Algorithm

```
for(i=0; i<n; i++)
{
  for(j=0; j<n; j++)
  {
     c[i,j] = 0
     for(k=0; k<n; k++)
     {
       c[i,j] += A[i,k] * B[k,j]
     }
  }
}
```

It has 3 for loops & each will iterate n times

∴ Time complexity = $O(N^3)$

By using divide & conquer

- The Strassen's algorithm is a recursive method for matrix multiplication. where we divide the matrix into 4 sub-matrices of dimensions $n/2 \times n/2$ in each recursive step.
- 'n' should be power of 2

Eg: Consider two 4×4 matrices A and B that we need to multiply. A 4×4 can be divided into four 2×2 matrices.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

(A11) (A12) (A21) (A22) (B11) (B12) (B21) (B22)

i.e., $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$

where A11 & B11 ... are not elements. They are matrices.

Algorithm

```
MM(A, B, n)
{
  if (n<=2)
  {
    C11 = a11 * b11 + a12 * b21
    C12 = a11 * b12 + a12 * b22
    C21 = a21 * b11 + a22 * b21
    C22 = a21 * b12 + a22 * b22
  }
```

```
else
{
    mid = n/2 ; // divide matrix into 2x2
    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)
}
}
```

Recurrence Relation:

$$T(n) = \begin{cases} 1, & n \le 2 \\ 8\,T(n/2) + n^2 & n > 2 \end{cases}$$

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

$a = 8 \quad b = 2 \quad f(n) = n^2$

$$\log_b^a = \log_2 8 = \frac{\log 8}{\log 2} = 3$$

~~Ans~~

$$n^{\log_b a} = n^3$$

$$f(n) < n^{\log_b a}$$

$$n^2 < n^3$$

$$\theta(n^{\log_b a}) = \underline{\underline{\theta(n^3)}}$$

Module II

GREEDY STRATEGY

- One of the algorithm design strategy.
- for solving optimisation problem.
- Algorithm picks the best solution at the moment without regarding consequences.
- It picks the immediate o/p, but doesnot consider the big picture, hence it is considered as greedy.
- It shows an optimal solution that may or may not be true.

application :

1) Knapsack problem
2) MST
3) Job sequencing
4) Huffman coding
5) Optimal merge pattern
6) Dijkstra's algorithm.

Greedy algorithm has 5 components :

1) Candidate set
2) A selection function
3) A feasibility solution
4) An objective function
5) A solution function

① Candidate set : A solution is created from this set

② A selection function : used to choose the best candidate to be added to the solution

③ A feasibility solution : Used to determine whether a candidate can be used to contribute the solution.

④ An objective function : Used to assign a value to a solution or a partial solution

⑤ A solution function : Used to indicate whether the complete soln has been reached.

→ A subset that satisfies the constraint is called a feasible solution.

↓

A feasible solution that maximizes or minimizes a given (objective) function is called optimal solution

→ The greedy method finds optimal solution from feasible solutions.

Eg: • Let us take a problem, P

• We want to move from location A to location B

• But the constraint is that we want to cover this journey in ~~12 th~~ 12 hours.

P: (A) —12 hour→ (B)

There are many solution, we can travel by

* walk
* car
* auto

* train } but the feasible solution are by train or
* flight } by flight

Feasible ⟶ satisfying the constraint

→ Now we wand to cover the journey with minimum cost (MINIMIZATION PROBLEM)

→ So out of two feasible solutions, the above constraint is satisfied i.e. minimum cost or best result

→ train :—— minimum cost satisfying solution

→ so there can be
> more than one feasible solution
> only one optimal solution.

CONTROL ABSTRACTION

• Greedy method says that a problem can be solved in stages.

• In each stage we will consider one input from a given problem & if that i/p is feasible, then we include it in the solution.

• So by including all those input which are feasible we get optimal solution.

algorithm

```
Algorithm - greedy(a, n)        n → no. of solutions
                                a → problem
{
  for(i=1 to n) do
  {
    x = select(a); // check sol^n
    if (feasible(x)) then // whether each sol^n is
                          feasible
    {
```

```
        Solution = Solution + x;  // add to solution
    }
}
```

We want to buy the best car. For that all the brands are checked. checking with the features we select the best one by sorting latest released ones.

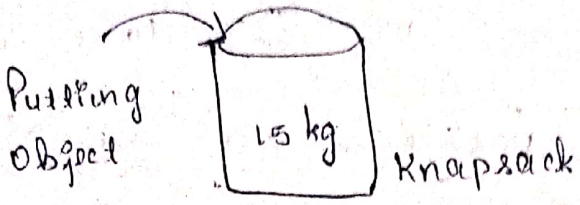
KNAPSACK PROBLEM

- Combinational optimization problem.
- It appears as a sub problem in many more complex mathematical model of real world.
- According to the problem statement
  - i) There are n items in the store
  - ii) weight of i^{th} item, $w_i > 0$
  - iii) Profit for i^{th} item, $p_i > 0$
  - iv) capacity of Knapsack, m

Eg:

> Suppose there are 7 objects
> Every object has some profit associated with it.
> Also object have some weight
> There is a bag or knapsack, here, its capacity is 15 kg.
i.e, n = 7, m = 15 kg

Object	1	2	3	4	5	6	7
Profit(p)	10	5	15	7	6	18	3
weight	2	3	5	7	1	4	1



Object	1	2	3	4	5	6	7
Profit(p)	10	5	15	7	6	18	3
weight(w)	2	3	5	7	1	4	1
P/w	5	1.5	3	1	6	4.5	3

Profit per kg = Profit/weight

- We are filling the bag with all these objects.
- And we will carry this bag to a different place & will sell these objects so that we get some profit.
- That profit is the gain that we get from by transferring these objects to some location. So for transferring we need to carry this container or bag.
- So we can say that the knapsack is a container. The problem is called CONTAINER LOADING PROBLEM (Problem is filling these objects to a container)
- When we are filling the objects, the problem is whether it exceeds knapsack's capacity.

our constraint :-> Total weight ≤ 15

> getting max^m profit (GREEDY APPROACH)

- Check whether the object is included in our knapsack & how much we can include

$$x = (x_1, x_2 \ldots x_n)$$

↓

Then x value can be either

$0 \leq x \leq 1$ (means we can take fractions)

- So the knapsack problem can be divisible

i.e, if object = 5 kg

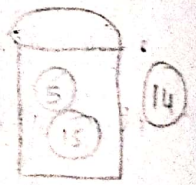
we can put 2 kg from 5kg.

- Next we are selecting a criteria for including item to our bag. So that we are getting maximum profit by including more object.
- Select the object having high profit:

∴ Here maximum profit = (18)

Object = 4.

But we may not be sure that we can include maximum number of objects. So we are finding the Profit/weight value. i.e, Profit/kg.



* So first we are adding 5th object.

i.e, we get profit 6. (object weight = 1)

* Next high profit = 5 for

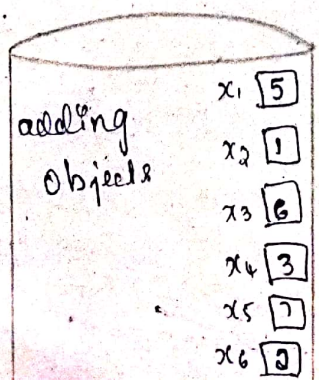
Object = 1

weight = 2

Remaining weight = 14

So add object 1 to the bag & get the profit filled as 5.

similarly



$x_1 = 1$ (1 means we are taking full kg of that

$x_2 = 1$ particular object)

$x_3 = 1$

$x_4 = 1$

$x_5 = 1$

$x_6 = 2/3$

$x_7 = 0$

Total weight = $x_1 \times$ (weight of [5]) + $x_2 \times$ (weight of [1]) +
$x_3 \times$ (weight of [6]) + $x_4 \times$ (weight of [3]) +
$x_5 \times$ (weight of [7]) + $x_6 \times$ (weight of [2]) +
$x_7 \times$ (weight of [4]).

$= (1 \times 1) + (1 \times 2) + (1 \times 4) + (1 \times 5) + (1 \times 1) + \left(\frac{2}{3} \times 3\right)$
$+ 0 \times 7$

$= 1 + 2 + 4 + 5 + 1 + 2 + 0 = \underline{15 \text{ kg}}$

Total Profit = $x_1 \times$ (profit of [5]) + $x_2 \times$ (profit of [1]) +
$x_3 \times$ (profit of [6]) + $x_4 \times$ (profit of [3]) +
$x_5 \times$ (profit of [7]) + $x_6 \times$ (profit of [2]) +
$x_7 \times$ (profit of [4])

$= (1 \times 6) + (1 \times 10) + (1 \times 18) + (1 \times 15) + (1 \times 3) + \left(\frac{2}{3} \times 5\right)$
$+ (0 \times 7)$

$= 6 + 10 + 18 + 15 + 3 + \frac{10}{3} + 0 = \underline{54.6}$

So the ~~constant~~ constraint for the given problem:

$\sum_{i=1}^{n} x_i w_i \leq m \longrightarrow$ maximum capacity of bag

total weight

Objective:

$$\max \sum_{i=1}^{n} x_i p_i \quad (\max^{m} \text{profit})$$

Algorithm:

```
Greedy_fractional_knapsack( w, m, x[] )
{
  for i=1 to n
    do x[i] = 0
  weight = 0    // initially
  while( weight < m )        // m = maximum weight
  do i = best remaining item
  if weight * wi ≤ m
    then x[i] = 1
    weight = weight + wi
  else
    x[i] = (m - weight) / wi
    weight = m
  return x
}
```

MINIMAL SPANNING TREE

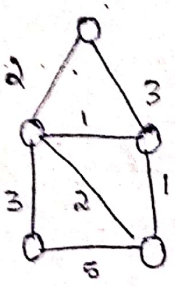
A minimum spanning tree or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles & with minimum possible total edge weight.

i.e. it is a spanning tree whose sum of edge

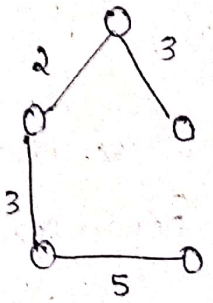
weights is as small as possible.

Minimization

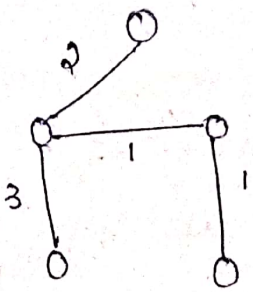
Spanning Tree : Given a undirected graph, final graph should contain all vertices of initial graph, edges should be one less than initial graph + that should not form any cycle.



GRAPH



SPANNING TREE
COST = 13



MST COST = 7

$G = (V, E)$

In MST, $G' = (V', E')$,

$V' = |V|$

$E' = |V| - 1$

Number of spanning tree can be made out of a graph = $|E| C_{|V|-1}$ → no. of cycles.

There are algorithms (Greedy) finding out MST without finding all the spanning tree. They are

① Prim's Algorithm

② Kruskal's Algorithm

PRIM's ALGORITHM

* It is a greedy algorithm that finds MST for a connected weighted undirected graph.

* It finds a subset of edges that form a tree that includes every vertex, where the total weight of all the edges in a tree is minimised.

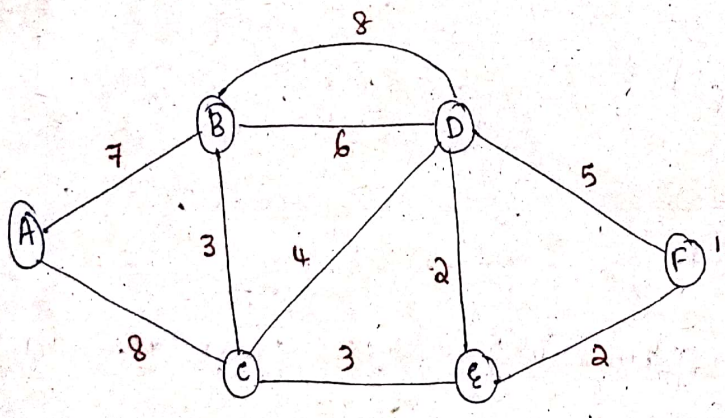
Steps:

(1) Initialise the MST with a vertex chosen at random.

(2) Find all the edges that connect the tree to new vertices.
Find minimum & add it to the tree.

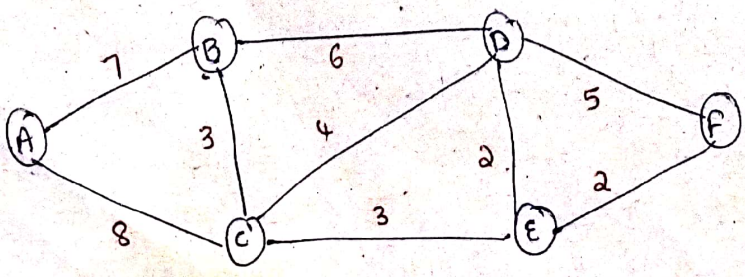
(3) Keep repeating step (2) until we get a MST

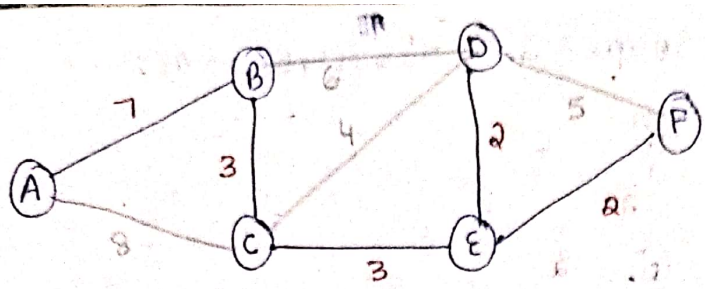
eg:



First, we have to eliminate
→ self loops
→ parallel edges

Remove loops & parallel edge, we get





cost = 7 + 3 + 3 + 2 + 2 = 17

solution2.

KRUSKAL'S ALGORITHM

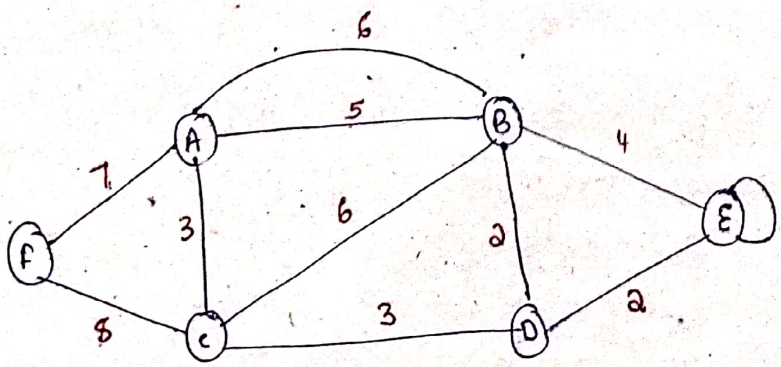
- To find MST
- Uses greedy approach
- This algorithm treats the graph as a forest & every node it has as an individual tree.
- A tree connects to another only & only if, it has the least cost among all available option & does not violate MST properties.

① Sort all the edges in increasing order of their weight.

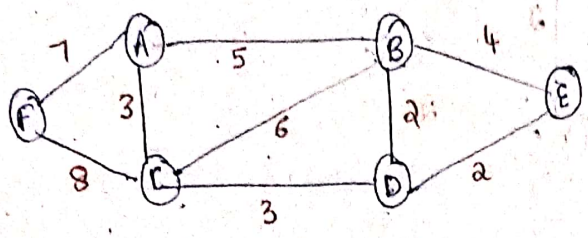
② Pick the smallest edge. Check if it forms cycle with spanning tree formed so far. If cycle is not formed, include the edges; else discard it.

③ ~~Repeat step 2 until~~ Repeat step 2 until there are (v-1) edges in spanning tree.

Eg:



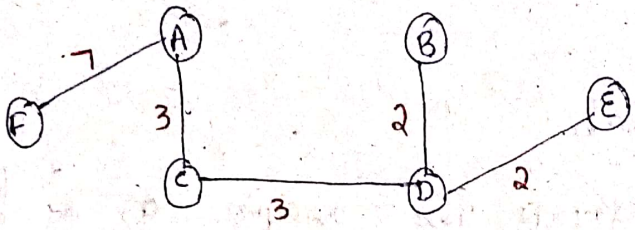
> Remaining self loops & parallel edges



> Choose the path in the ascending order of weight

BD = 2 CD = 3 BE = 4 CB = 6 FC = 8

ED = 2 AC = 3 AB = 5 FA = 7



Total cost = 17//

Job Sequencing with deadline

Given an array of jobs where every job has a deadline & ~~also~~ ~~creates~~ associated profit if the job is finished before deadline. It is also given that every job takes a single unit of time, so the maximum possible deadline for any job is n.

How to ~~maximum~~ maximize the total profit if only one job can be scheduled at a time ? ⟶

Eg:

Jobs	J_1	J_2	J_3	J_4	J_5
PROFIT	20	15	10	5	1
DEAD LINE	2	2	1	3	3

Here Maximum time is 3 hours. No job can wait after that.

So the time slot is. 0 —— 1 —— 2 —— 3

0 —— J_2 —— 1 —— J_1 —— 2 —— J_4 —— 3

J_3 & J_5 cannot be included since the slots are already filled.

JOB CONSIDER	SLOT ASSIGN	SOLUTION	PROFIT
-	-	0	0
J_1	[1, 2]	J_1	20
J_2	[0,1] [1, 2]	J_1, J_2	20 +15
X J_3	[0,1] [1, 2]	J_1, J_2	20+15
J_4	[0, 1] [1, 2] [2, 3]	J_1, J_2, J_4	20+15+5
X J_5	[0,1] [1,2] [2,3]	J_1, J_2, J_4	40 //

Algorithm

① Sort all jobs in decreasing order of profit.

② Iterate on jobs in decreasing order of profit, for each job, do the following

a) Find a time slot, i, such that the slot is empty & i < deadline and i is greatest. Put the job in this slot & mark this slot filled.

b) If no such i exists, then ignore the job.

21/01/2022

DYNAMNIC PROGRAMMING

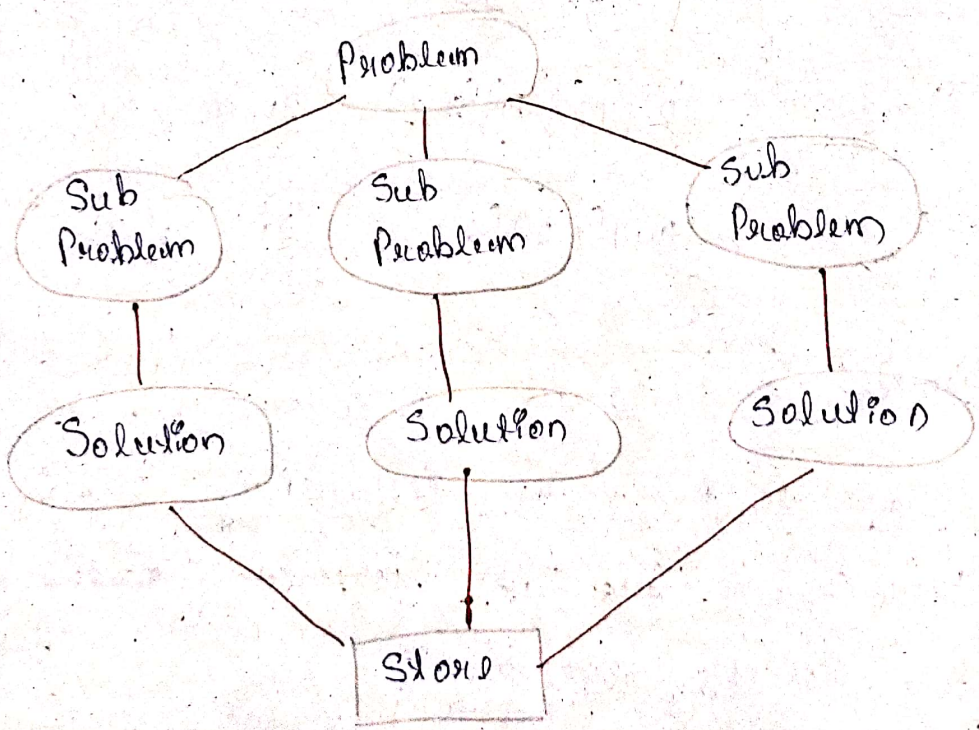
> One of the algorithm design technique.

> It is ~~can~~ used when problem breaks down into recurring smaller sub-problems.

> It is typically applied to optimisation problem.

> In such optimisation problems, there can be many solutions & each solution has a value and we ~~will~~ wish to find a solution with optimal value.

Structure:



Every dynamic problem has a Properties

① overlapping subproblem

② Optimal sub structure

① OVERLAPPING SUB PROBLEM

If the problem can be broken down into sub problems with which are reused several times rather than generating new sub problems.

② OPTIMAL SUB STRUCTURE

A given problem is said to have optimal sub structure, if the solution of the problem can be obtained from the optimal sol^n of its sub problems.

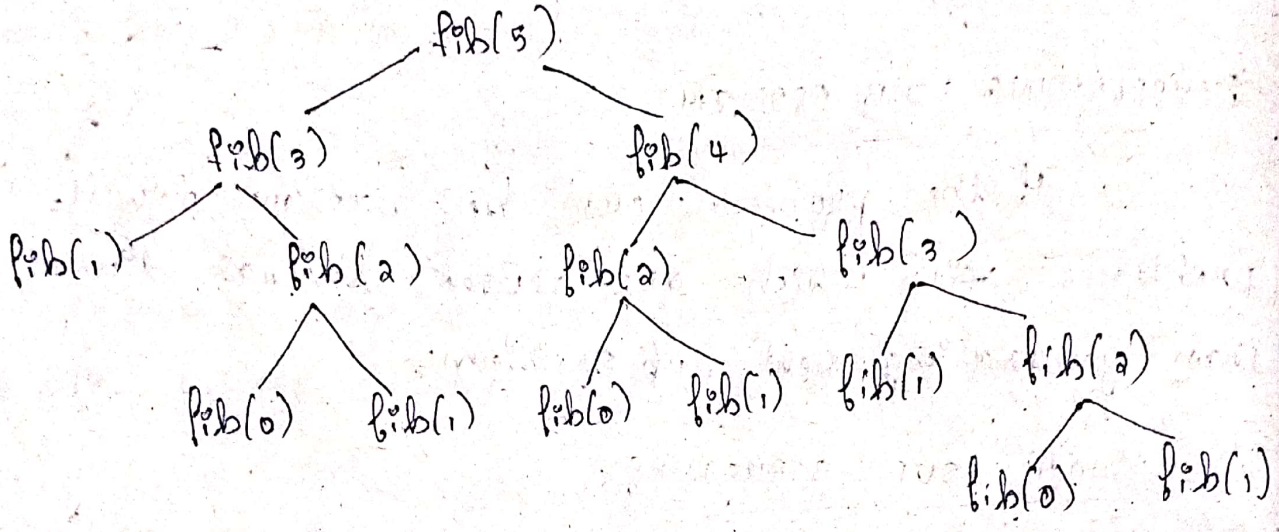
Solutions are stored in a table.

NOTE → D.P is not useful in problems which do not have overlapping sub problems.

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

```
int fib(int n)
{
  if (n<=1)
    return;
  else
    return fib(n-2) + fib(n-1);
}
```

Suppose we want to find the 5th term of the fibonacii series, the recursive tree is shown below.



complexity of this function is

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

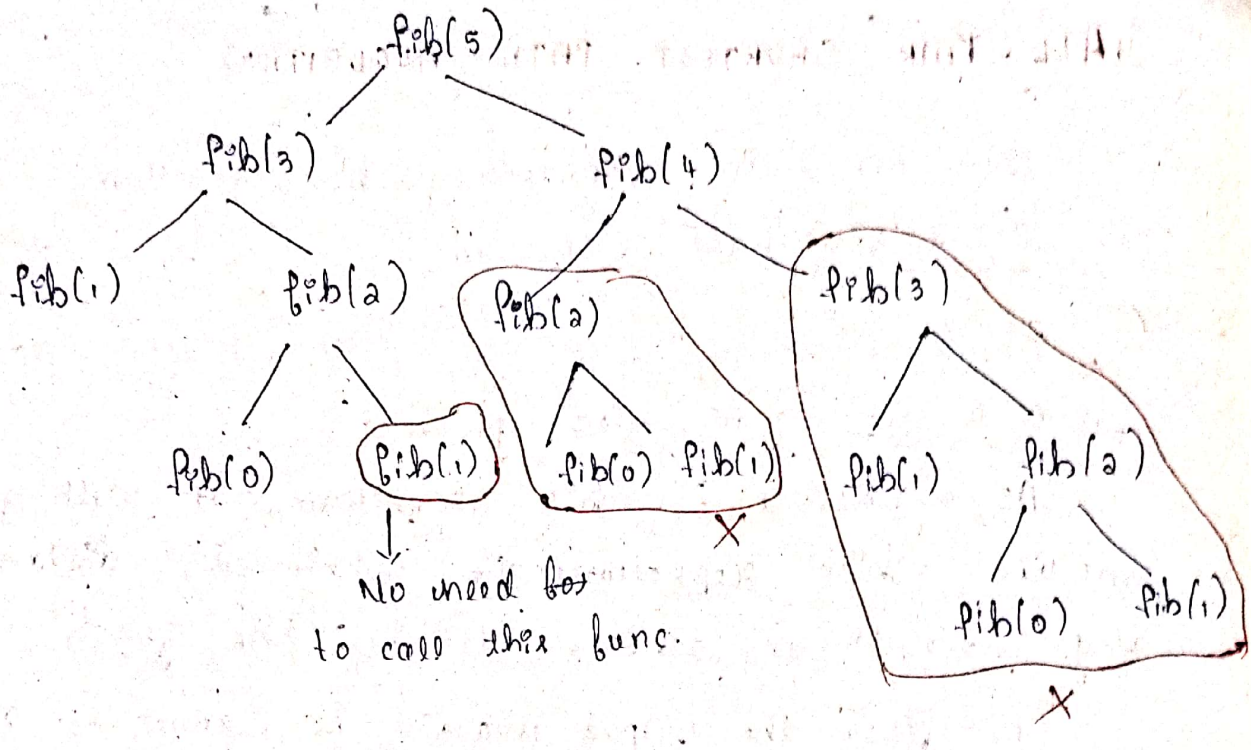
i.e, $O(2^n)$

By using D.P., we can reduce the no. of recursive calls & there by, the time required of its execution. For that we use global array for storing the sub solutions.

f	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5

Initially the values are set to -1. Whenever we get a solution for a value, it is filled & reused later

Considering the recursion tree,



F | -1 | 1 | -1 | -1 | -1 | -1 |
0 1 2 3 4 5

fib(1) = 1

F | 0 | 1 | -1 | -1 | -1 | -1 |
0 1 2 3 4 5

fib(0) = 0

F | 0 | 1 | 1 | 2 | -1 | -1 |
0 1 2 3 4 5

fib(2) = fib(0) + fib(1)
= 0 + 1 = 1 //

fib(3) = fib(1) + fib(2)
= 1 + 1 = 2 //

F | 0 | 1 | 1 | 2 | 3 | 5 |

fib(4) = fib(2) + fib(3)
= 1 + 2 = 3

fib(5) = fib(3) + fib(4)
= 2 + 3 = 5

fib(n) = n+1 calls

i.e, complexity = O(n)

Three algorithms are ① All pair shortest path algorithm
② Travelling Salesman problem
③ Bellman - Ford algorithm

25/10/22

① ALL - PAIR SHORTEST PATH ALGORITHM

The all pair shortest path algorithm is also known as Floyd - Warshall algorithm.

It is used to find all pair shortest path problem from a given weighted graph.

As a result of this algorithm, it will generate a matrix, which represents the minimum distance from any node to all other nodes in the graph.

At first, the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

The time complexity of this algorithm is $O(V^2)$, where V is the number of vertices in the graph.

Algorithm

floyd warshal (cost)

Input — The cost matrix of the given graph

Output — Matrix for shortest path b/w any vertex to any other vertex

```
Begin
    for k := 0 to n, do
    for i := 0 to n, do
    for j := 0 to n, do
        if cost[i,k] + cost[k,j] < cost[i,j] then
            cost[i,j] = cost[i,k] + cost[k,j]
```

Done

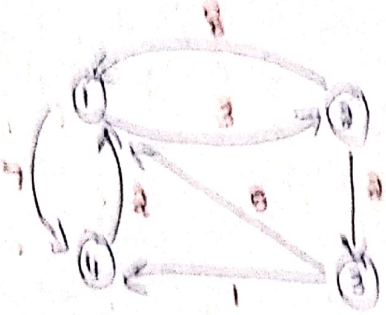
Done

Done

Display the current root navigator

Find

Eg:



A^0 =

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

A^1 =

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

A^2 =

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

$A^0[3,1] = A^0[2,1] + A^0[1,3]$

$2 < 8 + \infty$

$A^0[3,4] = A^0[3,1] + A^0[1,4]$

$\infty > 8 + 7 = 15$

$A^0[3,2] = A^0[3,1] + A^0[1,2]$

$\infty > 5 + 3 = 8$

$A^0[4,3,4] = A^0[3,4] + A^0[1,4]$

$1 < 5 + 7$

$A^0[4,2] = A^0[4,1] + A^0[1,2]$

$\infty > 2 + 3 = 5$

$A^0[4,3] = A^0[4,1] + A^0[1,3]$

$\infty > 2 + 5$

(1,2)
(1,4)
(4,2)
(4,3)

$A^3 =$

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

(1,2)

(1,4) = (1,3)(3,4)
5

(2,1) = (2,3)(3,1)
2 5

(2,4) = (2,3)(3,4)
2 - 1

(4,1) = (4,3)(3,2) = 7+5 = 12
(4,1) 7+8 = 15

(1,2) = 6+5

(1,5) = (2,1) = 0+6
7,4 = 3-
(2,3)

$A^4 =$

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

$=$

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

② TRAVELLING SALESMAN PROBLEM

Problem statement

A traveller needs to visit all the cities from a list, where distances b/w all the cities are known & each city should be visited just once. What is the shortest possible route that he visits each city exactly once & returns to the origin city?

Solution

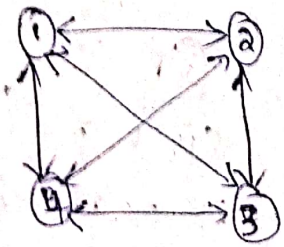
If we use brute force approach to evaluate all possible tour & select the best one;

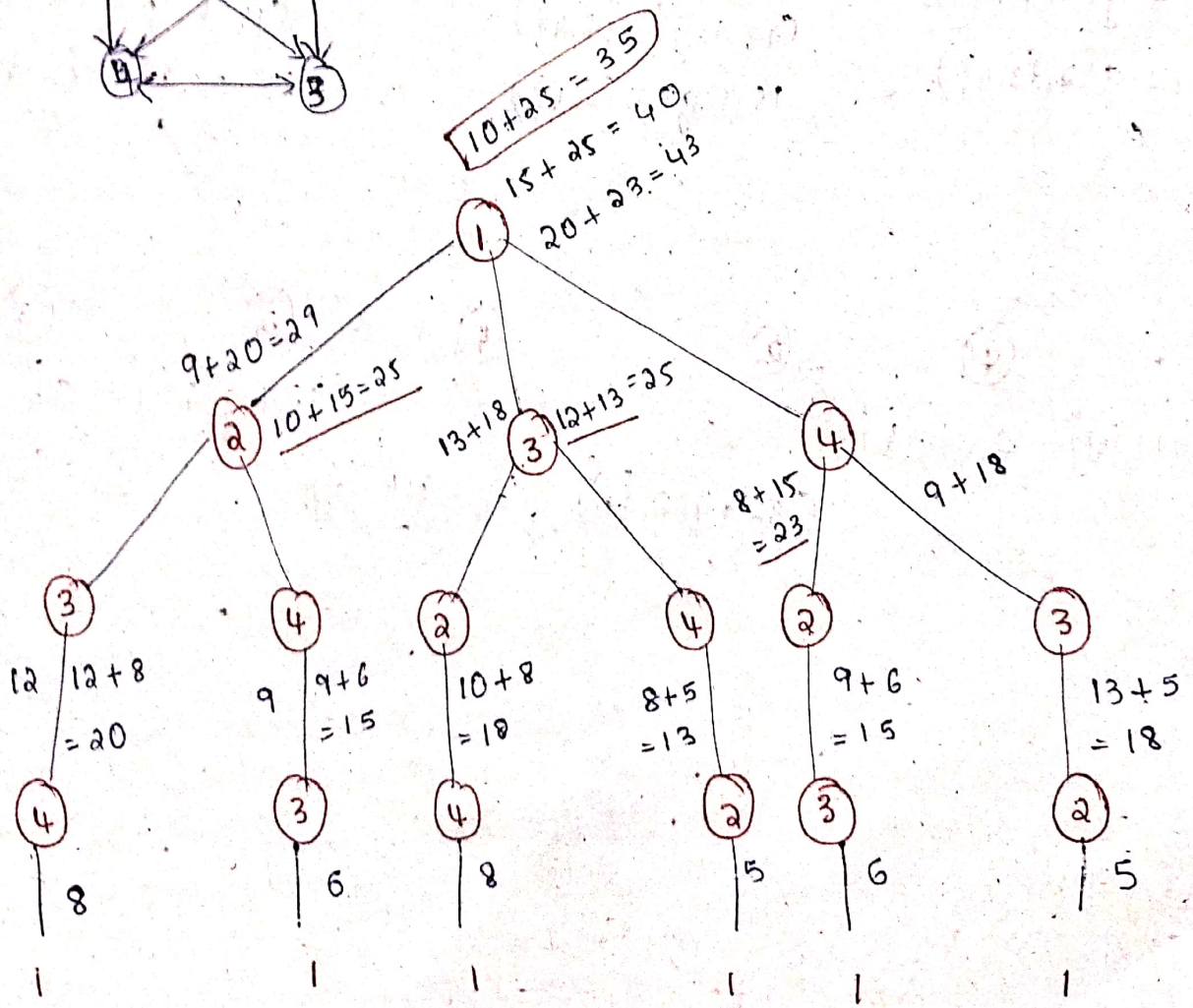
For n number of vertices in a graph, there are (n-1)! number of possibilities.

Instead of brute force, using dynamic approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Eg:

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix} \end{array} \longrightarrow \text{cost Adjacency matrix}$$





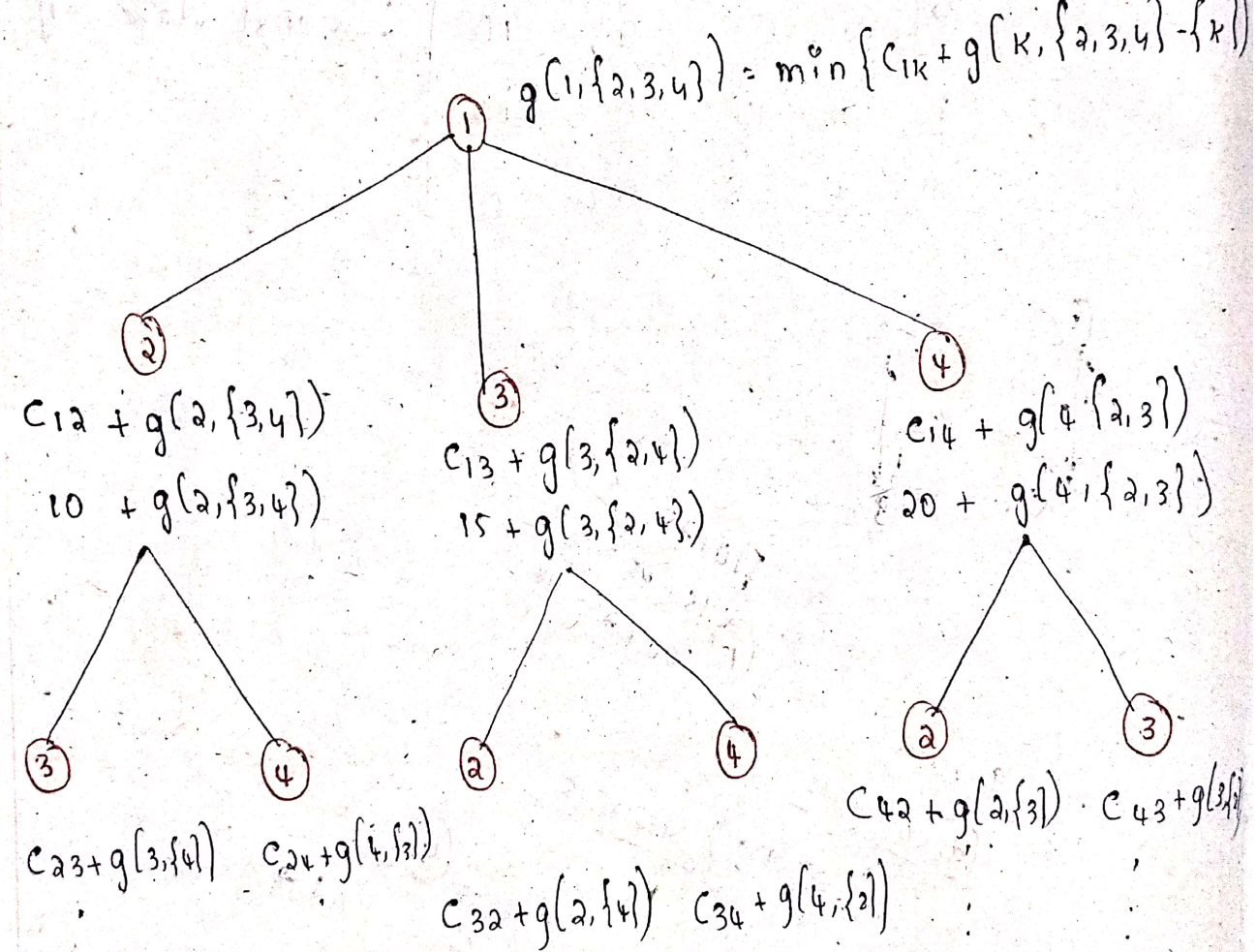
Formula: (for starting vertex 1)

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

$$K \in \{2,3,4\}$$

$$g(i,S) = \min\{ C_{iK} + g(K, S-\{K\}) \}$$

$$K \in S$$



BELLMAN-FORD ALGORITHM (Single Source shortest path)

→ To find the shortest path from source vector to other vertices in a weighted graph.

→ It will work for negative edge.

→ Slower than Dijkstra's algorithm.

→ If there are n vertices, (n-1) iterations should be performed.

Steps:-

① Select a source vertex & assign the distance of source vertex is 0 & the distance of all other vertices as infinity (∞).

② Calculate the shortest distance from u to v by checking if $(d(u) + c(u,v) < d(v))$ } : RELAXATION.
then update $d(v) = d(u) + c(u,v)$ }

③ On relaxing with all edges (n-1) times, where n = no of vertices, now we get the shortest path from source vertex to all other vertex.

Algorithm

BELLMAN-FORD (G, C, S)

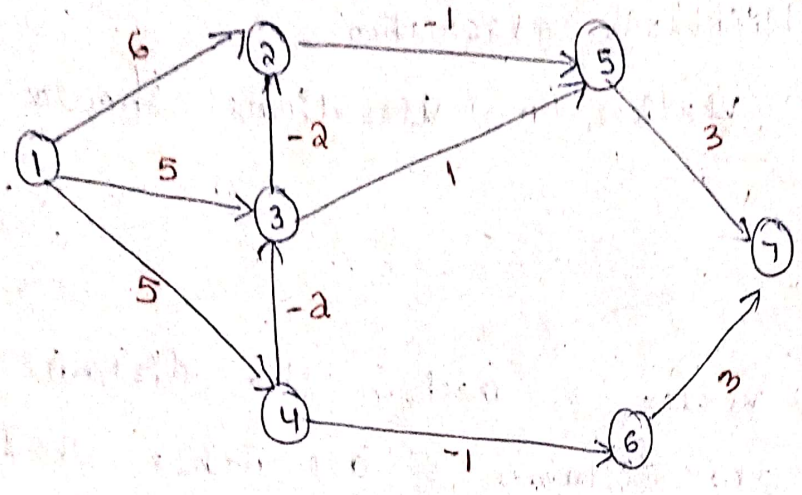
1. Initialize_single_source (G, S)
2. for i=1 to |G.V| - 1
3. for each edge (u, v) ∈ G.E
4. RELAX (u, v, c)

5. for each edge (u, v) ∈ G.E

6. if d(u) + c(u,v) < d(v)

return false

7. return true.

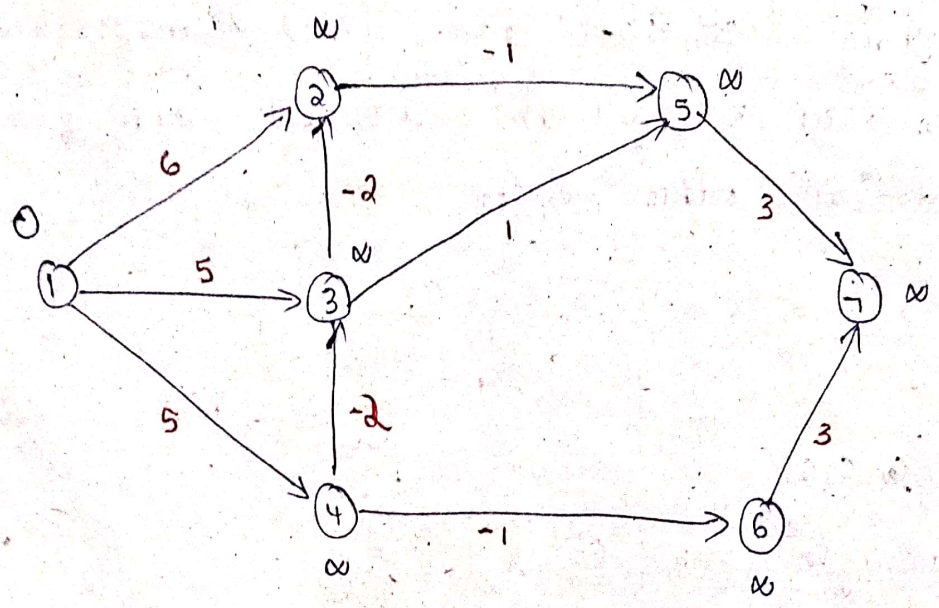


Here there are 7 vertices and therefore 6 iterations are required.

> First write all pairs of vertices

(1,2) (1,3) (1,4) ⟹ (3,2) (4,3) (2,5) (3,5) (4,6) (5,7) (6,7)

> Now relax all these edges



Make the distance of starting vertex as 0 & the remaining as ∞.

Iteration 1

(1,2)

0+6 < ∞

d(2) = 6 ✗

(1,3)

0+5 < ∞

d(3) = 5 ✗

(1,4)

0+5 < ∞

d(4) = 5

(2,5)

6-1 = 5

5 < ∞

d(5) = 5

(3,2)

5-2 = 3

3 < ∞

d(2) = 3 ✓

(3,5)

5+1 = 6

6 > 5

d(5) = 5

(4,3)

5-2 < 5

3 < 5

d(3) = 3

(5,7)

5+3 = 8

8 < ∞

d(7) = 8

(6,7)

4+3 = 7

7 < 8

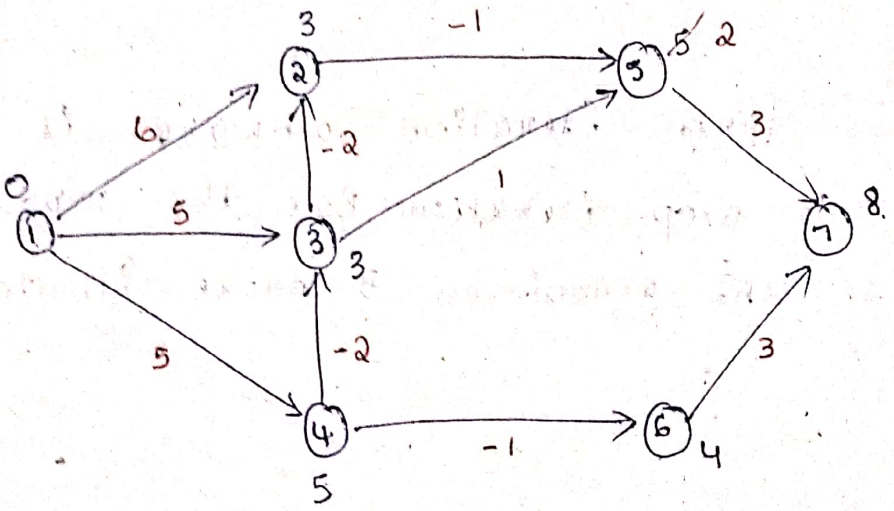
d(7) = 7

(4,6)

5-1 = 4

4 < ∞

d(6) = 4



Iteration 2

(1,2)

0+6 > 3

d(2) = 3 ✗

(1,3)

0+5 > 3

(1,4)

0+5 = 5

(2,5)

3-1 = 2

2 < 5

d(5) = 2

(3,2)

-2+3 = 1

1 < 3

d(2) = 1

(3,5)

3+1 = 4

No change

(4,3)

5-2 = 3

No change

(5,7)

2+3 = 5

d(7) = 5

(4,6)

5-1 = 4

No change

(6,7)

4+3 > 5

No change

Heration 3

(1,2)

No change

(1,3)

No change

(1,4)

No change

(2,5)

1-1 < 2

0 < 2 ∴ d(2) = 0

(3,2)

No change

(3,5)

No change

(4,5)

No change

(4,6)

No change

(5,7)

0+3 < 5

∴ d(7) = 3

(6,7)

No change

In the forth iteration onwards it stops changing. So stop iteration. But the progress will continue till the remaining 3 more iterations are completed.

Result:

d(1) = 0

d(2) = 1

d(3) = 3

d(4) = 5

d(5) = 0

d(6) = 4

d(7) = 3

Time Complexity

→ $O(\varepsilon.(|v|-1)) = O(\varepsilon \cdot v)$

→ In case of complete graph, time complexity

In complete graph, no of edges (ε) = $\frac{n(n-1)}{2}$

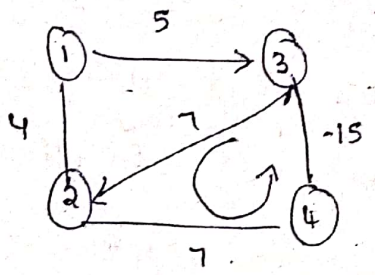
(where n is the no of vertices)

So, $O(\varepsilon v)$ becomes → $\left[\frac{n(n-1)}{2}(n-1)\right] = O(n^3)$ //

① Drawback

This algorithm will not work with graphs having cycles. If there is a negative cycle in the graph the value continue changing even after (n-1) iterations.

Eg:



28/01/2022

Module III

BACK TRACKING

→ It is one of the problem solving / algorithm design strategy.

→ Backtracking is an algorithm technique to solve a problem by incremental way.

→ It was a recursive approach to solve the problem.

→ We can say that backtracking is used to find all the possible combinations to solve an optimisation prob.

In backtracking, the problems can be categorized in three :

① DECISION PROBLEM :- Here we find whether there is any feasible solution.

② OPTIMIZATION PROBLEM :- Here we find whether there exists any best solution.

③ ENUMERATION PROBLEM :- Here we find all possible feasible solutions.

It incrementally builds candidates to the solution, & abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

Applications of backtracking

1) N-Queen's Problem
2) Graph Colouring
3) 0/1 Knapsack problem
4) Hamiltonian cycles
5) Sum of subsets.

Working of backtracking with an example

3 students → 2 boys
→ 1 Girl

3 chairs are there

* We have to arrange them in these chairs.

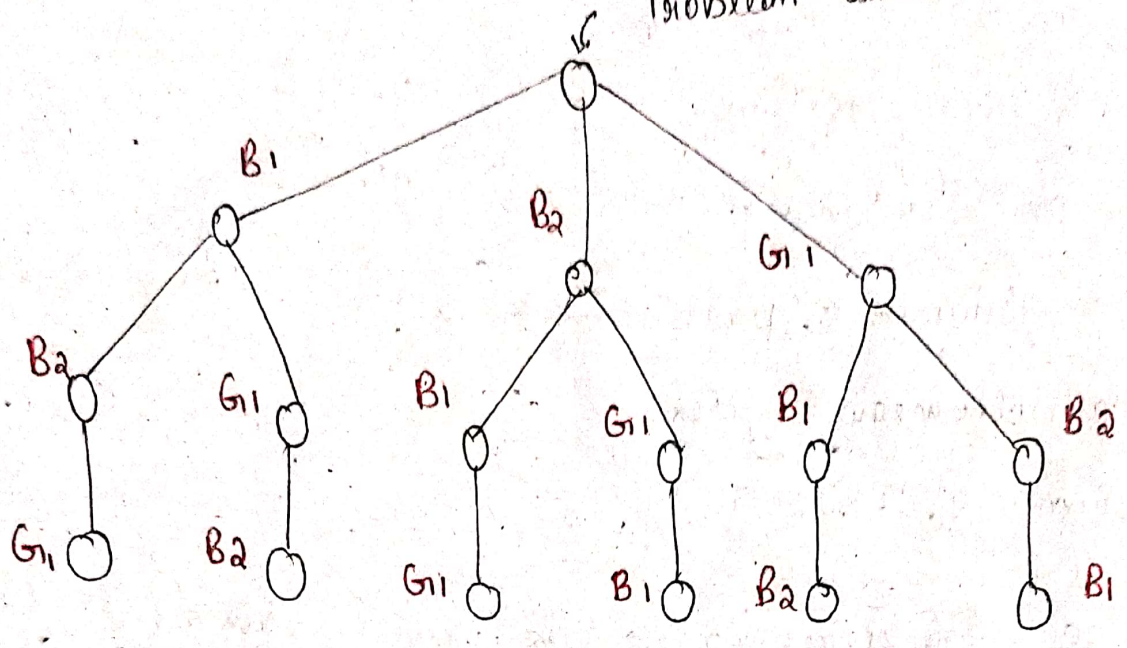
PROBLEM :- How many ways we can arrange them ?

SOLUTION :- 3! ways = 3 × 2 × 1 = 6 ways.

We can represent the solution in the form of State Space Tree.

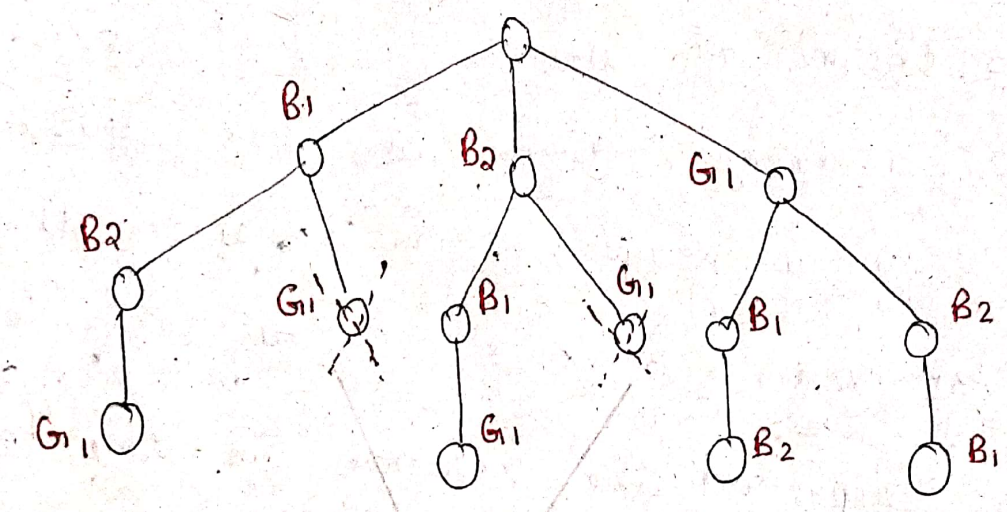
↓

A state space tree is the tree of the construction of the solution from partial solution, starting with the root with no component solution.



B1	B2	G

Next, we are applying a constraint that the girl should not be in b/w the boys.



Bounding Function

↓

It is needed to kill some live nodes without actually Expanding them.

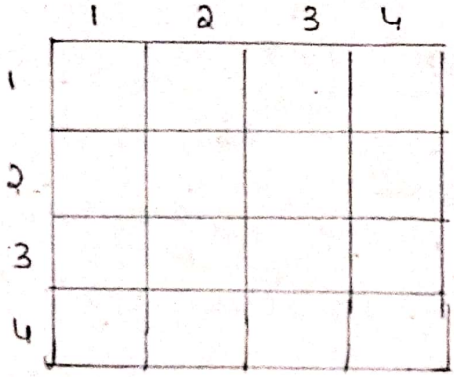
N-Queens Problem

N-Queens problem is to place n-queens in

such a manner on an $N \times N$ chessboard that no Queens attack each other by being in the same row, column or diagonal. It can be seen that for $n=1$, the problem has a trivial solution & solution doesn't exists for $n=2$ and $n=3$.

So first we will consider the 4 Queens problem.

Given a 4x4 chessboard & number the rows & columns of the chessboard 1 through 4.

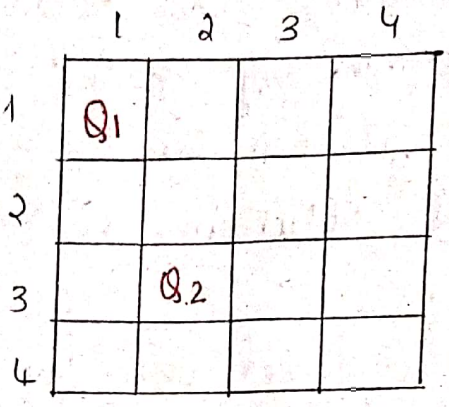


4x4 chessboard.

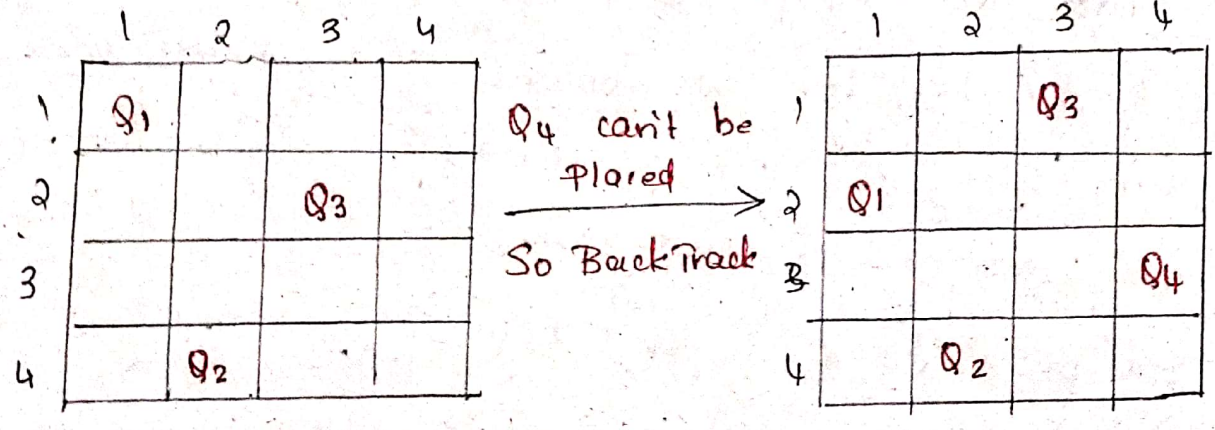
Since we have to place 4 Queens such as Q_1, Q_2, Q_3 & Q_4 on the chessboard, such that no 2 Queens attack each other. In such a condition, each Queen must be placed on a different row.

i.e, we put Queen 'i' on row 'i'

Now, we place Queen Q_1 in the very first acceptable position (1,1). (Whenever we place a Queen, increment the column & find a position for the next Queen). Next we put Queen Q_2 so that both the Queens do not attack each other.



We find that we have placed Q_2 in column 2 & row 3. Then increment the column & when we try to place Q_3, then the dead end is encountered. So we backtrack one step & replace the last placed Queen Q_2 (by incrementing row). Then we obtain the position for Q_2 in the 4th row & now we can place Q_3 in 3rd column & 2nd row. These steps are continued until all the queen are placed in the safe position.



Algorithm

1) Start in the leftmost column
2) If all queens are placed, then return true.
3) Try all row in the current column. Do following for every tried row -

a) If the Queen can be placed safely in this row, then mark this [row, column] as part of the soln & recursively check if placing queen here leads to a solution.

b) If placing the Queen in [row, column] leads to a solution then return true.

c) If placing queen doesn't lead to a solution then unmark this [row, column] (backtrack) & go to step(a) to try other rows.

d) If all rows have been tried & nothing worked, return false to trigger backtracking.

11/02/2022

SUM OF SUBSETS

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicate are presented).

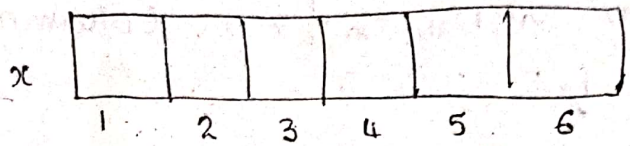
Consider the following example:

$w[1:6] = \{5, 10, 12, 13, 15, 18\}$ $\quad n=6$

We have to take subsets of those weights such that their sum total is exactly equal to a given value (here it is 30). i.e, $m=30$.

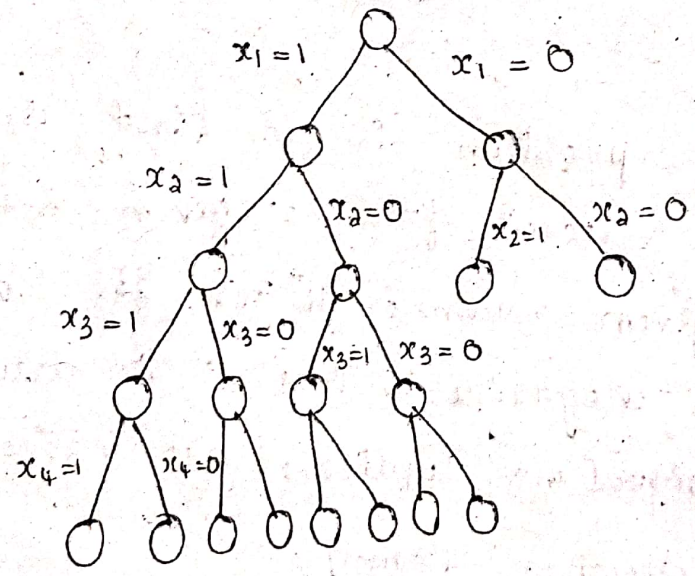
which all weights are including; we will write

the solution in an array which contains either 0 or 1 as values. i.e, If we include a value, 1 will be written in the array. & If a value is not included, then 0 will be written.



$x_i = 0/1$

The solution can be found out by using different methods. One way is to consider all possible selections & find out which of those selections is giving us then sum 30. If we draw the ~~set~~ state Space tree.

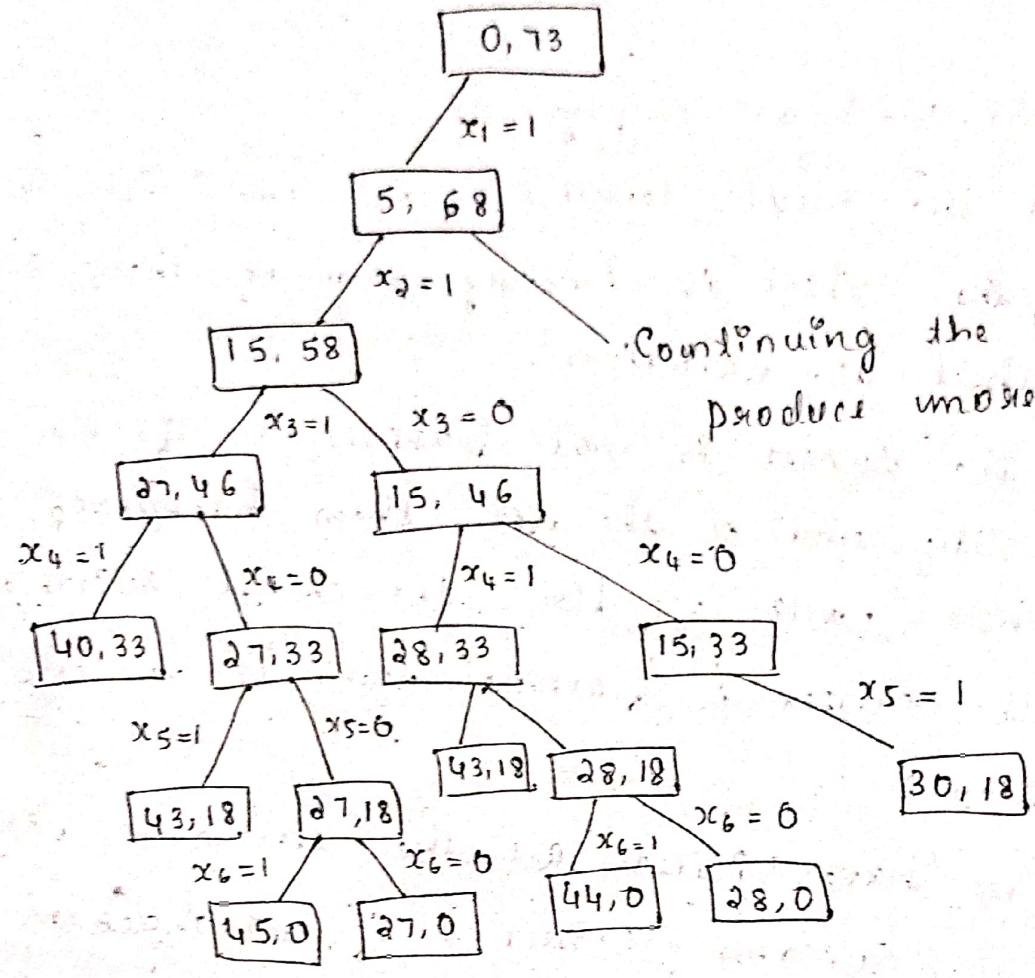


But it extremely time consuming & the complexity is 2^n.

If we apply backtracking, we have to kill the nodes by applying bounding function.

The state space tree using back tracking is

$w[1:6] = \{5, 10, 12, 13, 15, 18\}$



Continuing the process will produce more results.

Bounding functions

$$\sum_{i=1}^{K} w_i x_i + w_{K+1} \leq m$$

$$\sum_{i=1}^{K} w_i x_i + \sum_{i=K+1}^{n} w_i > m$$

Solutions x =

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

Complexity = $O(nm)$

Steps

① Start with an empty set

② Add the next element from the list to the set

③ If the subset is having sum M, then stop with that subset as solution.

④ If the subset is not feasible or if we have reached the end of the set, then backtrack through the subset until we find the most suitable value.

⑤ If the subset is feasible (sum of subset < M), then go to step 2.

⑥ If we have visited all the elements without finding a suitable subset & if no backtracking is possible then stop without solution.

Algorithm

```
Subset-sum ( list, starting-index, target-sum )
{
  if ( target-sum = sum )
  {
    subset-count++;
    if ( starting-index < list length )
    {
    Subset-sum( list, sum-list [ starting-index-1]
                        target-sum );
    }
  else
  {for(i = starting-index; i< list length; i++)
    {
```

```
        subset_sum(list, sum + list[i], target_sum);
    }
  }
}
```

3/02/2022

BRANCH AND BOUND

* A systematic method for solving optimization problems.

* It is used when the Greedy method & dynamic programming method may fail.

* Branch & bound is much slower. It often leads to exponential time complexity in the worst case.

On the other hand, if applied ~~any~~ carefully, it can lead to algorithms that run reasonably fast on average.

* The general idea of branch & bound is, it is a BFS like search for optimal solution, but not all the nodes get expanded. Rather, a carefully selected criteria determines which node to expand & when & another criteria tells the algorithm a optimal solution has been found.

* Both BFS & DFS generalizes to branch & bound strategies.

1) BFS is a FIFO search in terms of live nodes. List of live nodes are represented in the form of a Queue.

2) DFS is a LIFO like search in terms of live nodes. List of live nodes are represented in the form of a stack.

3) Least cost Search is based on minimum cost

Like backtracking, in branch & bound also we apply a bounding function to avoid searches that do not lead to a solution.

* Branch & bound refers to all the state space search method in which all the children of an E-node are generated before any other live node can become the E-node.

{

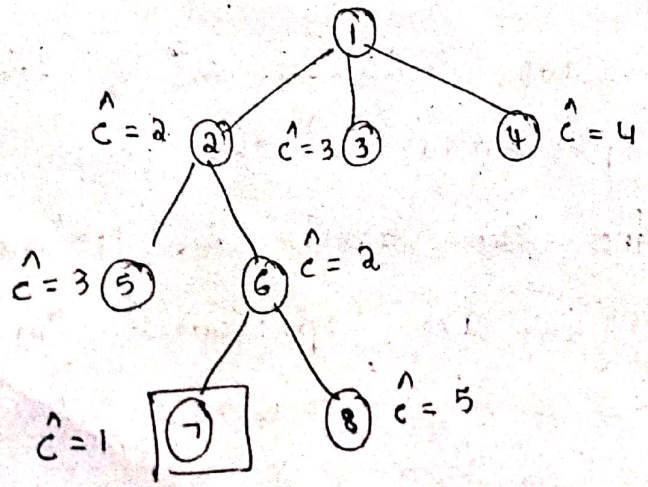
→ Live node :- A node we get generated but its children not generated.

→ E node :- A node is generated & also its children are generated.
expand

→ Dead :- The expansion of this node has no use. So, we will kill the node.

}

Eg: state space tree



ĉ = cost function

> Initially we will take node 1 as E node.

> Next generate the childeren of node 1. The childeren of node 1 are 2, 3 & 4

> By using ranking function, we will evaluate the cost of node 2, 3 & 4

$\hat{c}(2) = 2$

$\hat{c}(3) = 3$

$\hat{c}(4) = 4$

respectively.

> Now we will select a node which is minimum cost i.e, node 2. Generate the childeren of node 2, i.e, 5 & 6.

> Between 5 & 6, we will select node 6 since its cost is minimum.

> Generate the childeren of node 6 i.e, 7 & 8.

> We will select node 7, since the cost of 7 is minimum.

> ∴ cost of node 7 is the answer.

> Terminate the selection process.

> Here, E node ⟶ 1, 2, 6

Live node ⟶ 3, 4, 5, 8

* Branch & Bound method of algorith design involves:

① Tree organisation of solution space

② Use of bounding functions to limit the search.

i.e, to avoid the generation of subtrees that do not contain an answer node.

* Search techniques used in Branch & Bound

a) BFS
b) DFS
c) least cost search (LC)

* Two types of bound used in LCBB

c.a) Lower bound ($\hat{c}$)

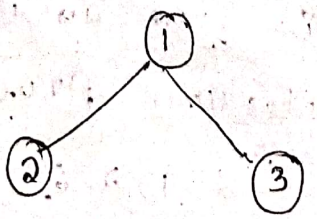
c.b) Upper bound ($\hat{u}$)

* While calculating $\hat{c}$ for a node in the state space tree, fractions are allowed.

* while calculating $\hat{u}$ for a node, fractions are not allowed.

$\hat{c}$ → fractions

$\hat{u}$ → no fractions



$\hat{c}(2) = 5$

$\hat{u}(2) = 8$

$\hat{c}(3) = 6$

$\hat{u}(3) = 7$

lowest upper bound are taken if lowest bound are equal.

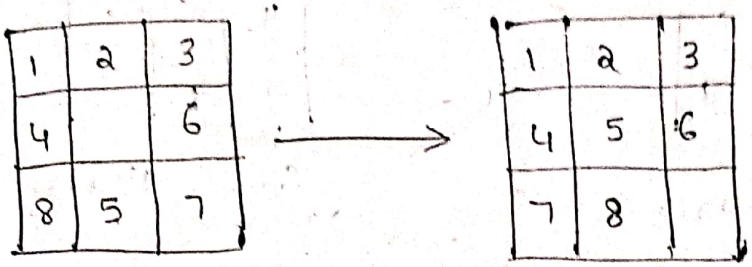
8 - puzzle Problem

→ In 8 puzzle problem, there are 8 tiles which are numbered from 1 to 8 placed on a 9 tile

capacity square frame

→ The objective of 8 puzzle problem is to transforms the arrangement of tiles from initial arrangement to a goal arrangement

→ The initial & goal arrangement is shown in below figures.



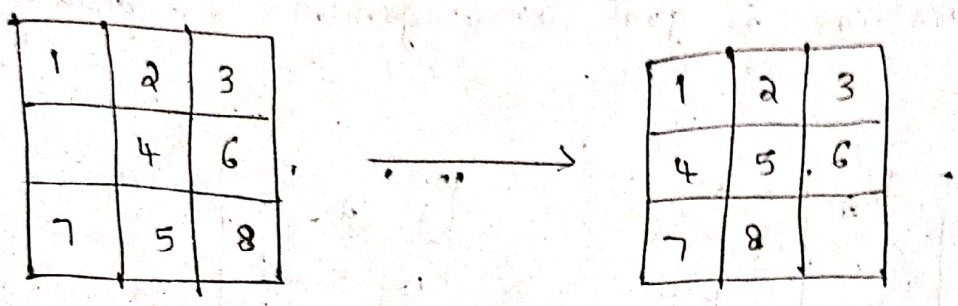
a) Initial arrangement b) Goal arrangement

- There is always an empty slot in the initial arrangement.
- Legals moves are the moves in which the tile adjacent to empty slots are moved to either left, right, up, down.
- The state space tree for 8 puzzle is large because there can be 9! different arrangements
- In state space tree, the nodes are numbered as per the level.
- Each next node is generated based on empty slot positions.
- Edges are labelled according to the direction in which empty spaces moves.
- Root node becomes the E-node.
- We can decide which node to become an

E-node based on estimation formula

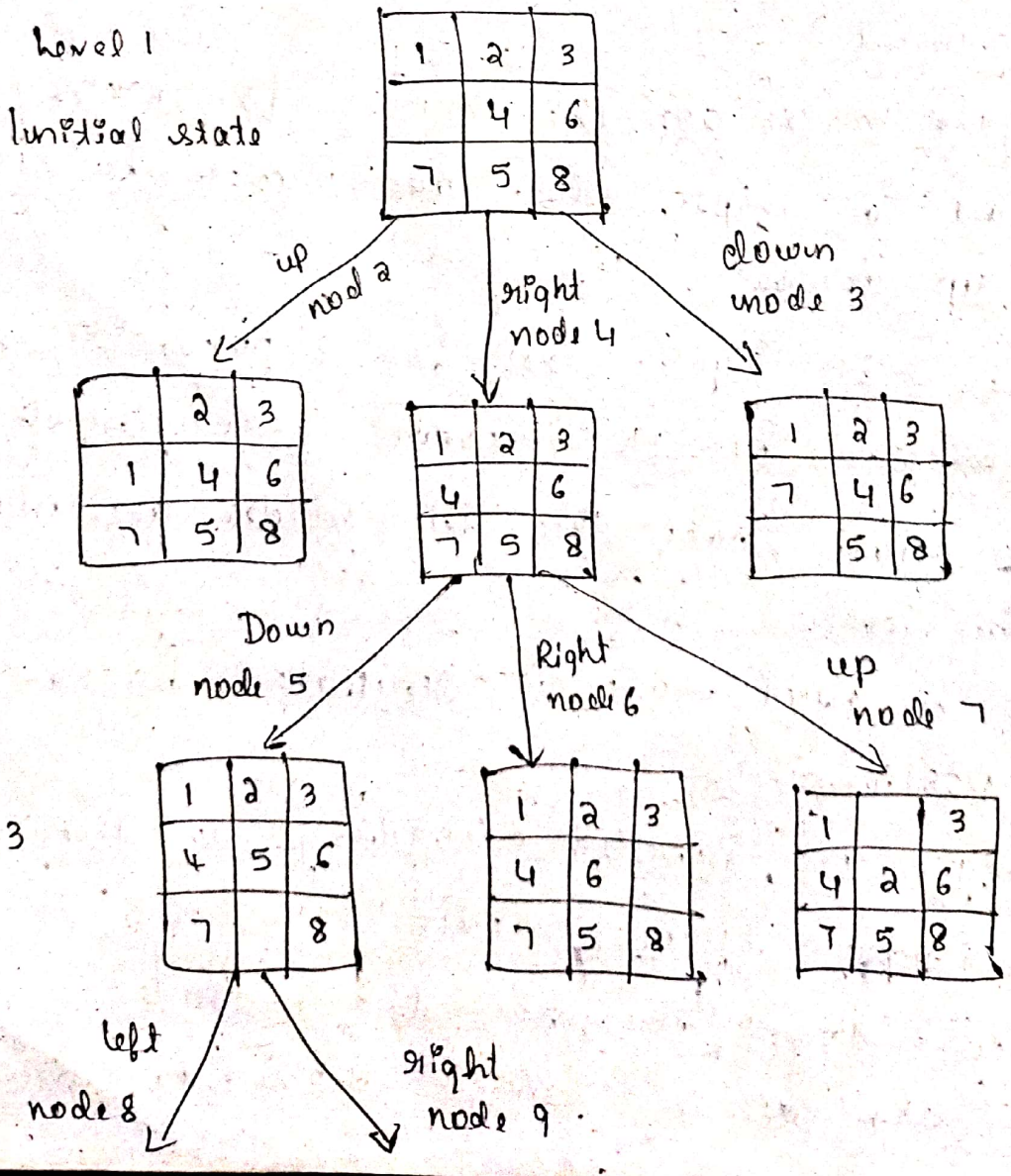
$$\hat{c}(x) = f(x) + \hat{g}(x)$$

Consider the given example

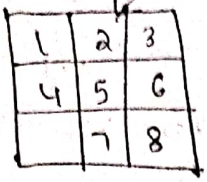


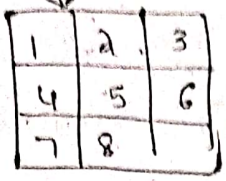
Initial state Goal State

State space tree



Level 4





Goal state

$$\hat{c}(x) = f(x) + \hat{g}(x)$$

where

$\hat{c}(x)$ = lower bound cost of node 'x'

$f(x)$ = length of the path from root node to node x

$\hat{g}(x)$ = number of tiles which are not in their goal ~~prot~~ position.

$\hat{c}(1) = 0 + 3 = 3$ $\quad$ $\hat{c}(5) = 2+1 = 3$ (expand) $\quad$ $\hat{c}(8) = 3+2 = 5$

$\hat{c}(2) = 1+4 = 5$ $\quad$ $\hat{c}(6) = 2+3 = 5$ $\quad$ $\hat{c}(9) = 3+0 = 3$

$\hat{c}(3) = 1+4 = 5$ $\quad$ $\hat{c}(7) = 2+3 = 5$

$\hat{c}(4) = 1+2 = (3)$ expand

9/2/2022

Lower Bound Theory

Lower bound theory concept is based upon the calculation of minimum time that is required to execute an algorithm.

→ Lower bound theory is used to calculate the minimum number of comparison required to execute an algorithm.

→ According to lower bound theory, for lower bound $L(n)$ of an algorithm, it is not possible to

have any algorithm (for common problem) whose time complexity is less than L(n) for random i/p.

→ Once lower bound is calculated, then we can compare it with the actual complexity of an algorithm & if their orders are same then we can declare our algorithm as optimal.

→ The techniques which are used by lower bound theory for obtaining lower bounds are:

* Decision tree (comparison tree)
* Oracle & adversary argument
* Techniques for the algebraic problem.

Decision Tree / Comparison Tree

Decision trees are the computational model useful for determining lower bounds for searching & sorting problems.

A decision tree is a full binary tree that shows the comparisons b/w elements that are executed by an appropriate sorting algorithm operating on an i/p of a given size where

* control, data movement & other conditions of the algorithm are ignored.

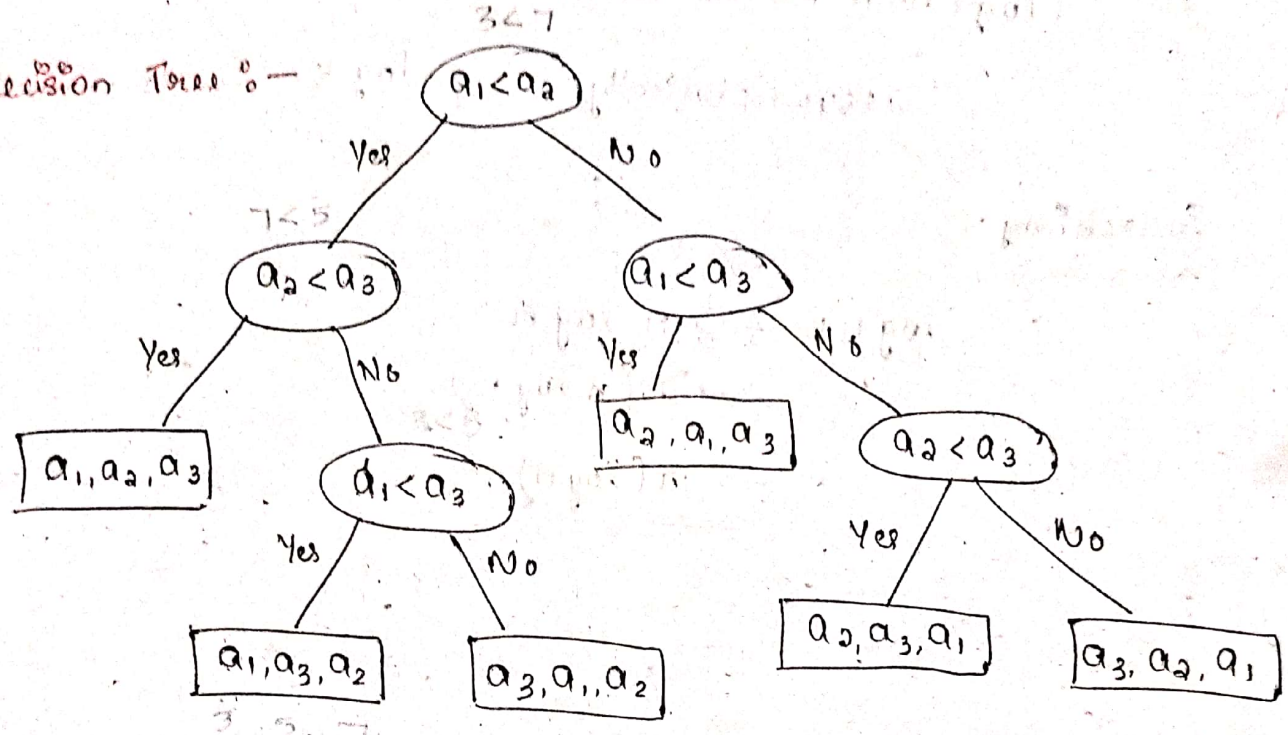
In a decision tree, there will be an array of length n. So total leaves will be n! (total no. of comparisons)

Eg: Comparison of 3 elements a_1, a_2 & a_3.

Left subtree will be true condition i.e, $a_i \leq a_j$.
Right subtree will be false condition i.e, $a_i > a_j$

Eg: $a_1 = 3$, $a_2 = 7$ & $a_3 = 5$

Decision Tree :-



In this tree all permutations are formed at leaves of tree. For n elements, there are n! permutations. If the decision tree has height h [need h comparison].

A tree with height h has atmost 2^h leaves.

Let l be number of leaves in the tree,

$n! \leq l \leq 2^h$

$\Rightarrow h \geq \log(n!)$

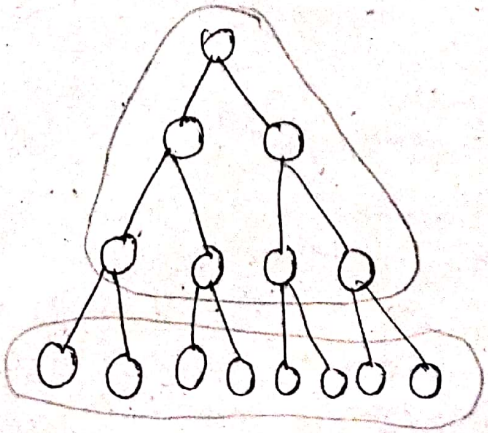
$\Rightarrow h = \Omega(n \log n)$

$n! \leq 2^h$

$h \geq \log(n!)$

$h = \Omega(n \log n)$

General case :



~~Take~~ Total No. of nodes = 2^k

Height of the tree = $\log 2^k$

i.e. (log (total no. of nodes) = $\log 2^k$

assymptotically. = $\log k$

Searching:

$\log n! \Longrightarrow n \log n$

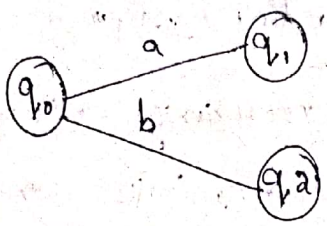
$\Longrightarrow \Omega(n \log n)$

$\Rightarrow \Omega(\log n)$

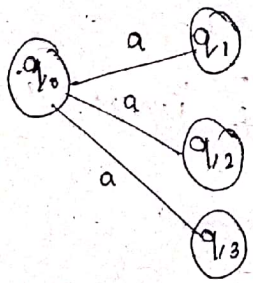
Module 4

Non-deterministic Algorithm

An algorithm in which every operation is uniquely defined is called deterministic algorithm.



An algorithm in which every operation may not have unique result, instead there can be specified set of possibilities for every operation. Such an algorithm is called non-deterministic algorithm.



Non deterministic means that no particular rule is followed to make the guess.

Non-deterministic algorithm is a two stage algorithm.

1. Non deterministic (Guessing) stage
2. Deterministic (verification) stage.

→ Non deterministic (Guessing) stage

> In this stage, generate an arbitrary string that can be thought of as a candidate solution.

→ Deterministic (verification) stage.

In this stage, candidate solution & instance of the problem are given as input if it returns "Yes" the candidate solution represents the actual solution.

A non-deterministic algorithm terminates unsuccessfully if & only if there exist no set of choices leading to a successful solution/ signal.

A machine capable of executing non-deterministic algorithm is called non-deterministic machine (NFA - non-deterministic finite automata)

algorithm

```
Algorithm Non-deterministic()
// A[1:n] is a set of elements
// we have to deterministic the index 'i' of A at which
element is located.
{
The following for loop is for guessing stage
for i:=1 to n do
    A[i] = choice()
// Next is verification stage
if (A[i]==x) then
{
    write(i);
    Success();
}
    write(0);
    Success failure();
}
```

In the above given non-deterministic algorithm, there are 3 functions used

1. choice()
2. Failure()
3. Success()

(1) choice() : arbitrarily choose one of the elements from the given input set.

(2) failure() : It indicate the unsuccessful completion

(3) Success() : It indicate successful completion.

The above non-deterministic search algorithm complexity is O(1) whereas the deterministic search algorithm time complexity is O(n) where Array A is not ordered.

10/2/2023

X. P-NP-Hard and NP complete

The problems are classified into two group: first one consist of the problem that can be solved in polynomial time by using deterministic algorithm.

Eg: Searching of an element from an array - O(log n) Worst

Searching of given n elements - O(n log n)

All pairs shortest path problem - $O(n^3)$.

Second one consist of the problem that can be solved in polynomial time by using non-deterministic algorithm

Eg: 0/1 Knapsack problem - $O(2^n)$

Travelling Salesman Problem — $O(a^n)$

Sum of subset — $O(a^n)$

P-class Problem

Problems that can be solved in polynomial time are called P class Problem where P stands for "polynomial time.

Linear Search — n

Binary Search — $\log n$

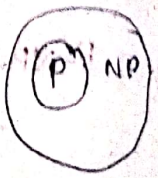
Quick Sort — n^2

Merge Sort — $n \log n$

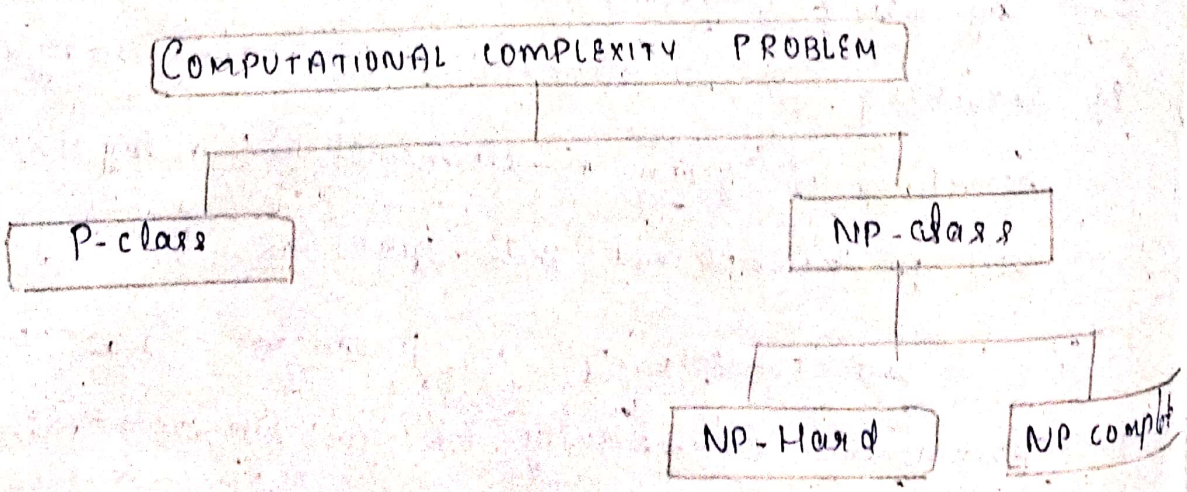
Matrix Multiplication — n^3

NP-class Problem

Problems that can be solved in non-deterministic polynomial time is called NP class problem.



Relationship b/w P & NP class Problem



* Every problem which is a P class is also in NP-class.

* Every problem which is a NP-class is not in P-class.

* P class problem can be solved efficiently, NP class problem cannot be solved efficiently as P-class Problem.

NP-Complete Problem

- A problem L, is NP complete if & only if it satisfies two conditions.

① L_1 is in NP
i.e, $L_1 \in NP$

② Every problem L_2 in NP is polynomial time reducable to L_1.

i.e, Determining whether a graph has hamiltonian cycle.
Determining whether a boolean formula is satisfactory

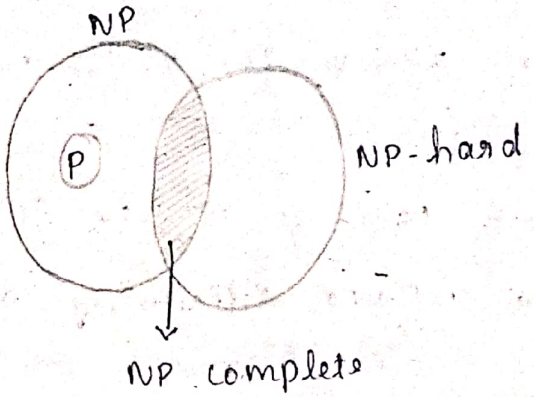
A problem L is **NP Hard** if and only if satisfiability reduces to L.

Eg: Set Tower problem
Vertex Tower problem
Travelling salesperson

Satisfiability $\propto$ L

Let L_1 & L_2 be two problem. Problem L_1 reduces to L_2 (symbolic representation) if and only if there is a way to solve L_1 by a deterministic algorithm in polynomial time, then solve L_2 in polynomial time.

Relation b/w P, NP-Hard, NP, NP-Complete Problems



Cook's theorem:

Satisfiability is in P if and only if P = NP.