

# DESIGN AND ANALYSIS

Complete Notes

SAVE AND SHARE WITH YOUR CONNECTIONS TO HELP THEM

**Created by:- Bhabani Shankar Pradhan**

(Linkedin)

<https://www.linkedin.com/in/bhabanishankarpradhan07/>

## Design & Analysis of Algorithm

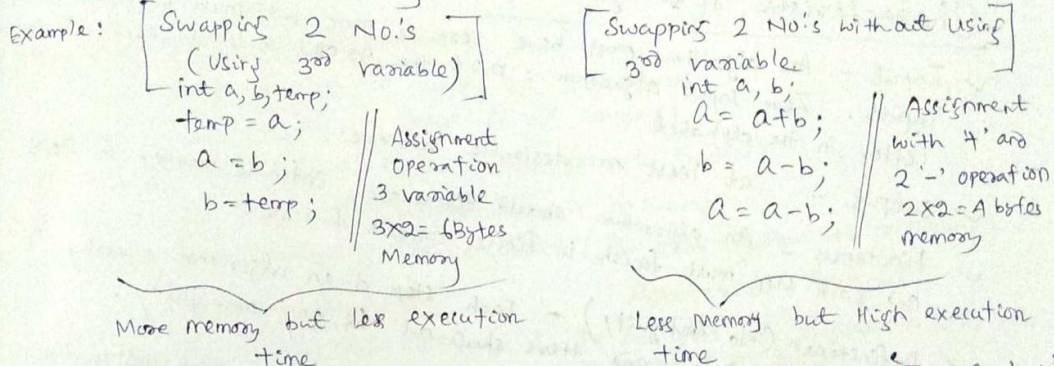
Definition: It is a sequence of computational steps that transform the input into output.

- ✓ It is an efficient method that can be expressed within finite amount of time and space.
- ✓ It is a step by step solution of a specific mathematical or computer related problem.
- ✓ Independent of prog. language.

### Pseudocode

- ✓ It is the statement in plain English which may be translated later into a programming language.

Imp Time  $\downarrow$  [ if execution time is more than memory consumption is less ]



Memory day by day less price, so we have to more focus on "Time Complexity"

### Algorithm Application

- ✓ Merge sort, Quick sort, Heap sort  
(Gmail chronological display of emails in inbox)
- ✓ Fourier Transform  
(Transform signal from their time domain to frequency domain & vice-versa)
- ✓ Dijkstra's Algorithm  
(shortest path between two nodes)
- ✓ RSA Algorithm  
(Cryptography, Cyber security)
- ✓ Secure Hash algorithm  
(prevent phishing attack)
- ✓ Link analysis [ Relation and connection between network nodes ]
  - Example: Google Page Rank
  - Facebook news feed.
  - Facebook friend suggestion
  - LinkedIn suggestion for jobs.
  - Youtube video suggestion
- ✓ Human Genome Project  
(100,000 genes), Data analysis
- ✓ quickly access & retrieve large amount of information  
(Search Engine Optimization)
- ✓ E-commerce (online transactions)
- ✓ public key cryptography, digital signatures

- ✓ Data Compression [Cloud Computing, Database Storage]
- ✓ Random Number generation [Lottery, Card games, Cryptography)

#### Books:

- Introduction to Algorithms, T.H. Cormen, 2<sup>nd</sup> Ed.
- Design & Analysis of Algorithm, S. Saha, Oxford University Press.

#### Online Source:

- TutorialsPoint.com // tutorial on DAA
- GeeksforGeeks

#### Android Apps:

- ✓ Design and analysis of algorithm complete - Engg. wale baba.

### Characteristic of an Algorithm (5)

- ✓ Input - An algorithm must have zero or more but must be finite no. of inputs. Zero Input algorithm : print the ASCII code of each of the letters in the alphabet.
- ✓ Output - at least one desirable outcome.
- ✓ Finiteness - An algorithm should terminate infinite number of steps and each step must finish in finite amount of time.
- ✓ Definiteness (No Ambiguity) - Each step of an algorithm clearly & precisely define and there should not be any ambiguity.
- ✓ Effectiveness - It must be possible to perform each instruction in a finite amount of time.

Example of Non-effectiveness:

$$e = \frac{1}{1!} + \frac{1}{(2!)^2} + \frac{1}{(3!)^3} + \dots \text{ (infinite terms)}$$

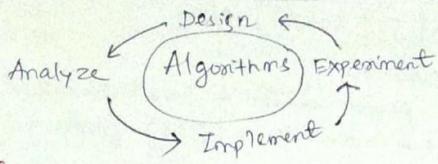
Definition of Algorithm basing upon the above characteristics

- It is a collection of finite sequence of definite and effective instructions which terminates with the production of correct output from the given input after a finite time."

### Classification of Algorithm

- ✓ Deterministic Algorithm - give a particular input, will always produce the same output.
- ✓ Non-Deterministic - even for the same input can exhibit different behaviors on different runs.
- ✓ Sequential/ Serial - Computer execute one instruction of an algorithm at a time.
- ✓ Parallel - to process several instructions at once.

## A Major stages of Algorithm Analysis and Design



## Life cycle of an algorithm

1. Designing the algorithm (Using different Techniques)
2. Writing the algorithm (Using pseudocode)
3. Testing the algorithm (Using validation Techniques)
4. Analyzing the algorithm (Analysis)
 

(To analyze an algorithm, there is need to determine how much Time an algorithm takes when implemented and how much Space it occupies in primary memory)

## How to analyze an algorithm

1. Time (How much time is taking) - Time function
2. Space (How much memory space it will consume)
3. Network consumption (How much data going to transfer)
4. Power consumption (How much power is consuming)
5. System level programming / Device drivers (How many register it's consuming)

## Apriori Analysis

1. We do analysis (Space and time) of an algorithm prior to running it on specific system. [Industry]

2. Algorithm
3. Independent of language
4. Hardware independent
5. Time and space function

## Aposteriori Analysis

1. Do analysis of algorithm only after running it on system.
2. Program
3. Language dependent
4. Hardware dependent
5. watch time and bytes.

## Growth of functions

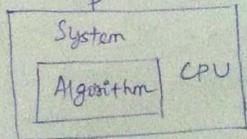
- Growth functions are used to estimate the number of steps an algorithm uses as its input grows.

## Algorithm Complexity

To describe running time of algorithms, we consider the size of input is very large, so we can ignore constants in describing running time.

divide the I/P with 3 subsets

- ↳ Running time less [It is the function defined by the minimum number of steps taken on any instance of size  $n$ ] [Best]
- ↳ Running time more [Maximum no. of steps] [Worst]
- ↳ Running time average [Average no. of steps] [Average]



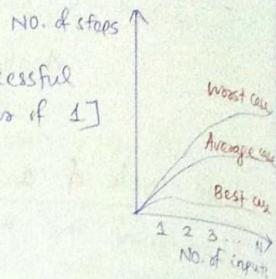
Best case  $\leq$  Avg. case  $\leq$  Worst case

Ex:

### Linear Search (N-element Array)

|    |    |    |    |     |    |    |    |      |
|----|----|----|----|-----|----|----|----|------|
| 40 | 10 | 20 | 80 | 100 | 50 | 60 | 70 | A[n] |
|----|----|----|----|-----|----|----|----|------|

- if  $X = 40$ 
  - 1 - comparison to perform search - search successful
  - Best case = 1 no. of comparison =  $O(1)$  [order of 1]
- if  $X = 60$ , last element
  - Worst case =  $n$  no. of comparison =  $O(n)$
- if  $X = 100$ ,
  - Average case =  $\frac{n}{2}$  no. of comparison =  $O(\frac{n}{2}) \approx O(n)$



### Asymptotic Analysis

These are the languages that allow us to analyze an algorithm running time by identifying its behaviour as the input size of the algorithm increases.

Order of Growth: It is described by the highest degree term of the formula for running time. (Drop lower order terms. Ignore the constant coefficient in the leading term)

$$\text{Ex: } f(n) = an^2 + bn + c \quad [\text{Drop lower order terms}]$$

$$f(n) = an^2 \quad [\text{Ignore constant coefficient}]$$

$$f(n) = n^2 \text{ i.e. } O(n^2)$$

### Asymptotic notation

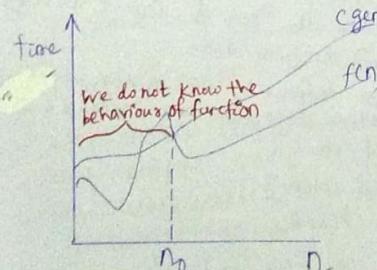
- It is a way to describe the characteristics of a function in the limit
- It describes the rate of growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- It is a way to compare "sizes" of functions:
  - 1. Big-oh  $O \leq$
  - 2. Big-Omega  $\Omega \geq$
  - 3. Big-Theta  $\Theta =$
  - 4. Small-oh  $o \ll$
  - 5. Little-omega  $\omega \gg$

### 1. Big-oh ( $O$ ) Notation : (Asymptotic Upper Bound)

$O(g(n)) = \{f(n)\}$ ; there exist positive constants  $c$  and  $n_0$  such that  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0$

$$g(n) \quad \text{Ex: } f(n) = 2n+3$$

$n_0$  = minimum possible value or greater value



$g(n)$  is an asymptotic upper bound for  $f(n)$

$$2n+3 \leq 2n+3n$$

$$2n+3 \leq 5n, n=1, O(n) \quad [\text{Nearest}]$$

$$5 \leq 5$$

Also write

$$2n+3 \leq 3n \rightarrow O(n) \quad [\text{Nearest}]$$

$$2n+3 \leq 2n^2 + 3n^2 - O(n^2) \quad \left\{ \begin{array}{l} \text{Not nearest} \\ \text{one} \end{array} \right.$$

$$2n+3 \leq 2^n - O(2^n)$$

$$f(n) \neq O(\log n)$$

Lower Bound

Classes & Functions:

$$\begin{array}{c} 1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \dots < 2^n < 3^n < n^k \\ \text{(constant)} \quad \text{(log)} \quad \text{(linear)} \quad \text{(log)} \quad \text{(Quadratic)} \quad \text{(cube)} \quad \text{(exponential)} \end{array}$$

for ex: 1, Lower Bound      Upper Bound.

Ex-2: Is  $2n^2 = O(n^3)$

$$2n^2 \leq Cn^3, c=2, n_0=2$$

$8 \leq 1 \cdot 8$  (satisfied)

so  $2n^2 = O(n^3) \checkmark$

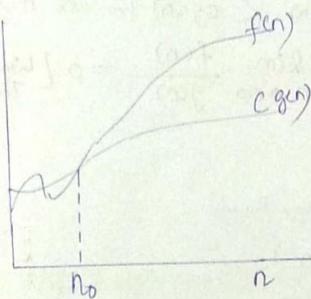
$$\begin{aligned} An^3 + 10n^2 + 5n + 1 &\leq An^3 + 10n^3 + 5n^3 + n^3 \\ An^3 + 10n^2 + 5n + 1 &\leq 19n^3 \text{ (satisfied)} \end{aligned}$$

i.e.  $f(n) = O(n^3)$   
or  $f(n) \leq 5n^3$ .

Note: if  $f(n)$  is  $O(g(n))$  i.e.  $f(n)$  grows no faster than  $g(n)$

2. Big-Omega ( $\Omega$ ) notation: (Asymptotic Lower Bound)

$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0 \}$$



$g(n)$  is an asymptotic lower bound for  $f(n)$

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

$$2n+3 \geq 1 \cdot n, n=1$$

$$5 \geq 1 \cdot 1, f(n) = \Omega(n) \text{ [Nearest one]}$$

$$2n+3 \geq \log n, f(n) = \Omega(\log n) \quad \begin{cases} \text{Not} \\ \text{Nearest} \end{cases}$$

$$2n+3 \geq \sqrt{n}, f(n) = \Omega(\sqrt{n}) \quad \begin{cases} \text{Nearest} \\ \text{Not} \end{cases}$$

$$2n+3 \geq \log \log n, f(n) = \Omega(\log \log n)$$

$$f(n) \neq \Omega(n^2)$$

$\underbrace{\text{Upper Bound.}}$

Ex-2: Is  $\sqrt{n} = \Omega(\log n)$

$$\sqrt{n} \geq c \log n, c=1, n_0=16$$

$$\sqrt{16} \geq 1 \log_2 16$$

$$4 \geq \log_2 4^2$$

$$4 \geq 4 \log_2 2 \quad (\text{satisfied})$$

so  $\sqrt{n} = \Omega(\log n)$

$$An^3 + 10n^2 + 5n + 1 \geq An^3$$

$$An^3 + 10n^2 + 5n + 1 \geq 1n^3$$

(satisfied)

$$f(n) = \Omega(n^3)$$

Note: if  $f(n)$  is  $\Omega(g(n))$  i.e.  $f(n)$  grows no slower than  $g(n)$

3. Big-Theta ( $\Theta$ ) notation: (Asymptotic tight Bound)

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive element constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}$$

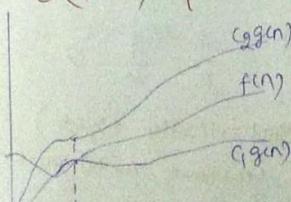
$f(n) = 2n+3$ , Ex-1

$$1n \leq 2n+3 \leq 2n+3n$$

$$\frac{1}{c_1}n \leq 2n+3 \leq \frac{c_2}{c_1}n \quad - f(n) = \Theta(n)$$

(it shows that exactly equal to  $n$ )

$$\times f(n) = \Theta(n^2), f(n) = \Theta(1 \cdot n)$$



$g(n)$  is an asymptotically tight bound for  $f(n)$

Note: we can use any notations for best case, worst case.

$$\text{Ex-2: } n^2/2 - 2n = \Theta(n^2), \quad c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$\frac{1}{4} \times n^2 \leq n^2/2 - 2n \leq \frac{1}{2} \times n^2$$

$$\frac{1}{4} \times 8^2 \leq \frac{8^2}{2} - 2 \times 8 \leq \frac{1}{2} \times 8^2$$

$$\frac{1}{4} \times 64 \leq \frac{64}{2} - 16 \leq \frac{1}{2} \times 64$$

$$16 \leq 16 \leq 32. \quad (\text{satisfied})$$

$$n^2/2 - 2n = \Theta(n^2)$$

$$n_0 = 8, c_1 = 1/4, c_2 = 1/2$$

$$f(n) = 4n^3 + 10n^2 + 5n + 1, g(n) = n^3$$

$$4 \cdot g(n) \leq f(n) \leq 5 \cdot g(n)$$

$$\frac{4 \cdot n^3}{c_1 g(n)} \leq 4n^3 + 10n^2 + 5n + 1 \leq \frac{5 \cdot n^3}{c_2 g(n)}$$

$$(\text{satisfied}) \quad (\text{satisfied})$$

$$4n^3 + 10n^2 + 5n + 1 = \Theta(n^3)$$

Note: if  $f(n)$  is  $\Theta(g(n))$  i.e.  $f(n)$  and  $g(n)$  grows at the same rate

4. Small-oh ( $o$ ) Notation (Upper Bound that is not asymptotically tight) [Loose Upper-bound]

$o(g(n)) = \{ f(n) : \text{for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < c g(n) \text{ for all } n \geq n_0\}$

Another view, probably easier to use  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$  [limit theorem]

$n^2 = O(n^2)$  - Tightest Upper Bound.

$n^2 = O(n^3)$  - Non Tightest Upper Bound.

Small-oh notation will not give Tightest Upper Bound.

$$n^2 = o(n^2) \times \quad 2n^2 \neq o(n^2)$$

$$n^2 = o(n^3) \checkmark \quad 2n = o(n^2) \checkmark$$

$$\text{Ex-1: } f(n) = n, g(n) = n^2 \quad \lim_{n \rightarrow \infty} \frac{n}{n^2} \Rightarrow \lim_{n \rightarrow \infty} \frac{1}{n} = \frac{1}{\infty} = 0$$

$$n = o(n^2) \checkmark$$

$$\text{Ex-2: } \text{Is } \log n = o(n), f(n) = \log n, g(n) = n$$

$$\lim_{n \rightarrow \infty} \frac{\log n}{n} \quad (\frac{\infty}{\infty} \text{ form, use L'Hopital's Rule i.e. } \frac{f'(n)}{g'(n)} \text{ derivative})$$

$$\lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{1} = \frac{1}{n} = 0$$

$$\log n = o(n) \checkmark$$

Note: if  $f(n)$  is  $o(g(n))$  i.e.  $f(n)$  grows slower than  $g(n)$

$$\text{Ex-3: Is } 4n^3 + 10n^2 + 5n + 1 = o(n^4)$$

$$\lim_{n \rightarrow \infty} \frac{4n^3 + 10n^2 + 5n + 1}{n^4} = \frac{n^3(4 + \frac{10}{n} + \frac{5}{n^2} + \frac{1}{n^3})}{n^4} = \frac{4}{n} = \frac{4}{\infty} = 0$$

5. Little-omega ( $\omega$ ) Notation (Lower Bound that is not asymptotically tight) [Loose Lower bound]

$\omega(g(n)) = \{ f(n) : \text{for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq c g(n) < f(n) \text{ for all } n \geq n_0\}$

$$\text{Another view, } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

$n^3 = \Omega(n^3)$  — Tight Test Lower Bound.

$n^3 = \Omega(n^2)$  — Non Tight Test Lower Bound.

Little-omega notation will not give tight test lower bound.

$$n^3 \neq o(n^3), n^3 = \omega(n^2) \checkmark$$

$$\text{Ex-1: } f(n) = 3^n, g(n) = 2^n \quad \lim_{n \rightarrow \infty} \frac{3^n}{2^n} = \lim_{n \rightarrow \infty} \left(\frac{3}{2}\right)^n = \lim_{n \rightarrow \infty} (1.5)^{\infty} = \infty$$

$$3^n = \omega(2^n)$$

$$\text{Ex-2: } f(n) = n^2, g(n) = \log n \quad \lim_{n \rightarrow \infty} \frac{n^2}{\log n} \left( \begin{array}{l} \text{form L'Hopital's Rule} \\ \text{find derivative} \end{array} \right)$$

$$\lim_{n \rightarrow \infty} \frac{2n}{1} \Rightarrow \lim_{n \rightarrow \infty} 2n^2 \Rightarrow 2\infty^2 \Rightarrow \infty.$$

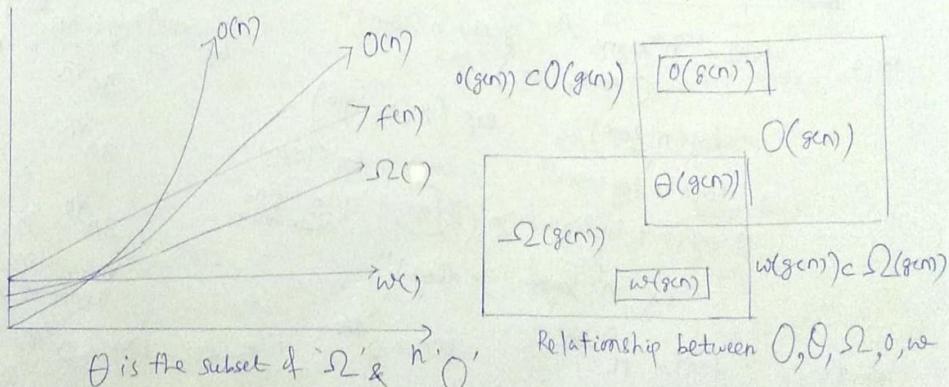
$$n^2 = \omega(\log n)$$

$$\text{Ex-3: Is } 4n^3 + 10n^2 + 5n + 1 = \omega(n^2) ?$$

$$\lim_{n \rightarrow \infty} \frac{4n^3 + 10n^2 + 5n + 1}{n^2} = \lim_{n \rightarrow \infty} \frac{4n^2(1 + \frac{10}{n} + \frac{5}{n^2} + \frac{1}{n^3})}{n^2} = 4\infty^2 = \infty$$

$$4n^3 + 10n^2 + 5n + 1 = \omega(n^2)$$

Note: if  $f(n)$  is  $\omega(g(n))$  i.e  $f(n)$  grows faster than  $g(n)$ .



$$(i) O(g) \cup \Theta(g) \subset O(g)$$

$$(ii) \omega(g) \cup \Theta(g) \subset \Omega(g)$$

$$(iii) \Theta(g) = \Omega(g) \cap O(g)$$

$$\text{Q) Is } n^2 \log n + n = O(n^2 \log n)$$

$$\text{Q) Is } 2n^2 + 3n + 4 = \Theta(n^2) ?$$

$$1n^2 \leq 2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$$

$$1 \cdot n^2 \log n \leq n^2 \log n + n \leq 10 n^2 \log n$$

$$\begin{aligned} 1n^2 &\leq 2n^2 + 3n + 4 \leq 9n^2 \\ 1 \cdot g(n) &\leq f(n) \leq 9g(n) \\ \text{i.e. } &\Theta(n^2) \end{aligned}$$

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

$$\text{i.e. } O(n^2 \log n)$$

$$Q) f(n) = n! = n \times (n-1) \times (n-2) \dots (3 \times 2 \times 1 \text{ or } \dots)$$

$$1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \leq n! \leq n \cdot n \cdot n \cdot n \dots$$

$$1 \leq n! \leq \frac{n^n}{n!} \quad (\text{Both sides same function not present})$$

$\Omega(1)$  we can find  $\Theta(n^n)$   
 Lower Bound or Tight Bound Upper Bound.  
 [can not fix a particular value]

$$Q) f(n) = \log n!$$

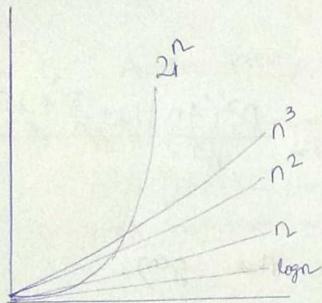
$$\log(1 \times 1 \times 1 \times 1 \dots) \leq \log(1 \times 2 \times 3 \dots n) \leq \log(n \times n \times n \times \dots)$$

$$0 \text{ means constant so take it as 1} \leq \log n! \leq \log n^n$$

$$\Omega(1) \quad 1 \leq \log n! \leq n \log n \quad O(n \log n)$$

not tight bound.

Compare class of functions:



$$\text{formula: 1. } \log ab = \log a + \log b$$

$$2. \log \frac{a}{b} = \log a - \log b$$

$$3. \log a^b = b \log a$$

$$4. a^{\log b} = b^{\log a}$$

$$5. a^b = n, \text{ then } b = \log_a n$$

$$\text{Ex-1: } f(n) = n^2 \log n \quad g(n) = n(\log n)^{10} \quad (\text{Apply log})$$

$$\begin{aligned} & \log(n^2 \log n) & \log(n(\log n)^{10}) \\ & \log n^2 + \log \log n & \log n + \log(\log n)^{10} \\ & (2 \log n) + \log \log n & \log n + 10 \log \log n \\ & \text{ie } n^2 \log n > n(\log n)^{10} & \text{small value} \end{aligned}$$

$$\text{Ex-3: } f(n) = n^{\log n} \quad g(n) = 2^{\sqrt{n}}$$

$$\begin{aligned} & \log n^{\log n} & \log 2^{\sqrt{n}} \\ & \log n \log n & \sqrt{n} \log 2 \\ & \log^2 n & n^{\sqrt{n}/2} \\ & \text{ie } n^{\log n} < 2^{\sqrt{n}} & \text{(Apply log again)} \\ & 2 \log \log n & \frac{1}{2} \log n \\ & \text{ie } n^{\log n} < 2^{\sqrt{n}} & \end{aligned}$$

$$\text{Ex-2: } f(n) = 3n^{\sqrt{n}} \quad g(n) = 2^{\sqrt{n} \log n}$$

$$\begin{aligned} & [\text{Also apply log}] \log(3n^{\sqrt{n}}) & \log(2^{\sqrt{n} \log n}) \\ & \log 3 + \sqrt{n} \log n & \log 2 + \sqrt{n} \log \log n \\ & 3n^{\sqrt{n}} & 2^{\sqrt{n} \log n} \\ & (n^{\sqrt{n}})^{\log 2} & \text{informed} \end{aligned}$$

$$3n^{\sqrt{n}} > n^{\sqrt{n}}$$

value wise greater but asymptotically equal.

$$\text{Ex-4: } f(n) = 2^{\log n} \quad g(n) = n^{\sqrt{n}}$$

$$\begin{aligned} & (\text{Apply log}) \quad \log n & \log 2^{\sqrt{n}} \\ & \log \log 2^{\sqrt{n}} & \sqrt{n} \log n \\ & \log n & < \sqrt{n} \log n \end{aligned}$$

$$\text{ie } 2^{\log n} < n^{\sqrt{n}}$$

$$\begin{aligned} & \text{Ex-5: } f(n) = 2^n \quad g(n) = 2^{\sqrt{n}} \\ & (\text{Apply log}) \quad \log 2^n & \log 2^{\sqrt{n}} \\ & n \log 2^n & 2n \log 2^{\sqrt{n}} \\ & \text{ie } n < 2n, 2^n < 2^{\sqrt{n}} \end{aligned}$$

1.  $(n+k)^m = \Theta(n^m)$        $(n+3)^2 = \Theta(n^2)$  Highest degree = 2      correct ✓

2.  $2^{n+1} = O(2^n)$        $2 \cdot 2^n$  (co-efficient is ignorable),  $\Theta(2^n)$ ,  $\Theta(2^n)$ ,  $\Omega(2^n)$

3.  $2^{2n} = O(2^n)$        $4^n > 2^n$ , incorrect ✓ ,

4.  $\sqrt{\log n} = O(\log \log n)$        $\frac{1}{2} \log \log n > \log \log \log n$ , incorrect.  
 (apply log)

5.  $n^{\log n} = O(2^n)$  (apply log)       $\log n \log n < n \log_2^2 n$ , correct

Compose :

$$\textcircled{1} \quad 3^n > 2^n \\ (\text{apply log}) \\ \log 3 > \log 2$$

$$\textcircled{2} \quad 2^n > n^2$$

$$n \log_2 2 - 2 \log n \quad \text{i.e. } 2^n > n^2$$

(Substitute big no. of  $n$ )

$$n \quad 2 \log n$$

$$n = 2^{100}$$

$$2^{100} \quad 2 \log_2 100 \quad \text{i.e. } 2^{100} > 2 \times 100 \log_2 2 =$$

$$\textcircled{3} \quad n^2 > n \log n$$

$n \times n \quad n \log n$

$$\text{A)} \quad n > (\log n)^{100}$$

|   | $n^{\log n}$       | $n \log n$                                  | $\log 2$ | $100 \log \log 2$              | $100 \log 28$                        | $128 < 700$                             |
|---|--------------------|---|----------|--------------------------------|--------------------------------------|---|
| 5 | $\log \log n$      | $\log n + \log \log n$                      | 128      | $100 \log \log 2$              | $100 \log 28$                        | $128 < 700$                             |
|   | $N = 2^{1024}$     |   |          |                                |                                      | we take for large values $N = 2^{1024}$ |
|   | $\log^3 2^{1024}$  | $\log 2^{1024} + \log \log 2^{1024} \log 2$ | 1024     | $100 \log \log 2^{1024}$       |                                      |   |
|   | $1024 \times 1024$ | $1024 + (\log 1024)$                        | 1024     | $100 \log \log 2^{1024}$       |                                      |   |
|   | $1024 \times 1024$ | $\cancel{1024} + \cancel{(\log 1024)}$      | 1024     | $100 \log 1024 \Rightarrow 10$ |                                      |   |
|   | $1024 \times 1024$ | $\cancel{1024} + \cancel{(\log 1024)}$      | 1024     | $100 \log 1024 \Rightarrow 10$ | $\frac{1024}{\text{(growing more)}}$ | $1024 > 1000$                           |

$$n = \frac{2^{20}}{2^2}$$

— 1 —

(6) Vogn > logogn

(7)  $\sqrt{n} \log n$  (apply log again)

$$\begin{array}{ccc} \textcircled{6} & \text{Vlogn} & \text{loglogn} \\ \frac{1}{2} \log 2^n & \log \log 2^{10} & n = 2^{10} \\ \frac{1}{2} \log \log 2^{2^{10}} & \log \log \log 2^{\frac{1}{2}} & \end{array}$$

$$\frac{1}{2} \log_2 \frac{\log_2 n}{\log_2 2} \quad \log_2 \log_2 \frac{n}{\log_2 2} \quad f(n) = \int_{2^k}^n k^3 \quad 0 < n < 10,000$$

$$\frac{1}{2} \times 10 \log_2^{\frac{n}{2}} n \geq 10,000$$

3.5      ;       $g(n) = \begin{cases} n & n \geq 100 \\ -2 & 0 < n < 100 \end{cases}$

| $n^3$  | $n^3$     | $n^2$ | $\infty$ |
|--------|-----------|-------|----------|
| 0 - 99 | 100 - 999 | 1,000 |          |

|        |       |       |                |  |
|--------|-------|-------|----------------|--|
| $f(n)$ | $n^2$ | $n^3$ | $n^3$          |  |
| $g(n)$ | $n$   | $n^3$ | $n^3$          |  |
|        |       |       | $n_0 = 10,000$ |  |

from  $10,000$  onwards  $f(n)$  becomes larger as compared  
i.e.  $f(n) = O(g(n))$ ,  $n_0 = 10,000$   $f(n)$

(8)  $f_1 = 2^n, f_2 = n^{3/2}, f_3 = n \log n, f_4 = n^{\log n}$  (compare function)

Ans:  $f_1 > f_4 > f_2 > f_3$

Limit Theorem

$$\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)}$$

|  
 $c > 0$   
 $T(n) = O(f(n))$   
 $T(n) = \Theta(f(n)) - 0 < c < \infty$   
 $T(n) = \Omega(f(n)) - c < \infty$   
 $T(n) = \omega(f(n)) - c > 0$

(1) Show that any real constant  $a & b$ , where  $b > 0$

$$(n+a)^b = \Theta(n^b)$$

$$\lim_{n \rightarrow \infty} \frac{(n+a)^b}{n^b}, \quad \lim_{n \rightarrow \infty} \frac{n^b \left(1 + \frac{a}{n}\right)^b}{n^b} = 1, \text{ i.e. } c > 0$$

i.e.  $(n+a)^b = \Theta(n^b)$

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

$$\lim_{n \rightarrow \infty} \frac{2^{n+1}}{2^n} \Rightarrow \lim_{n \rightarrow \infty} \frac{2 \times 2^n}{2^n} = 2 \text{ constant } c > 0$$

i.e.  $2^{n+1} = O(2^n)$

(3) Is  $n! = \omega(2^n)$   $n! = n \cdot (n-1) \cdot (n-2) \dots 3 \cdot 2 \cdot 1$

$$\lim_{n \rightarrow \infty} \frac{n!}{2^n} \Rightarrow \frac{n}{2^n} \cdot \frac{n-1}{2^n} \cdot \frac{n-2}{2^n} \dots \frac{3}{2^n} \cdot \frac{2}{2^n} \cdot \frac{1}{2^n}$$

$$\lim_{n \rightarrow \infty} \frac{n^n \left(1 - \frac{1}{n}\right)^n \left(1 - \frac{2}{n}\right)^n \dots \left(1 - \frac{3}{n}\right)^n \dots \left(1 - \frac{1}{n}\right)^n}{2^n} = \infty$$

i.e.  $n! = \omega(2^n)$

(4) Is  $n! = o(n^n)$

$$\lim_{n \rightarrow \infty} \frac{n!}{n^n} \Rightarrow \lim_{n \rightarrow \infty} \frac{n^n \left(1 - \frac{1}{n}\right)^n \left(1 - \frac{2}{n}\right)^n \dots \left(1 - \frac{3}{n}\right)^n \dots \left(1 - \frac{1}{n}\right)^n}{n^n}$$

$$\Rightarrow \lim_{n \rightarrow \infty} (1-0)(1-0)\dots 0 \cdot 0 \cdot 0 \Rightarrow 0, \quad n! = o(n^n)$$

Stirling's Formula:

Stirling's approximation gives an approximate value for the factorial  $n!$ .

Stirling's approximation gets better as the number  $N!$  gets larger.

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

$$\ln(n!) = \ln 1 + \ln 2 + \dots + \ln n \quad \text{ie} \quad \sum_{k=1}^n \ln k \asymp \int_1^n \ln x dx$$

$$= [\ln x - x]_1^n$$

$$= n \ln n - n + 1$$

$$\asymp n \ln n - n \quad \text{ie } O(n \ln n)$$

formulae:

$$\int u dv = uv - \int v du$$

or. Stirling's approximation:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right)$$

$$n! \asymp \left(\frac{n}{e}\right)^n$$

(apply log)

$$\log n! = \log \left(\frac{n}{e}\right)^n \quad [ \log b = b \log a ]$$

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

$$= n \log n - n \log e$$

$$\log n! = O(n \ln n)$$

### Properties of Asymptotic Notations :

General Properties if  $f(n)$  is  $O(g(n))$  then  $a + f(n)$  is also  $O(g(n))$  [constant]

$$\text{Ex: } f(n) = 2n^2 + 5 \text{ is } O(n^2)$$

$$f(n) = 2n^2 + 5 = 14n^2 + 35 = O(n^2) \text{ same degree.}$$

∴ true for  $\Omega$  and  $\Theta$  also.

Reflexive if  $f(n)$  is given then  $f(n)$  is  $O(f(n))$

$$\text{Ex: } f(n) = n^2, O(n^2) \quad [\text{same function upper bound of itself}]$$

(Relation on self) [same function lower bound of itself]

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

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

Transitive if  $f(n)$  is  $O(g(n))$  and  $g(n)$  is  $O(h(n))$  then  $f(n) = O(h(n))$

$$\text{Ex: } f(n) = n, g(n) = n^2, h(n) = n^3$$

$$n = O(n^2), n^2 = O(n^3), \text{ then } n = O(n^3)$$

∴ true for  $\Omega, \Omega, \Theta, O$  too.

Symmetric if  $f(n) = O(g(n))$  then  $g(n) = O(f(n))$ , true only for  $\Theta$ .

$$\text{Ex: } f(n) = n^2, g(n) = n^2$$

$$\begin{cases} f(n) = \Theta(n^2) \\ g(n) = \Theta(n^2) \end{cases} \quad \text{Symmetric to each other.}$$

Transpose Symmetric if  $f(n) = O(g(n))$  then  $g(n) = \Omega(f(n))$

$$\text{Ex: } f(n) = n, g(n) = n^2, n = O(n^2) \& n^2 = \Omega(n)$$

Here for  $n^2$ ,  $n$  is lower bound & for  $n^2$ ,  $n^2$  is the upper bound.

also true for  $f(n) = O(g(n))$  iff  $g(n) = \omega(f(n))$

Note: if  $f(n) = O(g(n))$  &  $f(n) = \Omega(g(n))$   
 $g(n) \leq f(n) \leq g(n)$ ,  $f(n) = \Theta(g(n))$   
[same function at both]

Note: if  $f(n) = O(g(n))$ ,  $d(n) = O(e(n))$

then  $f(n) + d(n) = ?$  let  $f(n) = n$ ,  $d(n) = n^2 = O(n^2)$

$$\Rightarrow O(n) + O(n^2) \Rightarrow O(n^2) \checkmark \text{ [Leading term taken]}$$

so we can say  $O(\max(g(n), e(n)))$

Note: if  $f(n) = O(g(n))$ ,  $d(n) = O(e(n))$ , let  $f(n)=n$ ,  $d(n)=n^2$

$$f(n) + d(n) = ? f(n) + d(n) = n + n^2 = n^3 = O(g(n) \cdot e(n))$$

Recurrences: Recursion is generally expressed in terms of recurrences.

When an algorithm call to itself, we can often describe its running time by recurrence equation which describes the overall running time of a problem of size  $n$  in terms of running time.

### Various Techniques to solve Recurrences

1) Substitution Method:

We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

2) Recurrence Tree Method:

We draw a recurrence tree and calculate the time taken by every level of tree, finally, we sum the work done at all levels.

3) Master Theorem Method: only for following types of recurrences or for recurrences that can be transformed to following type.

$$T(n) = aT(n/b) + f(n), \text{ where } a \geq 1 \text{ and } b \geq 1$$

### Recursive function (decreasing)

Ex-1 void Test(int n) — T(n)

{ if ( $n > 0$ )  
  may take 1 unit or may not  
  (bcz ignore constant)  
  { printf("%d\n", n); — 1 unit  
    Test(n-1); —  $T(n-1)$  unit  
  }  
}

From this example recurrence relation  
time we don't write '0', constant  
 $T(n) \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$

### Substitution Method

$$T(n) = T(n-1) + 1, T(n-1) = T(n-2) + 1 \quad | T(n-1) = T(n-2) + 1 |$$

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

$$T(n) = \underbrace{T(n-2)}_{\text{Substitute}} + 2$$

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

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

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

... continue for K times

$$T(n) = T(n-k) + k, \text{ we know when } n=0, \text{ result} = 1$$

Assume that reached at zero, i.e.  $n-k=0, n=k$

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

$$T(n) = T(0) + n$$

$$T(n) = 1+n \quad \text{i.e. } T(n) = O(n)$$

Ex-2:

void Test (int n) ————— T(n)

{ if (n>0) ————— 4 unit

{ for(i=0; i<=n; i++) ————— n+1 unit

{ printf("%d.%d", n); ————— n unit

}

Test(n-1); ————— T(n-1) + time

}

Add all the unit time.

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

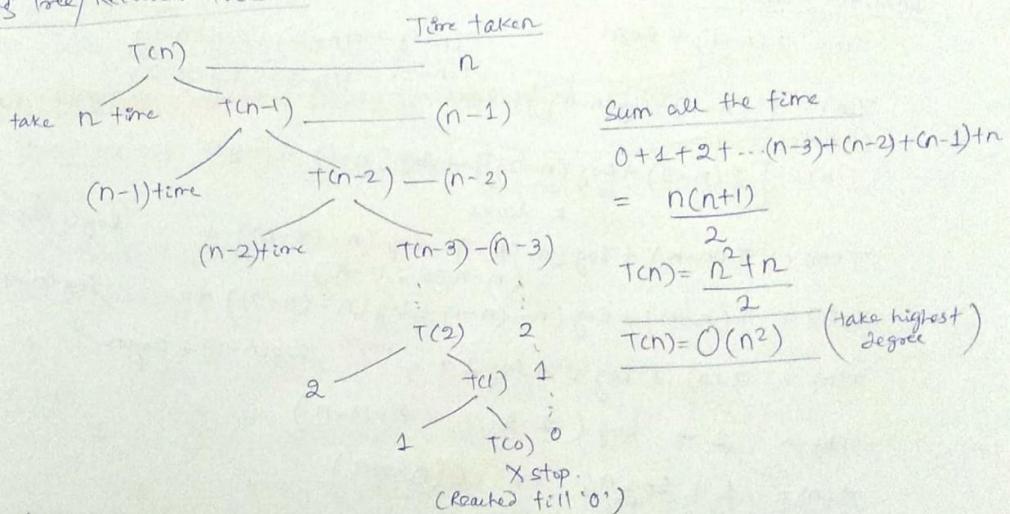
not easy to solve, convert to asymptotic notation.

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

Recurrence Relation

$$T(n) = \begin{cases} 1 & \text{Any constant} \\ n & n=0 (\text{do nothing}) \\ T(n-1) + n & n>0 \end{cases}$$

Tracing Tree / Recursive tree



Substitution

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

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

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

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

... K times ...

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

Assume  $n-K=0, n=K$

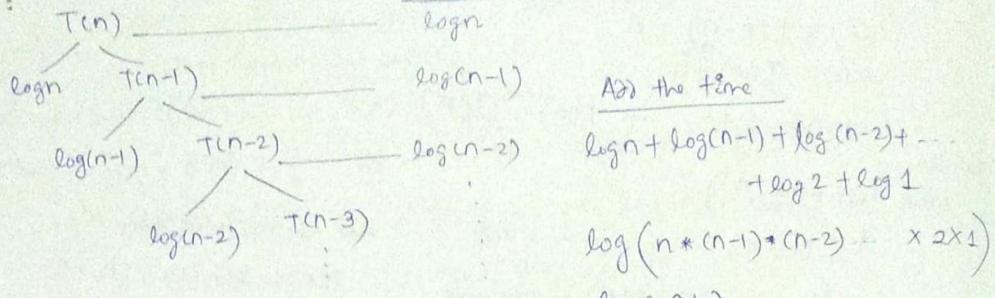
$$T(n) = T(n-n) + (n-n+1) + (n-n+2) + \dots + (n-1) + n$$

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

$$T(n) = 1 + n(n+1)/2 \quad T(n) = O(n^2)$$

Example : 3  $T(n) = \begin{cases} 1 & n=0 \\ T(n-1) + \log n & n>0 \end{cases}$

Tree :



$\log(n!)$

Stirling's formulae

$O(n \log n)$

or  $\log n!$  (no tight bound)  
Upper bound of  $\log n! = \log n^n$   
i.e.  $n \log n$  ✓

Induction / Substitution

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

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

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

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

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

... K times

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

$$(n-K=0, n=K)$$

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

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

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

$$T(n) = 1 + \log n! \text{ i.e. } O(n \log n)$$

Example : 4 \*

Algorithm Test (int n)

{ if ( $n > 0$ )

{ pointf("a/d", n); — print

Test(n-1); — T(n-1) print

Test(n-1); — T(n-1) print

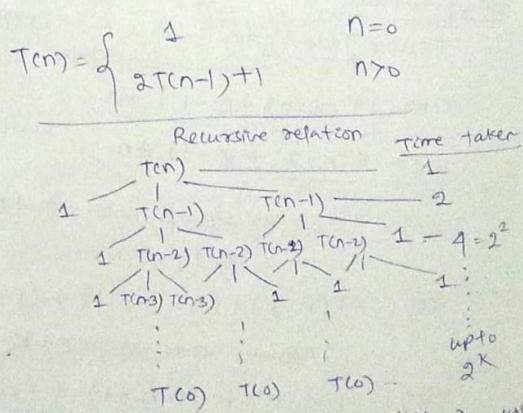
}

Add all the time :

$$1 + 2 + 2^2 + 2^3 + \dots + 2^K, \text{ GP series.}$$

$$a=1, r=2,$$

$$\text{i.e. } a + ar + ar^2 + ar^3 + \dots + ar^K = \frac{a(r^{K+1}-1)}{r-1} \text{ Put } a=1, r=2 \quad \frac{1(2^{K+1}-1)}{2-1} = 2^{K+1}-1$$



Assume  $n-K=0$ ,  $n=K$  i.e.  $2^{n+1}-1$  so  $O(2^n)$

Substitution:

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

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

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

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

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

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

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

... K times.

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

Assume  $n-K=0$ ,  $n=K$

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

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

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

$$T(n) = 2^{n+1} - 1 \text{ i.e. } O(2^n)$$

Recursive Relation (Dividing function) - Unit value of division = 1

Ex-5

void Test(int n)

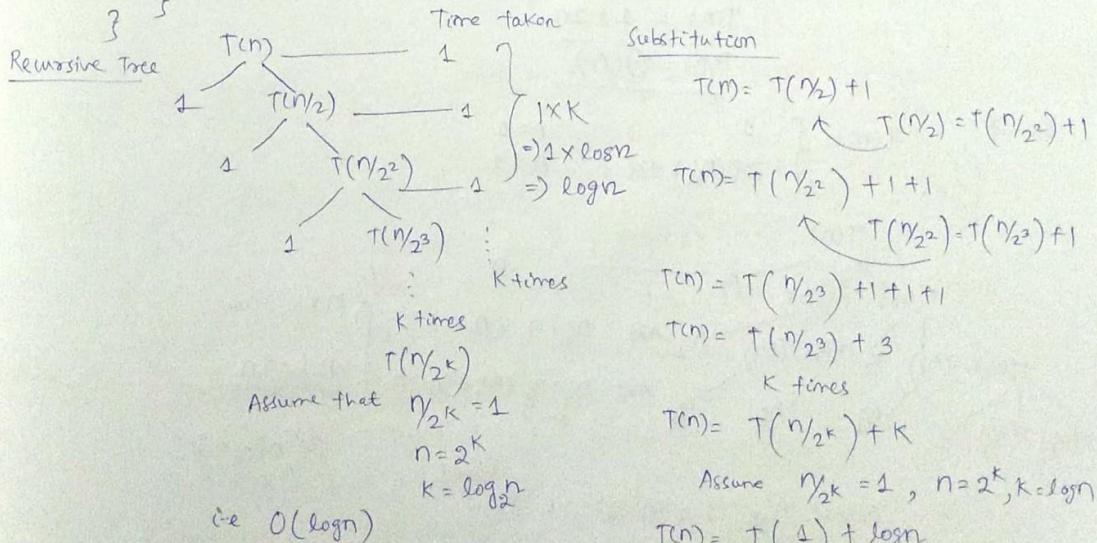
{ if( $n > 1$ )

{ printf("%d,%d",n);

Test( $n/2$ );

}

$$\rightarrow T(n) = \begin{cases} 1 & n=1 \text{ (nothing)} \\ T(n/2) + 1 & n > 1 \end{cases}$$



Ex - 6 :

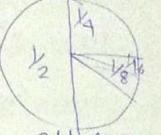
$$T(n) = \begin{cases} 1 & n=1 \\ T(n/2) + n & n>1 \end{cases}$$

Add :  $T(n) = n + n/2 + n/2^2 + \dots + \frac{n}{2^K} \text{ (approx)}$

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

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

$T(n) = 2n$   
i.e.  $T(n) = O(n)$



add all pieces get  $\frac{1}{2}$

Substitution

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

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

$$T(n) = [T(n/2^2) + n/2] + n$$

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

... K times

$$T(n) = T(n/2^K) + \frac{n}{2^{K-1}} + \frac{n}{2^{K-2}} + \dots + \frac{n}{2} + n$$

Assume that  $n/2^K = 1$ ,  $n = 2^K$ ,  $K = \log n$

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

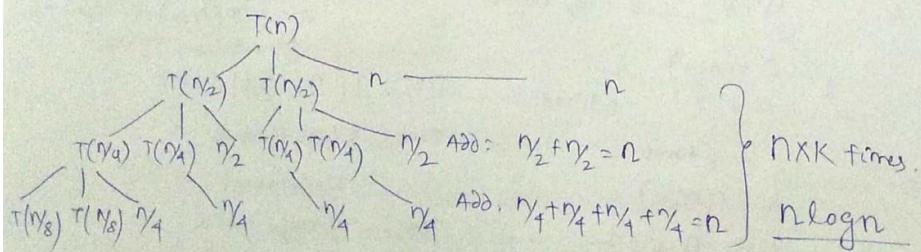
$$T(n) = 1 + n(1+1)$$

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

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

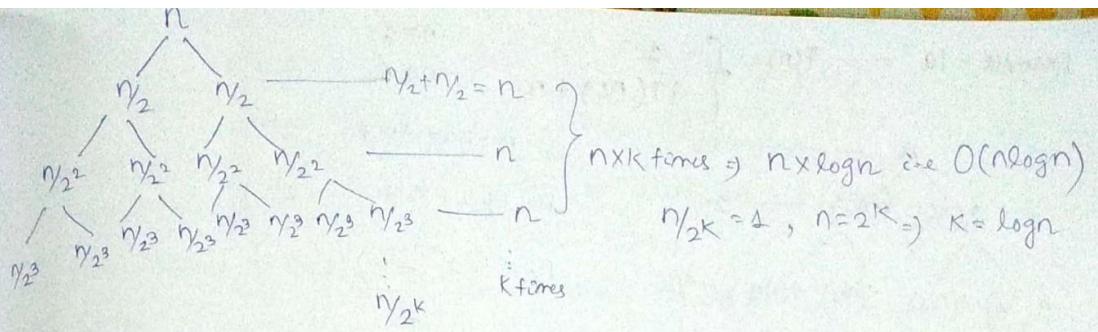
Ex - 7:

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



Assume  $n/2^K = 1$ ,  $n = 2^K$ ,  $K = \log n$   $O(n \log n)$

Note: Here we can consider the 2nd term of the recurrence as a cost. We do not consider the function. Avoid writing function.

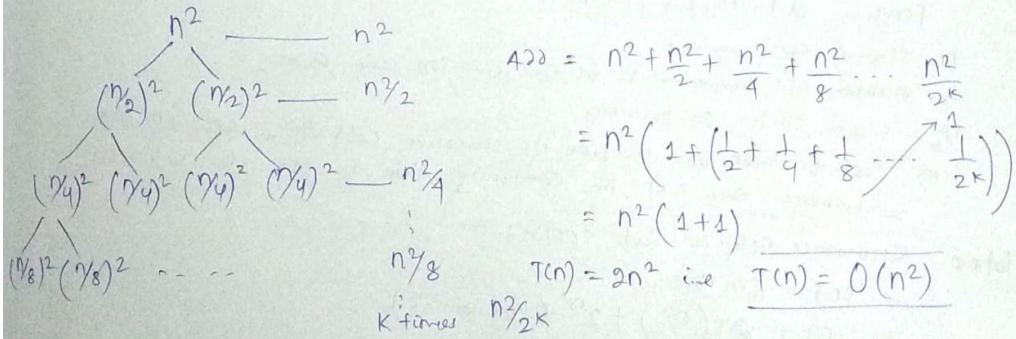


Substitute :

$$\begin{aligned}
 T(n) &= 2T(n/2) + n & T(n/2) &= 2T(n/2^2) + n/2 \\
 &= 2[T(n/2^2) + n/2] + n & T(n/2^2) &= 2T(n/2^3) + n/2^2 \\
 &= 2^2[T(n/2^3) + n/2^2] + n + n & & \\
 &= 2^3T(n/2^3) + n + n + n & & \\
 &= 2^3T(n/2^3) + 3n & \therefore K \text{ times} & \\
 &= 2^K T(n/2^K) + Kn & \text{Assume } n/2^K = 1, \text{ so } 2^K = n, K = \log n & \\
 T(n) &= n + (\overset{n}{\underset{1}{\sum}}) + n \log n & &
 \end{aligned}$$

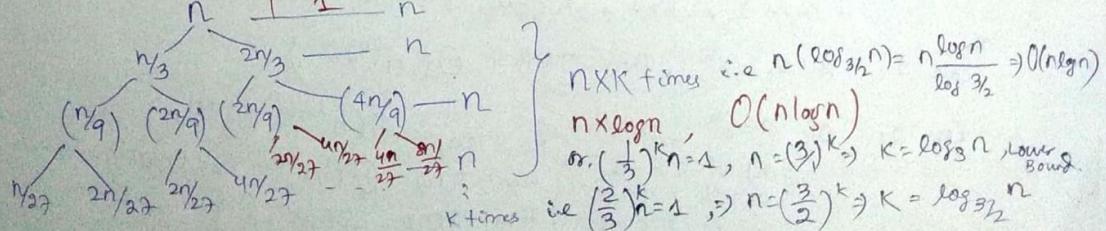
$$T(n) = n + n \log n \text{ ie } O(n \log n)$$

$$\underline{\text{Example - 8 : }} T(n) = \begin{cases} 2T(n/2) + n^2 & n > 1 \\ 1 & n = 1 \end{cases}$$



$$T(n) = 2n^2 \text{ ie } \underline{T(n) = O(n^2)}$$

$$\underline{\text{Example - 9 : }} T(n) = \begin{cases} T(2n/3) + T(2n/3) + n & n > 1 \\ 1 & n = 1 \end{cases}$$



Example - 10 -  $T(n) = \begin{cases} 1 & n=1 \\ 4T(n/2) + n & n>1 \end{cases}$

add  $n + 2n + 4n + \dots + 2^k n$   
 $n(1 + 2 + 4 + \dots + 2^k)$   
 $n \cdot 2 \frac{2^{k+1} - 1}{2 - 1}$   
 $\Rightarrow n \cdot 2^{\log_2 n + 1} - 1$   
 $\Rightarrow n(n \log_2 2^{\frac{n}{2}} - 2^{\frac{n}{2}} - 1)$   
 $\Rightarrow 2n^2 - n \text{ i.e. } O(n^2)$

#### Master Theorem for Decreasing function

$$T(n) = aT(n-b) + f(n), a > 0, b > 0 \text{ and } f(n) = O(n^k), \text{ where } k \geq 0$$

if  $a=1$  —  $O(n^{k+1})$  or  $O(n \cdot f(n))$

if  $a > 1$  —  $O(n^k a^{n/b})$  or  $O(n \cdot f(n))$

if  $a < 1$  —  $O(n^k)$  or  $O(f(n))$

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

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

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

$$T(n) = T(n-2) + 1 - n/2 \rightarrow O(n)$$

$$T(n) = T(n-100) + n \rightarrow O(n^2)$$

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

$$T(n) = 2T(n-2) + 1 \rightarrow O(2^{n/2})$$

$$T(n) = 2T(n-1) + n \rightarrow O(n2^n)$$

#### Master Theorem for Dividing function

#### Extension Master Methods / Advanced Master theorem

It is used to determine running time of algorithms (divide and conquer) in terms of asymptotic notations.

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

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

$n$  = size of the problem

$a$  = number of subproblems in the recursion and  $a > 1$

$n/b$  = size of each subproblem

$f(n)$  = cost of work done outside the recursive call like dividing into subproblems and cost of combining them to get the solution.

Note: Recurrence Relation not applied —

$$\bullet T(n) = \sin n$$

$$\bullet T(n) = 2T(n/2) + 2^n \text{ (Polynomial)}$$

$$T(n) = aT(n/b) + O(n^k \log^p n) \quad [\text{Advanced}]$$

$a > 1, b > 1, k \geq 0, p = \text{real number.}$

Then  
case 1. if  $a > b^k$ , then  $T(n) = \Theta(n^{\log_b a})$

case 2. if  $a = b^k$ , then  $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

a) if  $p > -1$ , then  $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

b) if  $p = -1$ , then  $T(n) = \Theta(n^{\log_b a} \log \log n)$

c) if  $p < -1$ , then  $T(n) = \Theta(n^{\log_b a})$

case 3. if  $a < b^k$ , then

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

b) if  $p < 0$ , then  $T(n) = \Theta(n^k)$

Examples: ①  $T(n) = 2^n T(n/2) + n^n$  (can't be solved, not in the form)

②  $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$  (case 3)

③  $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 \log n)$  (case 2)

④  $T(n) = T(n/2) + 2^n \Rightarrow \Theta(2^n)$  (case 3)

⑤  $T(n) = 16T(n/4) + n = \Theta(n^2)$  (case 1)

⑥  $T(n) = 2T(n/2) + n \log n \Rightarrow \Theta(n \log^2 n)$  (case 2)

⑦  $T(n) = 2T(n/2) + \frac{n}{\log n} \Rightarrow$  Does not apply, but by substitution/Recursion we get the answer, i.e.  $\Theta(n \log \log n)$ ,

⑧  $T(n) = 2T(n/4) + n^{0.51} \Rightarrow \Theta(n^{0.51})$  (case 3)

⑨  $T(n) = 0.5T(n/2) + 1/n \Rightarrow$  Does not apply ( $a < 1$ )

⑩  $T(n) = 16T(n/4) + n! \Rightarrow \Theta(n!)$  (case 3)

⑪  $T(n) = \sqrt{2}T(n/2) + \log n \Rightarrow \Theta(\sqrt{n})$  (case 1)

⑫  $T(n) = 3T(n/2) + n \Rightarrow \Theta(n^{2/3})$  (case 1)

⑬  $T(n) = 3T(n/3) + \sqrt{n} \Rightarrow \Theta(n)$  (case 1)

⑭  $T(n) = 4T(n/2) + cn \Rightarrow \Theta(n^2)$  (case 1)

⑮  $T(n) = 3T(n/4) + n \log n \Rightarrow \Theta(n \log n)$  (case 3)

⑯  $T(n) = 3T(n/3) + n^2 \Rightarrow \Theta(n \log n)$  (case 2)

⑰  $T(n) = 6T(n/3) + n^2 \log n \Rightarrow \Theta(n^2 \log n)$  (case 3)

⑱  $T(n) = 4T(n/2) + n/\log n \Rightarrow \Theta(n^2)$  (case 1)

⑲  $T(n) = 64T(n/8) - n^2 \log n \Rightarrow$  Does not apply,  $n$  not positive

⑳  $T(n) = 7T(n/3) + n^2 \Rightarrow T(n) = \Theta(n^2)$  (case 3)

㉑  $T(n) = 4T(n/2) + \log n \Rightarrow \Theta(n^2)$  (case 1)

㉒  $T(n) = T(n/2) + n(2 - \cos n) \Rightarrow$  Does not apply.

### Algorithm Design Techniques :

1. Divide-and-Conquer Method
2. Greedy Method
3. Dynamic programming method
4. Branch and Bound method
5. Approximation method
6. Backtracking method

f. Recursive method

g. Randomized method

h. Brute-force method

i. Incremental method.



## Analysis of Algorithm | Set 5 (Amortized Analysis Introduction)

**Amortized Analysis** is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.

The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes fast, but space required becomes high.

The solution to this trade-off problem is to use **Dynamic Table (or Arrays)**. The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full.

- 1) Allocate memory for a larger table of size, typically twice the old table.
- 2) Copy the contents of old table to new table.
- 3) Free the old table.

Initially table is empty and size is 0

|                             |   |   |   |   |   |   |   |  |
|-----------------------------|---|---|---|---|---|---|---|--|
| Insert Item 1<br>(Overflow) | 1 |   |   |   |   |   |   |  |
| Insert Item 2<br>(Overflow) | 1 | 2 |   |   |   |   |   |  |
| Insert Item 3               | 1 | 2 | 3 |   |   |   |   |  |
| Insert Item 4<br>(Overflow) | 1 | 2 | 3 | 4 |   |   |   |  |
| Insert Item 5               | 1 | 2 | 3 | 4 | 5 |   |   |  |
| Insert Item 6               | 1 | 2 | 3 | 4 | 5 | 6 |   |  |
| Insert Item 7               | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  |

If the table has space available, we simply insert new item in available space.  
Next overflow would happen when we insert 9, table size would become 16

### What is the time complexity of n insertions using the above scheme?

If we use simple analysis, the worst case cost of an insertion is  $O(n)$ . Therefore, worst case cost of n inserts is  $n * O(n)$  which is  $O(n^2)$ . This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take  $\Theta(n)$  time.

|            |   |   |   |   |   |   |   |    |    |    |       |
|------------|---|---|---|---|---|---|---|----|----|----|-------|
| Item No.   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8  | 9  | 10 | ..... |
| Table Size | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 16 | 16 | 16 | ..... |
| Cost       | 1 | 2 | 3 | 1 | 5 | 1 | 1 | 1  | 9  | 1  | ..... |

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1 \dots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\begin{aligned}\text{Amortized Cost} &= \frac{[(\underbrace{1+1+1+1\dots}_{n \text{ terms}}) + (\underbrace{1+2+4+\dots}_{\lceil \log_2(n-1) \rceil + 1 \text{ terms}})]}{n} \\ &\leq \frac{[n+2n]}{n} \\ &\leq 3\end{aligned}$$

$$\text{Amortized Cost} = O(1)$$

So using Amortized Analysis, we could prove that the Dynamic Table scheme has O(1) insertion time which is a great result used in hashing. Also, the concept of dynamic table is used in [vectors in C++](#), [ArrayList in Java](#).

Following are few important notes.

**1)** Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

**2)** The above Amortized Analysis done for Dynamic Array example is called [Aggregate Method](#).

**Method** . There are two more powerful ways to do Amortized analysis called [Accounting](#)

**Method** and [Potential Method](#) . We will be discussing the other two methods in separate posts.

**3)** The amortized analysis doesn't involve probability. There is also another different notion of average case running time where algorithms use randomization to make them faster and expected running time is faster than the worst case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing. We will soon be covering Randomized analysis in a different post.

#### Sources:

[Berkeley Lecture 35: Amortized Analysis](#)

[MIT Lecture 13: Amortized Algorithms, Table Doubling, Potential Method](#)

<http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec20-amortized/amortized.htm>

## Divide & Conquer Algorithm (Recursive strategy)

### UNIT: 2

We solve a problem recursively by applying 3 steps.

1. Divide : Break the problem into several sub problems of smaller size.
2. Conquer : Solve the problem recursively.
3. Combine : Confine these solutions to create a solution to the original problem.

Control Abstraction of Divide & Conquer algorithm.

Algorithm D and C (P)

{ of small(p)  
then return S(P)

else

{ divide P into smaller instances  $P_1, P_2, P_3 \dots P_k$   
Apply D and C to each sub problem.

Return combine (D and C( $P_1$ ) + D and C( $P_2$ ) + ... + D and C( $P_k$ ))

}

}

D & C problems : Binary Search, finding maximum & minimum, Merge sort,  
Quick sort, Strassen's matrix multiplication.

Binary Search : It is very efficient searching technique which works for sorted list.

Binary-Search(A, LB, UB, SITEM)

if ( $LB > UB$ ) then

return -1

Set  $MID = \lfloor (LB+UB)/2 \rfloor$  } floor

if ( $SITEM = A[MID]$ ) then

return (MID)

else if ( $SITEM < A[MID]$ ) then

return (Binary-Search(A, LB, MID-1, SITEM))

else

return (Binary-Search(A, MID+1, UB, SITEM))

end of if

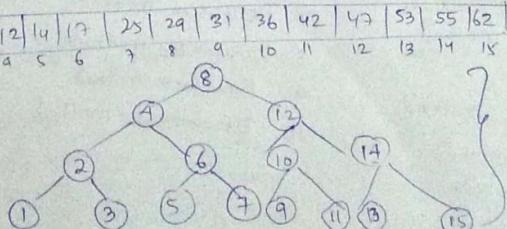
end of if

3 6 8 12 14 17 25 29 31 36 42 47 53 55 62  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Min - 1 = O(1)

Max - Logn - O(logn)

Avg - O(logn)



Maximum comparison depends on height of tree

Logn

Approx.  $15+1=16$  elements.  
 $\log_2 16 = 4$  comparisons

Algorithm RBinSearch(l, h, key)

```

    if (l == h)
        if (A[l] == key)      } — 1 unit
            return l;          } small problem
        else
            return 0; // Key element not found.
    }
    else
        mid = (l+h)/2;      } — 1 unit
        if (key == A[mid])  } — 1 unit
            return mid;      } large Problem
        else if (key < A[mid])
            return RBinSearch(l, mid-1, key); } T(n/2) times
        else
            return RBinSearch(mid+1, h, key);
    }
}

```

(Divide & Conquer approach) ✓ Best Case Time complexity =  $O(1)$   
   ✓ Worst Case Time complexity =  $O(\log n)$

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

Already done  
(Substitution &  
method)

Recursive Tree

Master Theorem:  $a=1, b=2, k=0, b^k = 2^0 = 1, a=b^k, p=0$

$$O(n^{\log_2 1 \log^{0+1} n}) \Rightarrow O(n^0 \log n) = \Theta(\log n)$$

Binary search takes constant ( $O(1)$ ) space, meaning that the space taken by the algorithm is the same for any number of elements in the array.

$O(1)$  in case of iterative implementation.  
 $O(\log n)$ , in case of recursive implementation - recursion call stack space.

Quick Sort : To sort ~~subarray~~ A[P...r]:  $p = \text{low}, A[q] = \text{pivot element}, r = \text{high}$

Based on 3-step Divide & Conquer

Divide: Partition A[P...r], into two (possibly empty) subarrays A[P...q-1] and A[q+1...r], such that each element in the <sup>first</sup> subarray A[P...q-1] is  $\leq A[q]$  and  $A[q] \leq A[q+1 \dots r]$  (second subarray).

Conquer: Sort the two subarrays by recursive calls to QUICKSORT.

Combine: No work is needed to combine the subarrays, because they are sorted in place.

- Perform the divide step by a procedure PARTITION, which returns the index  $q$ , that marks the position separating the subarrays.
- we can take pivot element at last or at first or in the middle.

We recursively perform 3 steps:

1. Bring the pivot to its appropriate position such that left of the pivot is smaller and right is greater.

2. Quick sort the left part.

3. Quick sort the right part.

### Quick Sort Algorithm

(Lomuto partition scheme)

PARTITION(A, P, r)

$$x = A[\sigma]$$

$$i = p-1$$

for ( $j=p$  to  $\sigma-1$ )

{ if ( $A[j] \leq x$ )

$$\{ i = i+1;$$

exchange  $A[i]$  with  $A[j]$

}

exchange  $A[i+1]$  with  $A[\sigma]$

return  $i+1$

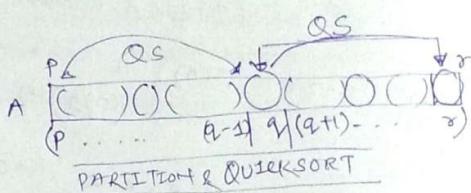
QUICKSORT(A, P, r)

{ if ( $P < r$ )

{  $q = \text{PARTITION}(A, P, r)$

$\text{QUICKSORT}(A, P, q-1)$

$\text{QUICKSORT}(A, q+1, r)$



### Performance of Quicksort :

It depends on the partitioning of the subarrays.

If the subarrays are balanced, then QUICKSORT can run as fast as mergesort.

If the subarrays are unbalanced, then it runs slowly.

If they are unbalanced, then it performs better as compared to mergesort.

Example:

| loop | partition                           | queue |
|------|-------------------------------------|-------|
|      | $10   80   30   90   140   50   70$ |       |

$i = -1$  (initially)

$j = 0$  (scan the element upto  $high-1$ )

Pass 1 :  $10 < 70$  (True) Action:  $i+1$ , swap( $arr[i], arr[j]$ )  $i=0, j=0$  (Both are same)

Pass 2 :  $80 < 70$  (False) No action  $i=0, j=1$

Pass 3 :  $30 < 70$  (True) Action,  $i+1$  & swap  $i=1, j=2$

|                                     |
|-------------------------------------|
| $10   30   80   90   140   50   70$ |
|-------------------------------------|

Pass 4 :  $90 < 70$  (False) No action  $i=1, j=3$

part 5:  $40 < 70$  (True)      if & swap      i=2, j=4  
[10] 30 [40] 100 [50] 150 [70]

pass 6:  $50 < 70$  (True)      if & Swap      i=3, j=5

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 10 | 30 | 40 | 50 | 80 | 90 | 70 |
|----|----|----|----|----|----|----|

( Before Pass 7, j becomes 6, so we come out of the loop.)  $i=3, j=6$

We know exchange  $A[i+1]$  with  $A[s]/\text{pivot}$ . ,  $i=3$   
 $A[i+1] = A[4]$

Right  
10 | 30 | 40 | 50 | 70 | 90 | 80 |

Now 70 is brought to its appropriate position by partition function.  
 begin Quick sorting on the left part.

We can begin Quick sorting on the left part

pivot  
so in apply partition function,  $50 = \text{Pivot}$ , already incorrect position.  
before pivot are

again apply partition function,  $50 = \text{pivot}$ , again Quicksort,  $40 = \text{pivot}$ , similarly  $30, 10$  become pivot and already in correct position.

So left array is sorted.

Right past

pivot to correct position.

80 & 90 swapped to bring

80 | 90

Now, Sorted Array is :-

### Time complexity Analysis of Quick Sort Best Case

## Worst Case

$$q = \text{Partition}(A, P, \alpha) \quad T(n/2)$$

$\text{Tr}^{(0)}$ ) [one element on one side]

QUICKSORT(A, P, Q-1)  $\Theta(n^2)$

$T(n-1)$  [n-1 element in  
other side]

$$\text{QUICKSORT}(A, q+1, r) \rightarrow T(n) = 2T\left(\frac{n}{2}\right) + n$$

Best Case: of the partitioning algorithm,  
there will be a equal size partition.  
↳ sorted in ascending order or descending order or

Worst case : If the array is already sorted.

Worst case : If the array is an having all equal elements.  $\therefore$  time is  $\Theta(n^2)$  unit of times approx. because j is scan

Partition requires  $(n-1)$  unit of time to form 0 to  $n-1$ .

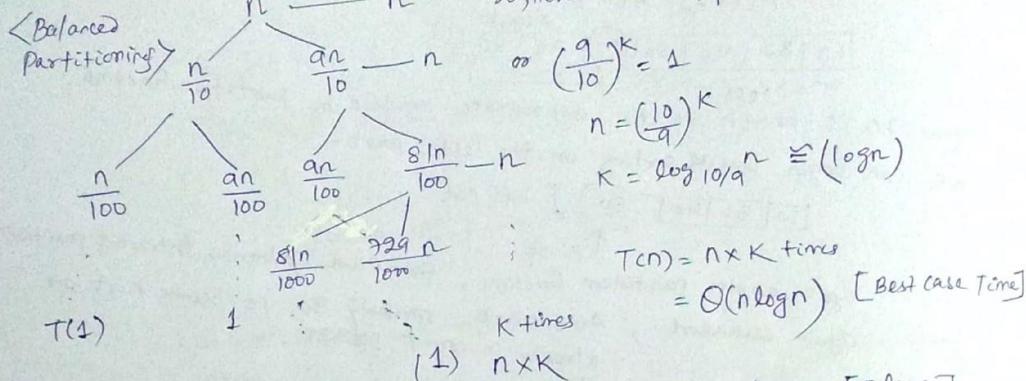
the each array element from 0 to  $n-1$ .  
only quick sort & find TCM?

Ex 2:  apply Quick sort & find TCM?  
 In this case 6 taken as pivot, but pivot element is already placed in exact/proper position. Then we take 5, 4, 3, 2, 1 as pivot & all placed in proper position.

Ex-2: Descending order  
 $(6 \ 5 \ 4 \ 3 \ 2 \ 1)$  so  $(1 \ 5 \ 4 \ 3 \ 2 \ 6)$  j & i same  
 No less element compared to 1. then swap with pivot

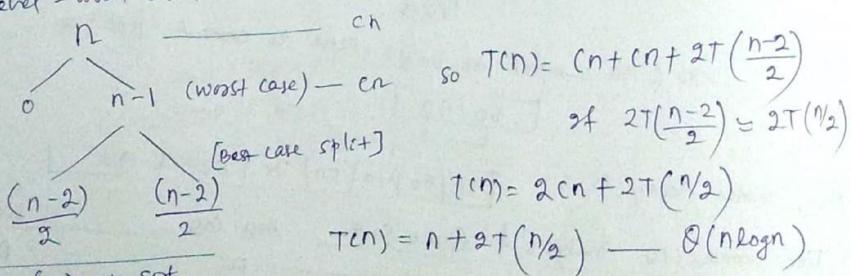
Ex-3: Array contains same element  
 $(1 \ 1 \ 1 \ 1 \ 1 \ 1)$  j & i same  $T(n) = O(n) + T(n-1) = O(n^2)$

Ex-4: partitioning algorithm, split the input in 1:9 ratio.  
 so, here  $n \rightarrow n/10 - n/(10/9)^2 - \dots - 1$



if the split is 1:9, 1:99, 1:999 ... - Time complexity [ $n \log n$ ]

Ex-5: 1-level - worst case, 2nd level best case ..., 3rd - worst, 4th - best case, find  $T(n)$ ?



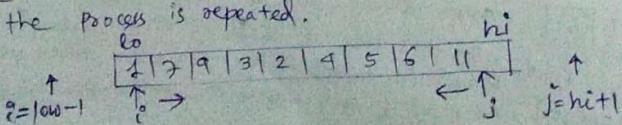
#### Different versions of Quicksort

There are many different versions of Quicksort that pick pivot in different ways.

1. Always pick first element as pivot
2. Always pick last element as pivot
3. Pick a random element as pivot
4. Pick median as pivot.

#### Hoare's partition scheme

Hoare's partition scheme works by initializing two indexes that start at two ends, the two indexes move toward each other until an inversion is (A smaller value on left side and greater value on right side) found. When an inversion is found, two values are swapped and the process is repeated.



#### Space Complexity:

For Best Case: it depends on the number of levels or height of tree i.e.  $O(\log n)$ , split equal halves

For Worst Case: not split equal halves, here height of tree might goes  $O(n)$  i.e.  $O(n)$

### Hoare's partition Algorithm (arr[], lo, hi)

#### partition (A, lo, hi)

Pivot = A[lo]

i = lo - 1 // initialize left index

j = hi + 1 // initialize right index

do

i = i + 1

while A[i] < pivot

do

j = j - 1

while A[j] > pivot

if i = j then

return j

swap A[i] with A[j]

Note: Hoare's scheme is more efficient than Lomuto's partition scheme because it does fewer swaps. If input array is already sorted & Time complexity  $O(n^2)$ .  
Lomuto's partition scheme is easy to implement as compared to Hoare's scheme.

### Randomized Quicksort

In Randomized Quicksort we use random number to pick the next pivot (or we randomly shuffle the array). We use randomization to improve the performance of quicksort against those worst-case instances. It randomly picks one element in the sequence, swaps it with the first element, and then calls partition.

#### Randomized - Partition (A, P, r)

1. i = Random (P, r)

2. exchange A[r] with A[i]

3. return partition (A, P, r)

If the pivot is selected uniformly at random, the expected number of comparisons of randomized version of Quicksort is bounded by  $O(n \log n)$ .

### Merge Sort :

It is one of the well-known divide-and-conquer algorithms.

Divide: Split array down the middle into two sub-sequences, each of size approx.  $\frac{n}{2}$ .

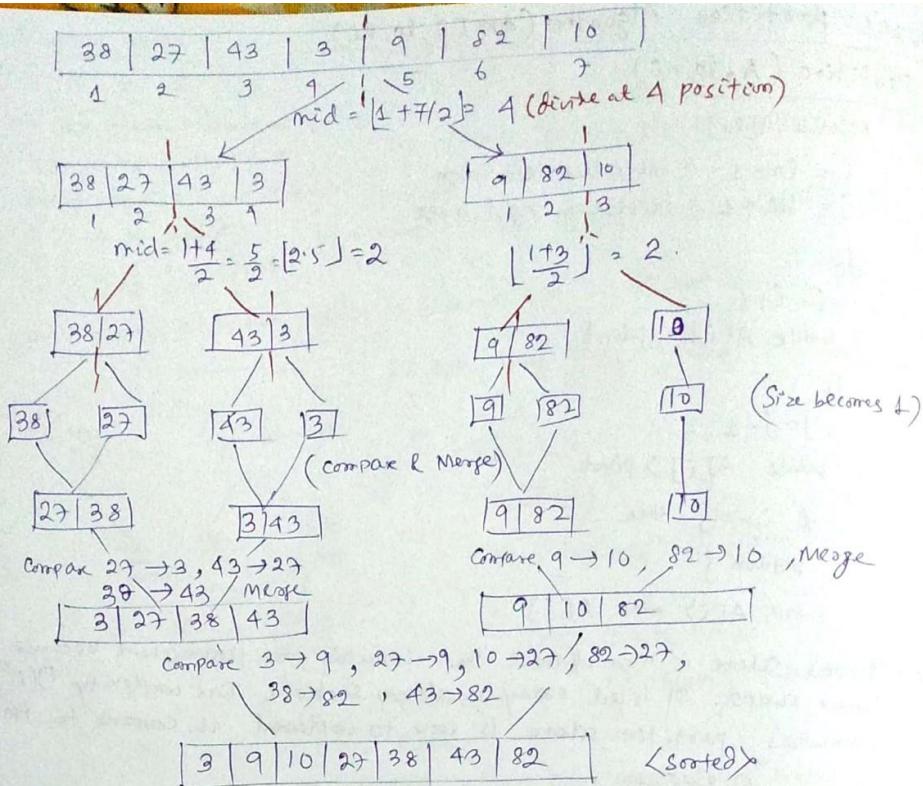
Conquer: Sort each subsequence (by calling mergesort recursively on each).

Combine: Merge the two sorted sub-sequences into a single sorted list.

So the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge process comes into action and starts merging arrays back till the complete array is merged.

Heart of Merge Sort "Merging"

Example:



### MERGE ( $A, P, q, r$ )

```

 $\{ n_1 = q - p + 1 \text{ // Count the no. of elements present in Left Sub-Array}$ 
 $n_2 = r - q \text{ // Count the no. of elements present in Right Sub-Array}$ 
let  $L[1 \dots n_1]$  and  $R[1 \dots n_2]$  be new arrays
for ( $i = 1 \dots n_1$ ) // copy the array element (left part)
     $L[i] = A[p+i-1]$ 
for ( $j = 1 \dots n_2$ ) // copy the array element (right part)
     $R[j] = A[q+j]$ 
 $L[n_1+1] = \infty \text{ // in program take big number i.e } L[n_1+1] = 9999$ 
 $R[n_2+1] = \infty \text{ // } R[n_2+1] = 9999$ 
 $i = 1, j = 1$ 

```

### Time complexity of Merge Algorithm

```

if ( $L[i] \leq R[j]$ ) // if left element
     $A[K] = L[i]$  // is less, copy the
     $i = i+1$  // left element into array
    & increase  $i+1$ 
else
     $A[K] = R[j]$  // if Right element is
     $j = j+1$  // less, copy the right
    element into array &
    increase  $j+1$ 

```

$n$  unit time required for  
 copying array element to  
 new array.  
 $n$  unit of time required for  
 comparison.  
 $n+n = 2n = O(n)$

Commented [Ma1]:

Why  $\infty$  is taken in both the left & Right Sub-Array?

|  |                            |
|--|----------------------------|
| $10   20   30   \infty$  | $5   6   9   \infty$       |
| Compare & merge, $10 \rightarrow 5, 6 \rightarrow 10$ , then $9 \rightarrow 10, \infty \rightarrow 10$ |                            |
| $20 \rightarrow \infty, 30 \rightarrow \infty$   | $5   6   9   10   20   30$ |

In program, we take the ' $\infty$ ' value as big number.

MERGE-SORT( $A, P, R$ )  $\rightarrow T(n)$  in Merging Procedure Extra Array is required, so it is called as 'out of place'

if  $P < R$

$$q = \lfloor (P+R)/2 \rfloor \text{ Floor value}$$

$$\text{MERGE-SORT}(A, P, q) - T(n/2)$$

$$\text{MERGE-SORT}(A, q+1, R) - T(n/2)$$

$$\text{MERGE}(A, P, q, R) - O(n)$$

Time

Space complexity

$$O(n) \rightarrow \text{extra space for extra array}$$

Recursive stack space

$$O(\log n) \rightarrow$$

Total space  $\Rightarrow O(n) + O(\log n)$

$$\Rightarrow O(n)$$

from merge

Merge Sort is efficient if number of array element is more as compared to Quick sort.

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

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

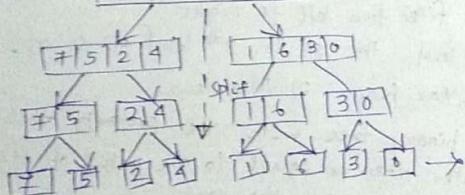
apply Master, Recursive or Substitution  $\rightarrow O(n \log n)$  - Time complexity

if the array is sorted, reverse sorted list the time taken is same.

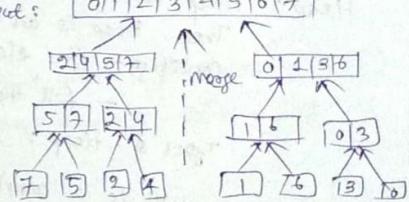
so Best Case, Average Case and Worst Case is same i.e.  $O(n \log n)$

High level view of algorithm Merge sort

input:  $7 | 5 | 2 | 4 | 1 | 1 | 6 | 3 | 0$



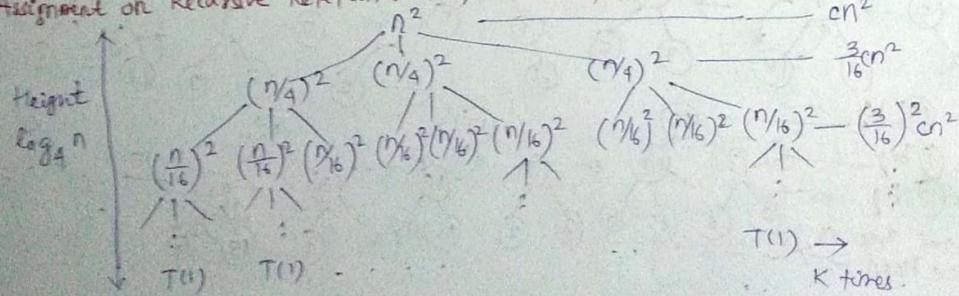
output:  $0 | 1 | 2 | 3 | 4 | 5 | 6 | 7$



Here the "divide" phase is shown on the left side. It works top-down splitting up the list into smaller sublists.

The "conquer and combine" phases are shown on right. They work bottom-up merging sorted lists together into larger sorted list.

Assignment on Recursive Relation:  $T(n) = 2T(n/4) + cn^2$



$$\text{Assume: } \frac{n}{q}k = 1, k = \log_4 n$$

Determine the cost at each level of the tree. Each levels has 3 times more nodes than the level above.

So no. of nodes at depth  $K$  is  $3^K$ . ( $k = \log_4 n$ )

$$3^K = 3^{\log_4 n} = n^{\log_4 3}$$

$n^{\log_4 3}$  no. of nodes, each contributing cost  $T(1)$ , for a total cost of  $n^{\log_4 3} T(1)$ , i.e.  $\Theta(n^{\log_4 3})$

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$= \sum_{K=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^K cn^2 + \Theta(n^{\log_4 3})$$

$$< \sum_{K=0}^{\infty} \left(\frac{3}{16}\right)^K cn^2 + \Theta(n^{\log_4 3})$$

[for infinite decreasing geometric progression:  $= \frac{1}{1-\frac{3}{16}}$ ]

$$= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

$$\underline{T(n) = \Theta(n^2)}$$

## Heap Sort

Heap: Heap data structure is a complete binary tree, Trinary Tree or  $n$ -ary tree. Heap is an almost complete binary tree too.

Condition - the element filled from left to right

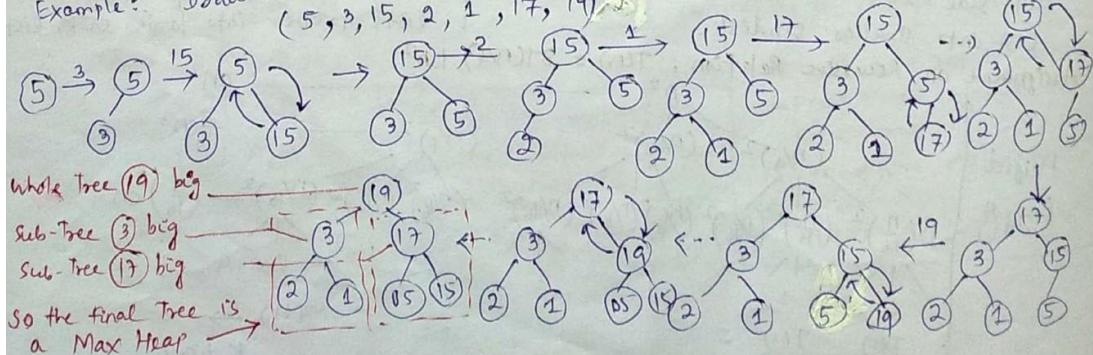
- fill the 2<sup>nd</sup> level then go to 3<sup>rd</sup> level and so on.

Types of heap: 1) Max heap 2) Min heap.

Max Heap: It is a complete binary tree with the property that the value of each node is greater than or equal to the value of its children.

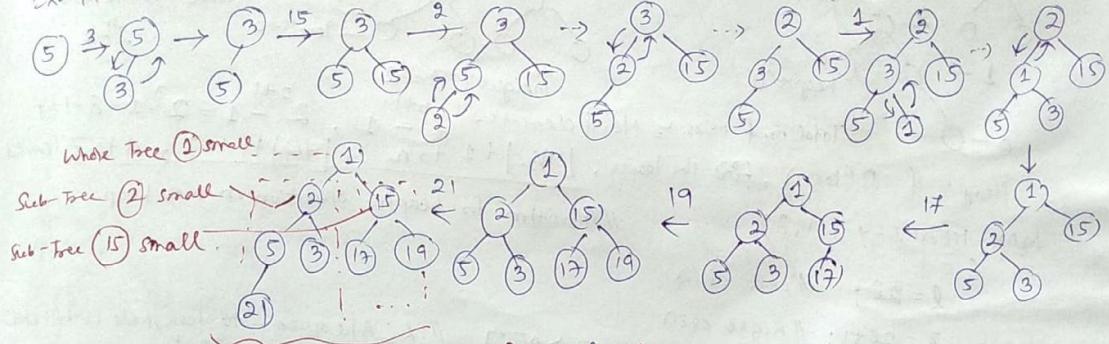
The largest element is at the root.

Example: Draw a max heap using the following sequence of nos.



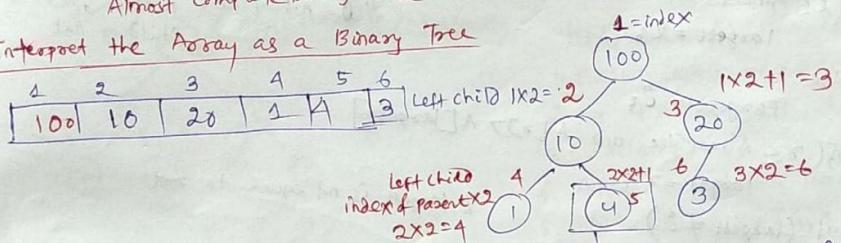
**Min Heap :** The min heap is a complete binary tree with a property that the value at each node is smaller than or equal to the values of its children. Smallest element is at root

Example: Draw a mean heap for the following sequence of no.  $\langle 5, 3, 15, 2, 1, 17, 19, 21 \rangle$



Almost complete binary tree of a Min Heap.

Interpret the Array as a Binary Tree



If we are present in the node (4) having index = 5, find parent?

$$P = \left\lfloor \frac{\text{index}}{2} \right\rfloor_{\text{Floor}} = \left\lfloor \frac{5}{2} \right\rfloor = \left\lfloor 2.5 \right\rfloor \leq 2 \text{ (Parent is at 2 index)}$$

for node (3), index = 6, parent =  $\left\lfloor \frac{6}{2} \right\rfloor = 3$  (parent is at 3 index)

$$so \quad p(i) = \lfloor i/2 \rfloor$$

$$\text{Left}(i) = \text{Left Child} = 2 \times i, \text{ for node } (1), \text{ Parent index } = 2, 2 \times 1 = 2 \\ \text{Left}(i) = \text{Left Child} = 2 \times i + 1, \text{ for node } (4), \text{ Parent index } = 3, 2 \times 2 + 1 = 5$$

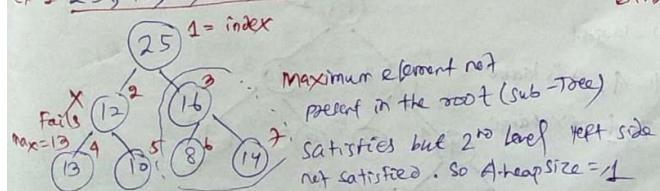
$\text{right}(i) = \text{Right child} = 2i + 1$ , too node (4),  
 $\text{right}(i) = 180 = 4$

$\text{Left}(i) = 2 = \text{Binary } 10, \text{ Left shift, } \frac{(100)}{\text{Binary}} = 4$

$$P(i) = \frac{1}{2}, \quad \text{child} = 5 = 101, \quad P(j) = \lfloor \frac{i}{2} \rfloor = \text{Rightmost bit } 1 = 100 + 1 = 4 + 1 = 5$$

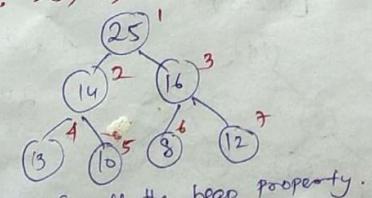
Right (i) = 2 = Binary 10, Right Shift,  $\frac{100}{\text{Binary}}$

Ex-1 25, 12, 16, 13, 10, 8, 14, represent in Binary Tree



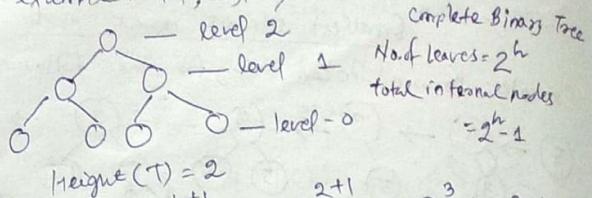
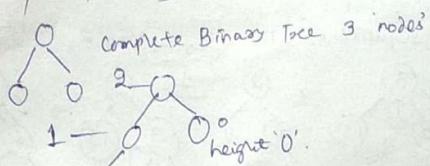
A.length = 7  
How many elements follow?  
A.heapsize = 1

; 25, 14, 16, 13, 10, 8, 12



Array in ascending order  $\rightarrow$  Min Heap, Examples: 8, 10, 12, 13, 14

Array in decreasing order  $\rightarrow$  Max Heap example: 14, 13, 12, 10, 8



$$\text{Total no. of nodes} \leq \text{No. of elements} = 2^{h+1} - 1, 2^{h+1} - 1 = 2^3 - 1 = 8 - 1 = 7$$

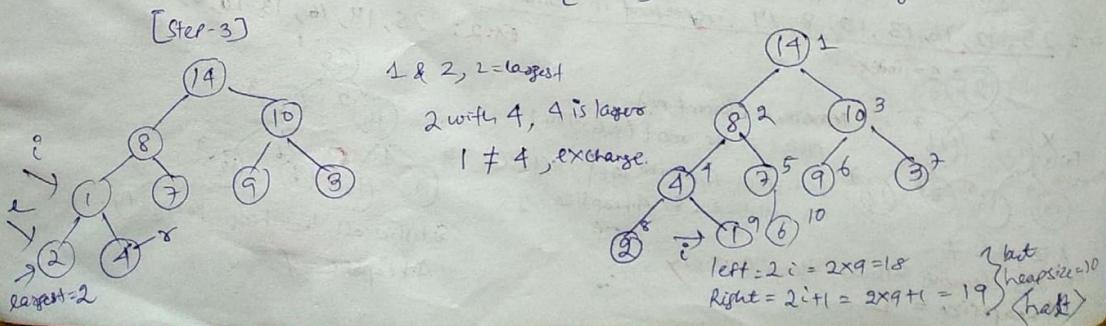
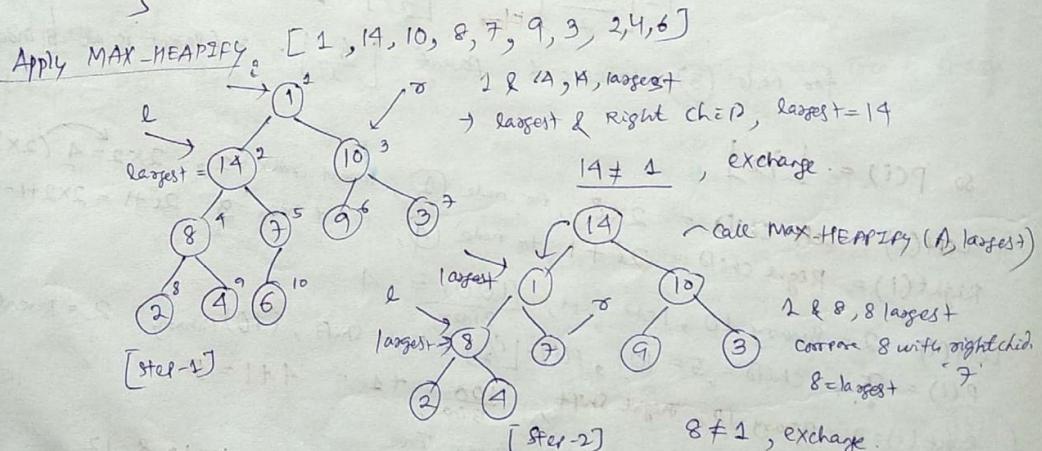
✓ Height of  $O(\log n)$ , find the leaves,  $\lfloor \frac{n}{2} \rfloor + 1$  to  $n$  i.e.  $\lfloor \frac{7}{2} \rfloor + 1$  to 7, 4 to 7 leaves

MAX-HEAPIFY ( $A, i$ ) // Algorithm for heapify operation on max heap.

```

    {
        l = 2i; // Left child
        r = 2i+1; // Right child
        if (l < A.heapSize and A[l] > A[i]) // l < A.heapSize - to check node is left child or not.
            largest = l;
        else
            largest = i;
        if (r < A.heapSize and A[r] > A[largest]) // r < A.heapSize - to check node is right child or not
            largest = r;
        if (largest != i)
            exchange A[i] with A[largest]
        MAX-HEAPIFY (A, largest)
    }

```



BUILD-MAX-HEAP(A) // Algorithm to build a max heap (A)

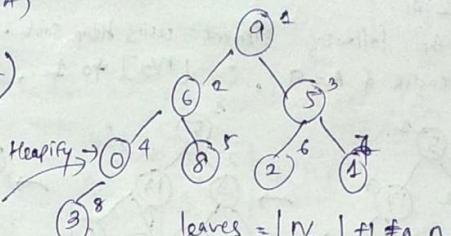
1. A.heap-size = A.length

for ( $i = \lfloor A.length/2 \rfloor$  down to 1)

MAX-HEAPIFY(A, i)

3

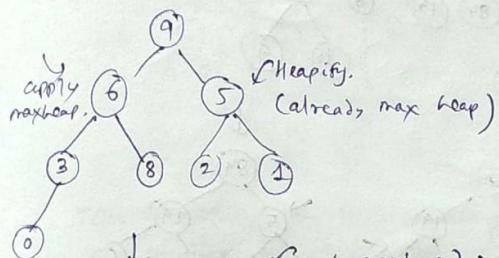
apply Heapiify from 4 to 1 node.



leaves =  $\lfloor n/2 \rfloor$  + 1 to  $n$

$\lfloor \frac{n}{2} \rfloor + 1$  to 8, 5 to 8.

Non leaves = 1 to  $\lfloor \frac{n}{2} \rfloor$ , 1 to  $\lfloor \frac{8}{2} \rfloor$ , 1 to 4



apply Heapiify (already max heap)

Algorithm for Heap Sort using max-heap.

1. BUILD-MAX-HEAP(A)

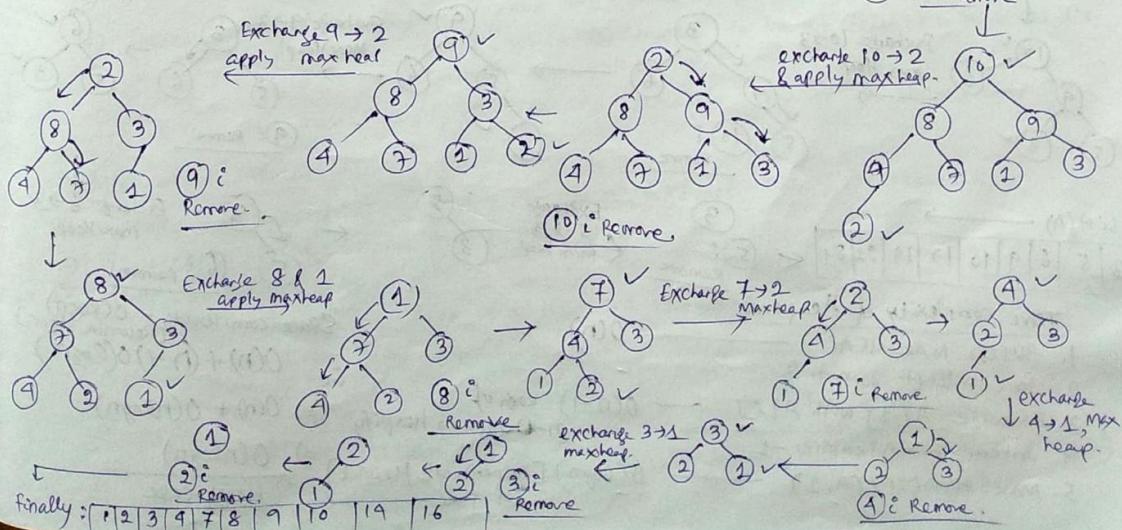
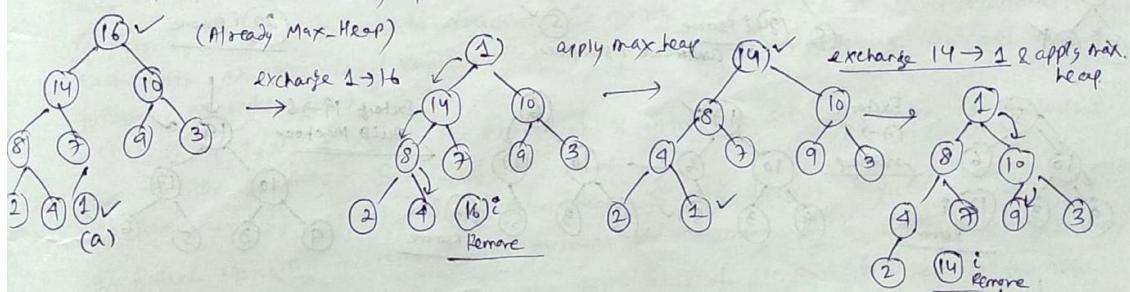
2. for  $i = A.length$  down to 2

3. exchange  $A[1]$  with  $A[i]$

4. A.heap-size = A.heap-size - 1

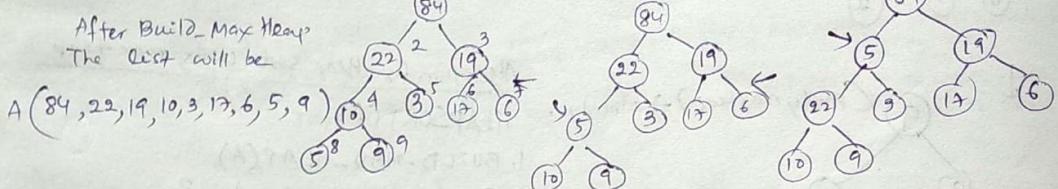
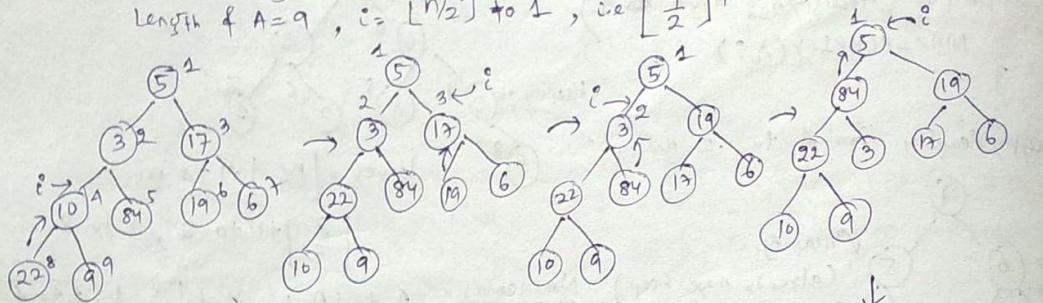
5. MAX-HEAPIFY(A, 1).

HEAPSORT(A) = {15, 12, 10, 8, 7, 9, 3, 2, 4, 13}

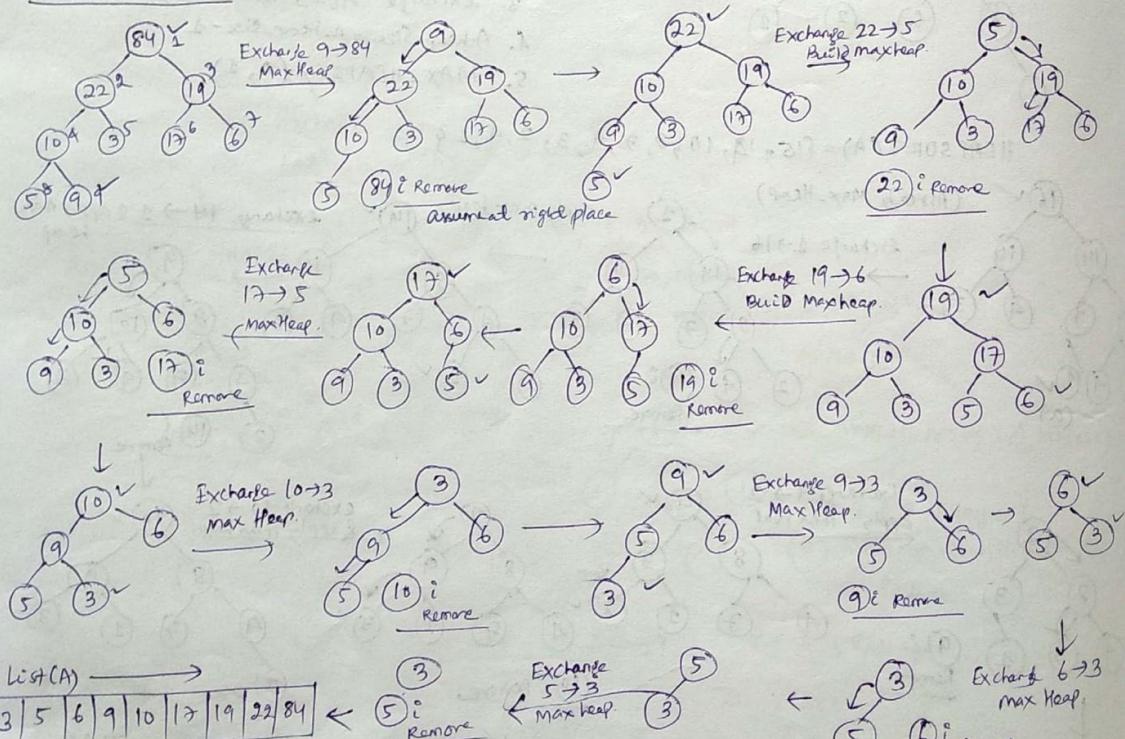


Example - 2 :

Sort the following element using Heap Sort.  $A[5, 3, 17, 10, 84, 19, 6, 22, 9]$   
 Length of  $A = 9$ ,  $i = \lfloor n/2 \rfloor + 1$ , i.e.  $\lfloor \frac{9}{2} \rfloor + 1 = 5 + 1 = 6$



Now HeapSort ( $A$ )



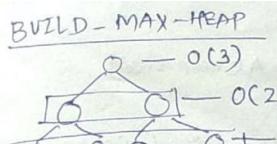
Time complexity of Heapsort

1. BUILD-MAX-HEAP —  $O(n)$
2. for  $i = A.length$  down to 2
3. Exchange  $A[1]$  with  $A[i]$  —  $(n-1)$  each up to  $(n-1)$  calls to Heapify  $\Rightarrow O(n) + O(n\log n)$
4.  $A.heapsize = A.heapsize - 1$
5. MAX-HEAPIFY ( $A, 1$ ) —  $O(\log n)$  [Dependent Height of Tree]  $\Rightarrow O(n\log n)$   
 Worst-Case

Space complexity =  $O(\log n)$   
 [Recursive Stack]

$$\text{ie } O(n) + (n-1)O(\log n)$$

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



no. of nodes at height 'h'  $\left\lceil \frac{n}{2^{h+1}} \right\rceil$  i.e.  $\left\lceil \frac{15}{2^{0+1}} \right\rceil = \left\lceil \frac{15}{2} \right\rceil = 8$

so we can write the time complexity as  $\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$  The work done by max heap at every level different

i.e.  $\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$  constant  $\frac{cn}{2} \sum_{h=0}^{\infty} \left( \frac{n}{2^h} \right)$  Implementing Integration & Differentiation Rule.

$$\Rightarrow \frac{cn}{2} \sum_{h=0}^{\infty} \left( \frac{n}{2^h} \right) \Rightarrow \frac{1}{(1-\frac{1}{2})^2} = 2 \text{ i.e. } \frac{cn}{2} \times 2 = O(n)$$

Time complexity of BUILD-MAX-HEAP  $\Rightarrow O(n)$

The Heap provides an efficient implementation of the Priority Queue.

### Priority Queue

It depends on priority of elements. The element with highest priority will be moved to the front of the Queue and one with lowest priority will move to the back of the Queue.

#### 2 Kinds of Priority Queue

1) Max-Priority Queue

2) Min-Priority Queue

Max-Priority Queue based on Max-Heap. Min Priority Queue based on Min-Heap. A priority queue is a data structure for maintaining a set of 'S' of elements, each with an associated value called a key. A max-priority queue supports the following operations →

1) Insert( $S, x$ ) — insert the element of  $x$  into the set  $S$ , which is equivalent to the operation  $S = S \cup \{x\}$ .

2) Maximum( $S$ ) — returns the element of  $S$  with largest key.

3) Extract-Max( $S$ ) — removes and returns the element of  $S$  with the largest key.

4) Increase-Key( $S, x, k$ ) — increase the value of element  $x$ 's key to the new value  $k$ , which is assumed to be at least as large as  $x$ 's current key value.

→ we can use max-priority queues to schedule jobs on a shared computer.

Min-priority queue: Insert, Minimum, Extract-Min, Decrease-Key.

It supports the operations Insert, Minimum, Extract-Min.

It is used in event-driven simulators.

### (3) HEAP-EXTRACT-MAX(A)

if  $A.\text{heap\_size} < 1$  / heap is at least one element  
error "heap underflow"

$\maxA[1]$

$A[1] = A[A.\text{heap\_size}]$

$A.\text{heap\_size} = A.\text{heap\_size} - 1$

$\text{MAX-HEAPIFY}(A, 1)$

3 return max

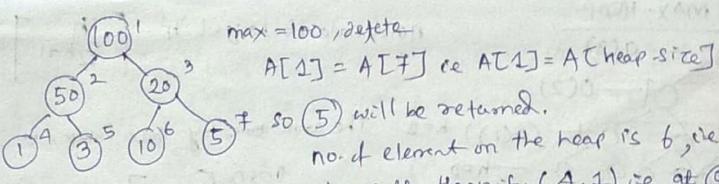
Time taken for max-HEAPIFY —

Height of tree =  $O(\log n)$

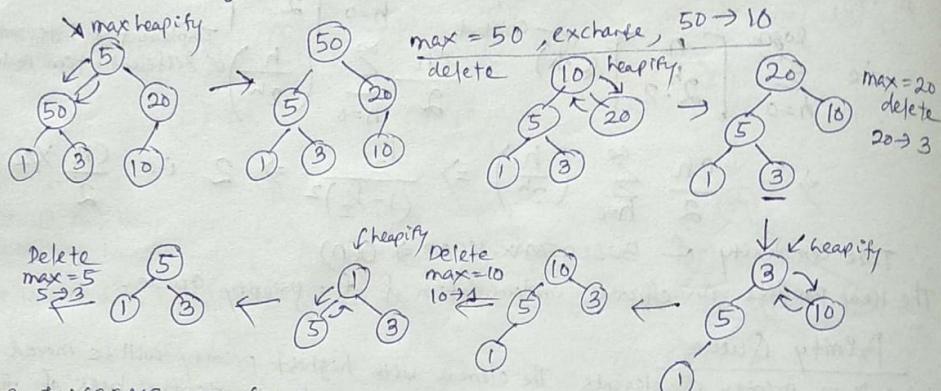
+ we call HEAPIFY at the Root (always)

Space complexity  $O(\log n)$

Example:



no. of element on the heap is 6, i.e.  $\text{heap-size} - 1$   
then call  $\text{Heapify}(A, 1)$  i.e. at 5



#### (A) HEAP-INCREASE-KEY ( $A, i, \text{key}$ )

if  $\text{key} < A[i]$  // increase the key not decrease

error "new key is smaller than current key"

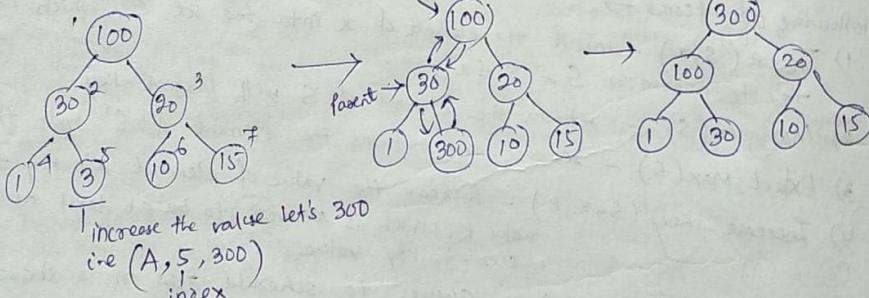
$A[i] = \text{key}$

while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$  // if parent is less exchange

exchange  $A[i]$  with  $A[\text{PARENT}(i)]$  // Parent =  $i/2$

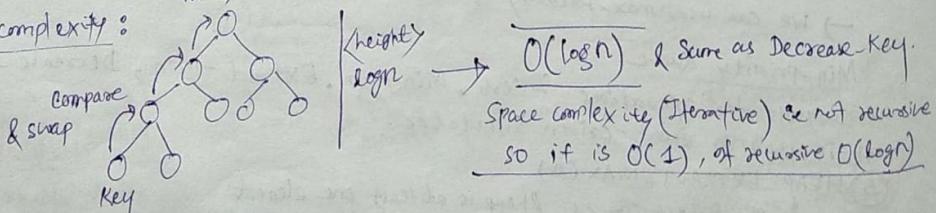
$i = \text{PARENT}(i)$  // check with all the parent

Example:



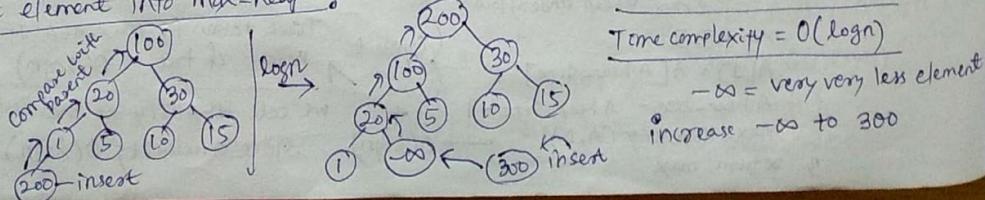
increase the value let's 300  
i.e.  $(A, 5, 300)$   
index

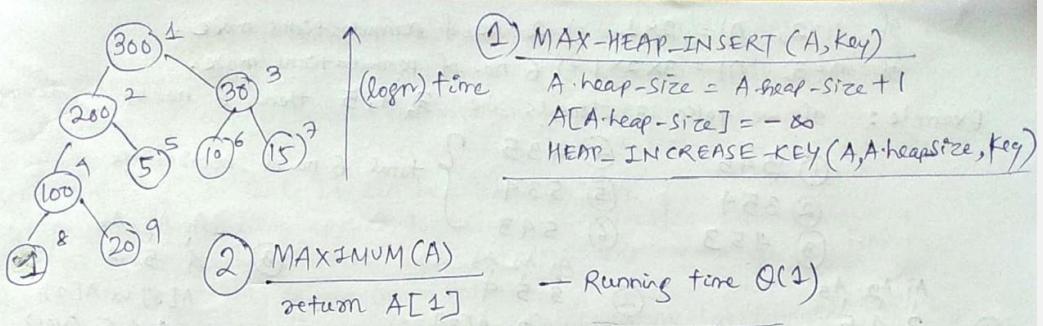
Time complexity:



Space complexity (Iterative) is not recursive  
so if is  $O(1)$ , of recursive  $O(\log n)$

Insert element into max-heap:





Note:

| Max Heap | Find max | Delete max  | Insert                                 | Increase Key   | Decrease Key |                  |
|----------|----------|-------------|--|--|--------------|------------------|
|          | $O(1)$   | $O(\log n)$ | $O(\log n)$                            | $O(\log n)$  | $O(\log n)$  | call max-heapify |
|          | $O(n)$   | $O(n)$      | $O(n)$<br>i.e. $O(n \cdot n) = O(n^2)$ | $O(n)$<br>$O(n) + \text{for build the heap again } O(n)$ |              |                  |

### Lower Bounds for Sorting :

A lower bound of a problem is the least time complexity required for any algorithm. Lower bound proves that we can not hope for a better algorithm.

The problem of sorting can be viewed as:

Input: A sequence of  $n$  numbers  $\langle a_1, a_2, a_3, \dots, a_n \rangle$

Output: A permutation (reordering)

A sorting algorithm is comparison based if it uses comparison operators to find the order between two numbers. Comparison sorts can be viewed in terms of decision trees. A decision tree is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm.

The execution of the sorting algorithm corresponds to traversing a path from the root of the decision tree to a leaf.

At each internal node, a comparison  $a_i \leq a_j$  is made.

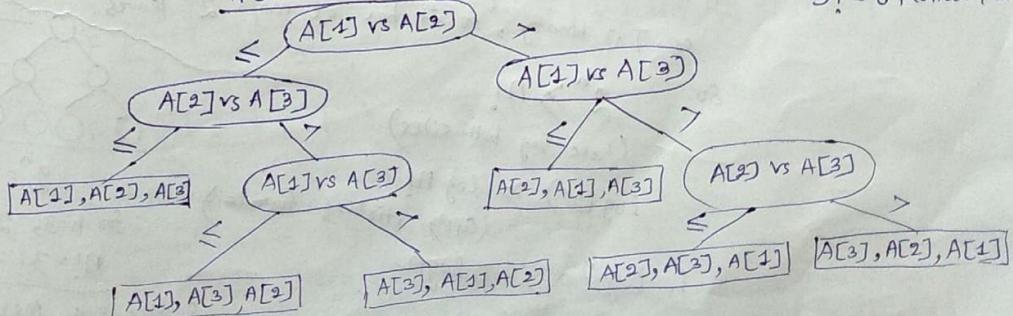
→ At each internal node, a comparison  $a_i < a_j$

→ Left subtree  $a_i < a_j$

→ Right subtree  $a_i > a_j$

The Decision Tree Model (Insertion sort with  $N=3$ )

$3! = 6$  permutations



Note: Here all the input elements are distinct.  
For  $n$  elements  $n!$  permutations must appear.

If  $n=2$ ,  $n! = 2 \times 1 = 2$  no. of permutations made.

If  $n=3$ ,  $n! = 3 \times 2 \times 1 = 6$  no. of permutations made.

Example: If we take 3 elements i.e. 3, 4, 5 then 6 no. of reorderings are:

$$\begin{array}{ll} \textcircled{1} \quad 345 & \textcircled{4} \quad 435 \\ \textcircled{2} \quad 354 & \textcircled{5} \quad 534 \\ \textcircled{3} \quad 453 & \textcircled{6} \quad 543 \end{array} \quad \left. \begin{array}{l} \textcircled{4} \quad 435 \\ \textcircled{5} \quad 534 \\ \textcircled{6} \quad 543 \end{array} \right\} \text{total 6 no. of reorderings}$$

$$\begin{array}{l} \textcircled{1} \quad A_1 \ A_2 \ A_3 \\ \quad 3 \ 4 \ 5 \\ A[1] \text{ vs } A[2] \\ 3 \leq 4 \ (\text{yes}) \\ A[2] \text{ vs } A[3] \\ 4 \leq 5 \ (\text{yes}) \\ \text{(Arrange in ascending)} \\ \boxed{A[1], A[2], A[3]} \end{array}$$

$$\begin{array}{l} \textcircled{2} \quad A_1 \ A_2 \ A_3 \\ \quad 3 \ 5 \ 4 \\ A[1] \text{ vs } A[2] \\ 3 \leq 5 \ (\text{yes}) \\ A[2] \text{ vs } A[3] \\ 5 > 4 \ (\text{yes}) \\ A[1] \text{ vs } A[3] \\ 3 \leq 4 \ (\text{yes}) \\ \boxed{A[1], A[3], A[2]} \end{array}$$

$$\begin{array}{l} \textcircled{3} \quad A_1 \ A_2 \ A_3 \\ \quad 4 \ 5 \ 3 \\ A[1] \text{ vs } A[2] \\ 4 \leq 5 \ (\text{yes}) \\ A[2] \text{ vs } A[3] \\ 5 > 3 \ (\text{yes}) \\ A[1] \text{ vs } A[3] \\ 4 > 3 \ (\text{yes}) \\ \boxed{A[3], A[1], A[2]} \end{array}$$

$$\begin{array}{l} \textcircled{4} \quad A_1 \ A_2 \ A_3 \\ \quad 4 \ 3 \ 5 \\ A[1] \text{ vs } A[2] \\ 4 > 3 \ (\text{yes}) \\ A[1] \text{ vs } A[3] \\ 4 \leq 5 \ (\text{yes}) \\ \boxed{A[2], A[1], A[3]} \end{array}$$

$$\begin{array}{l} \textcircled{5} \quad A_1 \ A_2 \ A_3 \\ \quad 5 \ 3 \ 4 \\ A[1] \text{ vs } A[2] \\ 5 > 3 \ (\text{yes}) \\ A[1] \text{ vs } A[3] \\ 5 > 4 \ (\text{yes}) \\ A[2] \text{ vs } A[3] \\ 3 \leq 4 \ (\text{yes}) \\ \boxed{A[2], A[3], A[1]} \end{array}$$

$$\begin{array}{l} \textcircled{6} \quad A_1 \ A_2 \ A_3 \\ \quad 5 \ 4 \ 3 \\ A[1] \text{ vs } A[2] \\ 5 > 4 \ (\text{yes}) \\ A[1] \text{ vs } A[3] \\ 5 > 3 \ (\text{yes}) \\ A[2] \text{ vs } A[3] \\ 4 > 3 \\ \boxed{A[3], A[2], A[1]} \end{array}$$

From the above, each of the  $n!$  permutations on  $n$  elements must appear as one of the leaves of the decision tree for the sorting algorithm to sort properly.

Theorem: Any decision tree  $T$  for sorting  $n$  elements has height  $\lceil 2(n \log n) \rceil$  [so any comparison sort algorithm requires  $\lceil 2(n \log n) \rceil$  compares in the worst case]

Proof: 1)  $n!$  permutations for  $n$  elements

2) Let  $X$  be the minimum no. of comparisons in a sorting algorithm.

The maximum height of the decision tree would be  $X$ .

A tree with maximum height  $X$  has at most  $2^X$  leaves

as  $T$  is binary, so it has at most  $2^h$  leaves.

So  $2^h \geq n!$

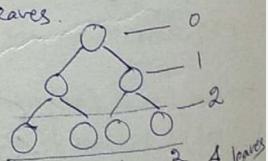
(take log both sides)

$\log(2^h) \geq \log n!$   
(apply Stirling's formula)

$h \log_2 2 \geq n \log n$

$h \geq n \log n$

$h = \lceil 2(n \log n) \rceil$  [Proved]



$$2^h = 2^3 = 8$$

$$n! = 3! = 6$$

So Heapsort, Mergesort are having  $(n \log n)$  Time complexity at worst case.

## Dynamic Programming (Richard Bellman)

Dynamic Programming solves problem by combining the solutions to subproblem.

Dynamic programming applies when subproblem overlap.

Dynamic programming algorithm solves each subproblem just once and then saves its answer in table, so that it can be used in future.

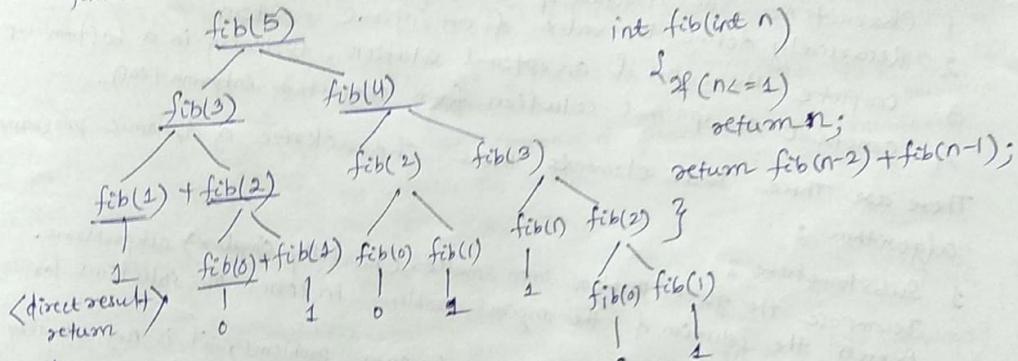
Dynamic programming mainly applied to optimization problems.  
Optimization means each solution has a value and we wish to find a solution with minimum or maximum value.

Dynamic Programming usually use/solved by Recursive method/formula.  
It follows principle of optimality & problem can be solved by taking decisions.  
Every stage we take a decision.

Example: Fibonacci Series (0 1 1 2 3 5 ...)

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

Recursive function



(Tracing Tree of fib(5)) — Total 15 calls.

Recurrence Relation:  $T(n) = 2T(n-1) + 1 \Rightarrow O(2^n)$  // so much of Time.

fib(1) called - 5 times (same function called again and again)

fib(2) called - 3 times

Reduce the function call & time by storing the results of Fibonacci functions

So that once fib(2) called, should not call again.

$F[-1, -1, -1, -1, -1, -1]$  initially fill -1, nothing is known  
 $\downarrow$

$F[0, 1, 1, 2, 3, 5]$  Store it in an array.

if we apply this, total 6 calls required (undesired calls in tree)

$$\text{fib}(n) = n+1 \text{ calls. i.e. } \approx O(n)$$

Storing the results called — memoization.

Here we see that  $2^n$  reduce to  $n$  (Exponential to linear)

Memoization follows Top-down approach.

Same Problem with iterative method. [Tabulation Method]

$\text{fib}(5), n=5$

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

Table is generated by the functions.

Here filling is done from smaller value onwards.  
Called [Bottom up approach]

Mostly Tabulation method is used in Dynamic Programming

int fib(int n)

{ if ( $n \leq 1$ )

return  $n;$

$F[0]=0, F[1]=1;$

for (int i=2; i<=n; i++)

{  $F[i] = F[i-2] + F[i-1];$

}

return  $F[n];$

}

### Iterative Method.

When developing a dynamic-programming algorithm, we follow a sequence of 4 steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

There are three basic elements that characterize a dynamic programming algorithm :

1. Substructure :

Decompose the given problem into smaller (hopefully simpler) subproblems.  
Express the solution of the original problem in terms of solutions for smaller problems.

Note that unlike divide-and-conquer problems, it is not usually sufficient to consider one decomposition, but many different ones.

2. Table Structure :

After solving the sub-problems, store the results to the subproblems in a table. Subproblems' solutions are reused many times, we do not want to repeatedly solve the same problem over and over again.

3. Bottom-Up Computation -

Using table (or something), combine solutions of smaller subproblems to solve larger subproblems, and eventually arrive at a solution to the complete problem.

Bottom-up means :

- i) Start with the smallest subproblems.
- ii) Combining their solutions obtain the solutions to subproblems of increasing size.
- iii) Until arrive at the solution of the original problem.

Examples :

- 1) Matrix-chain multiplication problem.
- 2) Longest common subsequence problem.
- 3) Assembly-line scheduling problem.

## Matrix-Chain Multiplication

The main objective of matrix chain multiplication problem is "to find the optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplications are minimized.

Rule for matrix multiplication : The no. of column of first matrix  
= no. of rows of 2<sup>nd</sup> matrix

$$\begin{bmatrix} A \\ \vdots \vdots \vdots \vdots \end{bmatrix}_{5 \times 4} \begin{bmatrix} B \\ \vdots \vdots \vdots \vdots \end{bmatrix}_{6 \times 3} = \text{result matrix} = \begin{bmatrix} C \\ \vdots \vdots \vdots \vdots \end{bmatrix}_{5 \times 3} = 15 \text{ elements}$$

Each element in 'C' matrix require 4-multiplication, Total multiplication =  $15 \times 4 = 60$

For performing chain matrix multiplication i.e.  $A_1 \times A_2 \times A_3 \dots A_n$   
we should use parenthesis.

Example: Consider 3 matrices of  $A_{5 \times 3}$   $B_{3 \times 4}$   $C_{4 \times 6}$

Possible way of multiplication by using parenthesis

$$((A \cdot B) \cdot C) = ((A_{5 \times 3} \cdot B_{3 \times 4}) \cdot C_{4 \times 6}) = (5 \times 3 \times 4) + (5 \times 4 \times 6) = 60 + 120 = 180 \text{ multiplication}$$

$$(A \cdot (B \cdot C)) = (A_{5 \times 3} \cdot (B_{3 \times 4} \cdot C_{4 \times 6})) = (5 \times 3 \times 6) + (3 \times 4 \times 6) = 90 + 72 = 162 \text{ multiplication.}$$

So we have seen that by changing the sequence of parenthesis the number of multiplication changes. So we require the optimal solution.

Example:  $A_1 \quad A_2 \quad A_3 \quad A_4$   
 $5 \times 4 \quad 4 \times 6 \quad 6 \times 2 \quad 2 \times 7$

Aim: take a pair and multiply, which pair I have to select such that the total cost of multiplying all of them is minimum.

Use parenthesis:  $A_1 A_2 A_3 A_4$  can be fully parenthesized in 5 distinct ways

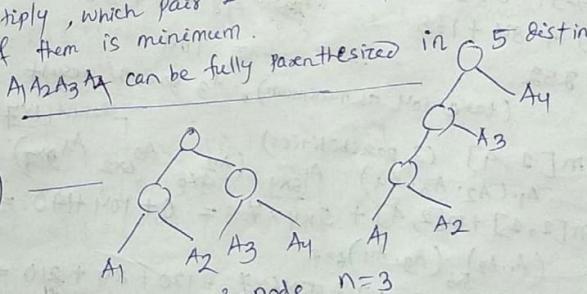
$$(((A_1 \cdot A_2) \cdot A_3) \cdot A_4)$$

$$((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$$

$$(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$$

$$(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$$

$$((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$$



It requires 3-node,  $n=3$

$$\text{Formula} = \frac{(2n)!}{(n+1)! \cdot n!} = \frac{(2 \times 3)!}{(3+1)! \cdot 3!} = \frac{6!}{4! \times 3!}$$

$$\Rightarrow \frac{6 \times 5 \times 4!}{4! \times 3 \times 2 \times 1} = 5 \text{ distinct ways.}$$

The matrix chain multiplication stated as follows :

Given a chain  $\langle A_1, A_2 \dots A_n \rangle$  of  $n$  matrices for ( $i=1 \dots n$ ) and matrix  $A_i$  has dimensions  $P_{i-1} \times P_j$ , fully Parenthesizing the product in a way that minimizes the no. of scalar multiplications. Let  $M[i, j] =$  minimum no. of scalar multiplications needs to compute matrix  $A_i \dots A_j$  i.e.  $(A_i \cdot A_{i+1} \dots A_j) = (A_i \cdot A_{i+1} \dots A_k) \cdot (A_{k+1} \dots A_j)$   
if  $i=j$ , then  $M[i, i] = 0$ , otherwise  $M[i, j] = M[i, k] + M[k+1, j] + P_{i-1} \cdot P_k \cdot P_j$   
where  $k=i$  to  $j-1$  and  $i < k$

Table Method & Bottom-up Approach.

$P_0, P_1, P_2, P_3, P_4$  - dimensions

$$\begin{array}{ccccccc}
 & A_1 & . & A_2 & . & A_3 & . & A_4 \\
 & 5 \times 4 & & A_1 \times 6 & 6 \times 2 & & 2 \times 7 \\
 & B & P_1 & P_2 & P_3 & P_4 & \\
 A_1 & & A_2 & & A_3 & & A_4 \\
 M[1,1] = 0 & M[2,2] = 0 & M[3,3] = 0 & M[4,4] = 0
 \end{array}$$

$$\begin{array}{ccc}
 M[1,2] & M[2,3] & M[3,4] \\
 A_1 \cdot A_2 & A_2 \cdot A_3 & A_3 \cdot A_4 \\
 5 \times 4 & 4 \times 6 & 6 \times 2 & 2 \times 7 \\
 5 \times 4 \times 6 = 120 & 4 \times 6 \times 2 = 48 & 6 \times 2 \times 7 = 84
 \end{array}$$

$$\begin{aligned}
 & \text{min} \left\{ \begin{array}{l} M[1,3] \text{ (2 possibilities)} \\ \text{ie } A_1 \cdot (A_2 \cdot A_3) \\ (5 \times 4) \times (4 \times 6) \times (6 \times 2) \end{array} \right. & & \text{Find minimum.} \\
 & M[1,1] + M[2,3] + 5 \times 4 \times 2 & & \\
 & = 0 + 48 + 40 & & \text{dimensions} \\
 & = 88 & & \\
 & \text{(take minimum, 88), } A_1 \text{ gives the result} & &
 \end{aligned}$$

$$\begin{aligned}
 & \text{min} \left\{ \begin{array}{l} M[2,4] \text{ (2 possibilities)} \\ \text{Find minimum.} \\ A_2 \cdot (A_3 \cdot A_4) \\ (4 \times 6) \times (6 \times 2) \times (2 \times 7) \end{array} \right. \\
 & M[2,2] + M[3,4] + 4 \times 6 \times 7 & & \\
 & 0 + 84 + 168 & & \\
 & = 252 & & \\
 & \text{(take 104 as minimum), } A_3 \text{ gives the result} & &
 \end{aligned}$$

$$\begin{aligned}
 & \text{min} \left\{ \begin{array}{l} M[1,4] \text{ (3 possibilities)} \\ A_1 \cdot (A_2 \cdot A_3 \cdot A_4) \\ A_1 \times 6 \times 4 \times 6 \times 2 \times 7 \end{array} \right. & & \text{Result } [4 \times 7] \\
 & M[1,1] + M[2,4] + 5 \times 4 \times 7 = 0 + 104 + 140 = 244 & & \text{the one which giving minimum} \\
 & (A_1 \cdot A_2) \cdot (A_3 \cdot A_4) \times 6 \times 7 & & \\
 & M[1,2] + M[3,4] + 5 \times 6 \times 7 = 120 + 84 + 210 = 414 & & \\
 & (A_1 \cdot A_2 \cdot A_3) \cdot A_4 \times 2 \times 7 & & \\
 & M[1,3] + M[4,4] + 5 \times 2 \times 7 = 88 + 0 + 70 = 158 \text{ (minimum)} & & \\
 & A_3 \text{ gives the result} & &
 \end{aligned}$$

From the above, state the formula.

$$m[i, j] = \min \{ m[i, k] + m[k, j] + P_{i-1} \cdot P_k \cdot P_j \}$$

Time complexity of matrix chain multiplication is, consider Table 1.,  $1+2+3+\dots+(n-1)$ , diagonal 'O'

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

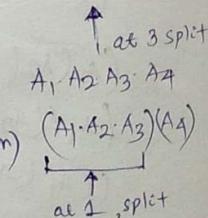
Each element we get by calculating all & find the minimum takes at most ' $n$ ' times.

$$n^2 \cdot n = n^3 = O(n^3)$$

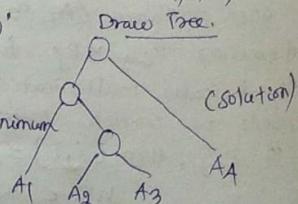
|   | 1 | 2   | 3  | 4   |
|---|---|-----|----|-----|
| 1 | 0 | 120 | 88 | 158 |
| 2 |   | 0   | 48 | 104 |
| 3 |   |     | 0  | 84  |
| 4 |   |     |    | 0   |

Table 1

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 1 | 3 |   |
| 2 |   | 2 | 3 |   |
| 3 |   |   | 3 |   |
| 4 |   |   |   |   |



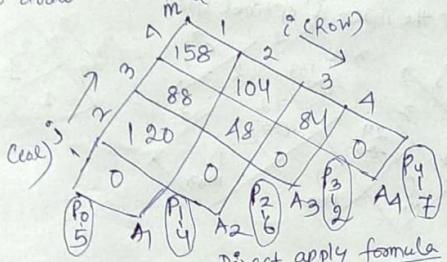
$((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$   
at 2, split, no need.



Time complexity for multiplying  $n$  matrix is  $n^3$   
 & solving the parenthesis problem is also  $n^3$

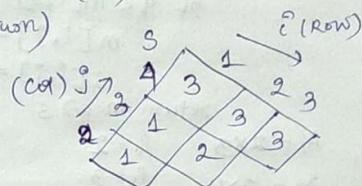
so Time complexity of matrix multiplication =  $O(n^3)$

We also draw the M & S Table as : (Just Tern)



Direct apply formula :-

Space complexity =  $O(n^2)$



Note: Algorithm requires  $O(n^2)$  space  
 to store M and S tables.

$$\begin{aligned}
 m[1,2] &= m[1,1] + m[2,2] + P_0 \cdot P_1 \cdot P_2 = 0 + 0 + 5 \times 4 \times 6 = 120 \\
 m[2,3] &= m[2,2] + m[3,3] + P_1 \cdot P_2 \cdot P_3 = 0 + 0 + 4 \times 6 \times 2 = 48 \\
 m[3,4] &= m[3,3] + m[4,4] + P_2 \cdot P_3 \cdot P_4 = 0 + 0 + 6 \times 2 \times 7 = 84 \\
 m[1,3] &= \min \left\{ m[1,1] + m[2,3] + P_0 \cdot P_1 \cdot P_3 \right\} \Rightarrow K=1 \\
 m[2,4] &= \min \left\{ m[2,2] + m[3,4] + P_1 \cdot P_2 \cdot P_4 \right\} \Rightarrow K=2 \\
 &= \min \left\{ 0 + 84 + 4 \times 6 \times 7 = 252 \quad \text{at } A_3, \text{ minimum} \right. \\
 &\quad \left. 48 + 0 + 6 \times 2 \times 7 = 104 \right\} \checkmark
 \end{aligned}$$

$$\begin{aligned}
 m[1,4] &= \min \left\{ m[1,1] + m[2,4] + P_0 \cdot P_1 \cdot P_4 \right\} \Rightarrow K=2 \\
 &\quad \left\{ m[1,2] + m[3,4] + P_0 \cdot P_2 \cdot P_4 \right\} \Rightarrow K=3 \\
 &\quad \left\{ m[1,3] + m[4,4] + P_0 \cdot P_3 \cdot P_4 \right\} \Rightarrow K=3 \\
 &= \min \left\{ 0 + 104 + 5 \times 4 \times 7 = 244 \quad \text{at } A_3, \text{ minimum} \right. \\
 &\quad \left. 120 + 84 + 5 \times 6 \times 7 = 414 \right. \\
 &\quad \left. 88 + 0 + 5 \times 2 \times 7 = 158 \right\} \checkmark
 \end{aligned}$$

### Algorithm

MATRIX-CHAIN-ORDER(P) *<To find M & S table>*

$n = p.length - 1$   
 let  $m[1..n, 1..n]$  and  $s[1..n-1, 2..n]$  be new tables

for  $i = 1$  to  $n$   
 $m[i, i] = 0$  // one matrix no multiplying

$m[i, i] = 0$  //  $i$  is the chain length

for  $l = 2$  to  $n$

for  $i = 1$  to  $n-l+1$   
 $j = i+l-1$  // end index for chain of length  $l$

$m[i, j] = \infty$

for  $K = i + j - 1$

$$q = m[i, K] + m[K+1, j] + p_{i-1} p_K p_j$$

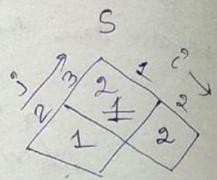
if  $q < m[i, j]$

$$m[i, j] = q$$

$s[i, j] = K \uparrow$  the index of optimal split

return  $m$  and  $s$

Example  $A_1 = 2 \times 3 \quad p_0 = 2$   
 $A_2 = 3 \times 4 \quad p_1 = 3$   
 $A_3 = 4 \times 6 \quad p_2 = 4$   
                         $p_3 = 6$



①  $n = 3 \quad \text{i.e. } p-1 (4-1) = 3$

②  $i = i \rightarrow 3$

③  $m[i, i] = 0$

④ for  $l \leftarrow 2 \rightarrow 3$

⑤ for  $i \leftarrow 1 \rightarrow 2$

⑥  $j = 2$

⑦  $m[2, 2] = \infty$

$2 \times 3 \times 4 = 24$

⑧  $K \rightarrow 1 \rightarrow 1$

⑨  $q = m[1, 1] + m[2, 2] + p_0 p_1 p_2$

⑩  $24 < \infty$

⑪  $m[1, 2] = 24$

⑫  $s[1, 2] = 1$

⑬ return 24      ⑭ return 1

⑮ for  $i=2$

⑯  $j = 2 + 2 - 1 = 3$

⑰  $m[2, 3] = \infty$

$3 \times 4 \times 6 = 72$

⑱  $K \leftarrow 2 \rightarrow 2$

⑲  $q = m[2, 2] + m[3, 3] + p_1 p_2 p_3$

⑳  $72 < \infty$

㉑  $m[2, 3] = 72$

㉒  $s[2, 3] = 2$

㉓ return 72, return 2

㉔ for  $l=3$

㉕  $i \leftarrow 1 \rightarrow 1$

㉖  $j = 3$

㉗  $m[1, 3] = \infty$

$2 \times 3 \times 6 = 36 \rightarrow 108$

㉘  $K \leftarrow 1 \rightarrow 2$

㉙  $q = m[1, 1] + m[2, 3] + p_0 p_1 p_3$

⑩  $108 < \infty$   
 ⑪  $M[1, 3] = 108$   
 ⑫  $S[1, 3] = 1$   
 ⑬ return 108, return 1  
 ⑭  $K=2$        $24$   
 ~~$2 = M[1, 2] + M[3, 3] + (P_0 P_2 P_3)$~~        $2 \times 4 \times 6 = 48 \rightarrow 72$   
 ⑮  $72 < 108$

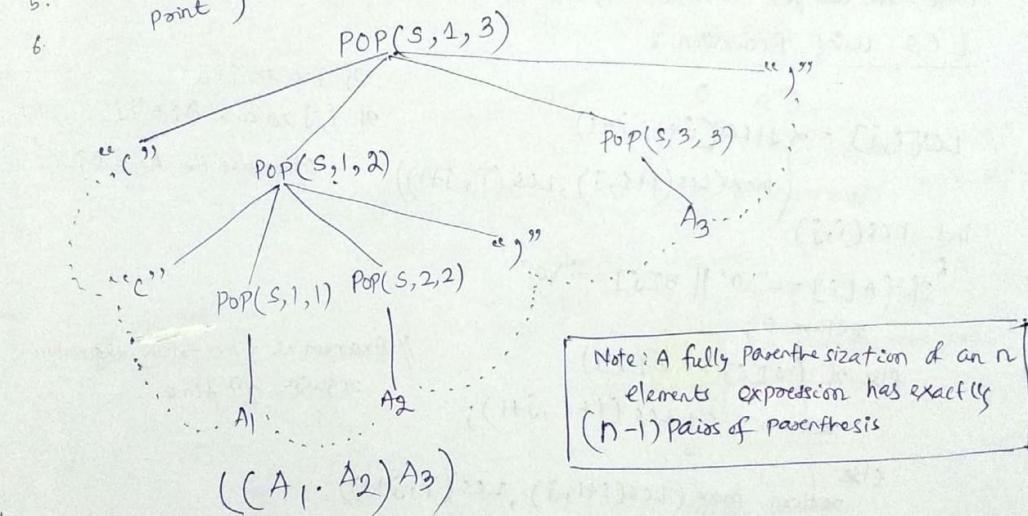
⑯  $M[1, 3] = 72$

⑰  $S[1, 3] = 2$

⑱ return 72, return 2

PRINT-OPTIMAL-PAREN(S, i, j) [Pointing the optimal parenthesis]

1. if  $i=j$   
point "A"?
2. else point "("
3. PRINT-OPTIMAL-PAREN(S, i, S[i, j])
4. PRINT-OPTIMAL-PAREN(S, S[i, j]+1, j)
5. point ")"
- 6.



### Longest Common Subsequence (LCS)

In LCS problem we are given 2 sequences  $X$  &  $Y$  where,

$$X = \langle x_1, x_2, \dots, x_m \rangle \quad Y = \langle y_1, y_2, \dots, y_n \rangle$$

and we have to find a maximum length common subsequence of  $X$  &  $Y$ .

$$\text{i.e } Z = \{z_1, z_2, \dots, z_k\}$$

Biological applications often need to compare the DNA of two or more different organisms. A strand of DNA consists of a string of molecules called bases, where possible bases are adenine, guanine, cytosine & thymine. We can express a strand of DNA as  $\{A, C, G, T\}$ . For example DNA of one organism may be  $S_1 = ACCGGT$ , & other may be  $S_2 = GTGAT$ . One reason to compare

two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are.

"A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous".

Sequences = "abcde fg", "abx dfg"

Common subsequences = "a", "b", "d", "f", "g", "ab", "df", "dfg", "abd"

"abcd fg"

Longest common subsequence = "abcd fg"

$X = \underline{a} b c d e f g h i j$   
 $Y = \underline{e} c d g i$   
 (skip) | { intersect, not in order  
 (skip)}

$X = \underline{a} b c d e f g h i j$   
 $Y = \underline{e} \underline{c} d g i$   
 skip

$X = a b d a c e$   
 $Y = a b c e$   
 {abc e}  
 $X = a b d a c e$   
 $Y = b \underline{a} b c e$   
 skip  
 {abc e}

Note: we can get multiple sequences with same length of subsequence.

LCS using Recursion:

$$LCS[i:j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + LCS[i+1, j+1] & \text{if } i, j > 0 \text{ and } A_i = B_j \\ \max(LCS(i+1, j), LCS(i, j+1)) & \text{otherwise i.e. } A_i \neq B_j \end{cases}$$

int LCS(i, j)

{ if ( $A[i] == '0' \text{ || } B[j] == '0'$ )  
 return 0;  
 else if ( $A[i] == B[j]$ )  
 return 1 + LCS(i+1, j+1);  
 else  
 return max (LCS(i+1, j), LCS(i, j+1));  
}

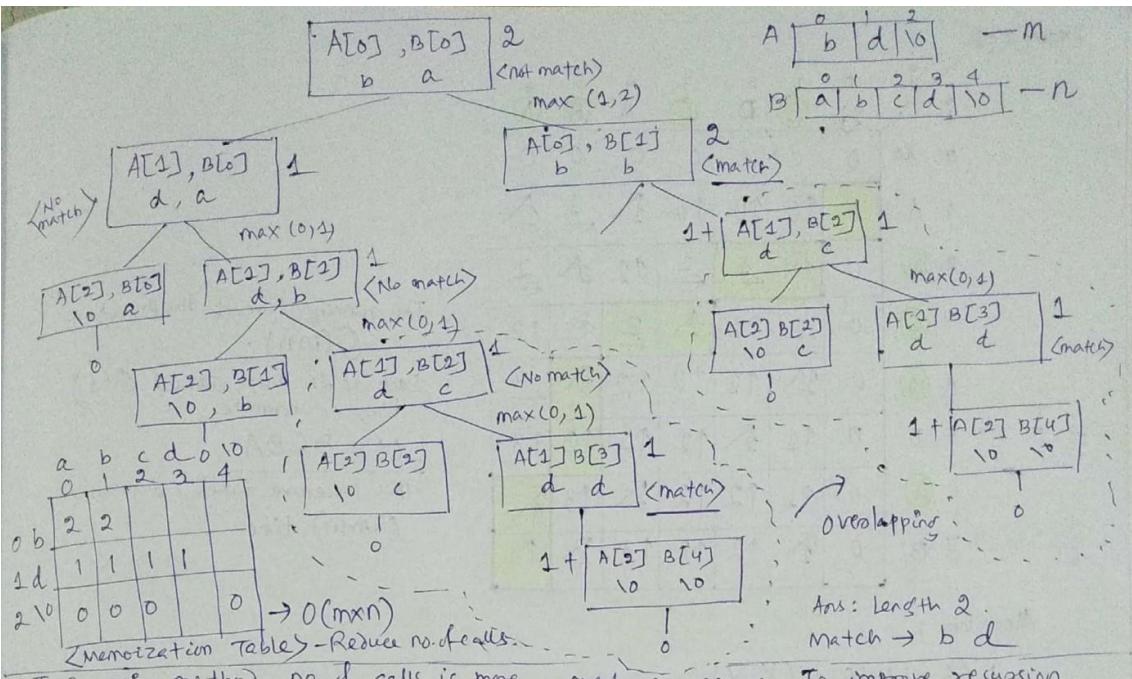
The algorithm says, if there is a match, skip 1 and move to the next alphabet for both A & B.

if there is no match, check either way more alphabet in 1<sup>st</sup> string A and more alphabet in 2<sup>nd</sup> string B.

Example:

|   |   |   |   |
|---|---|---|---|
| A | b | d | 0 |
|   | 0 | 1 | 2 |

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| B | a | b | c | d | 0 |
|   | 0 | 1 | 2 | 3 | 4 |



In recursive method no. of calls is more, overlapping occurs. To improve recursion we use memoization. In case of memoization remember the answer in a table & use in future. Reduce the time complexity from  $2^n$  to  $(n \times n)$ .

Note: Memoization (overlapping not occur)

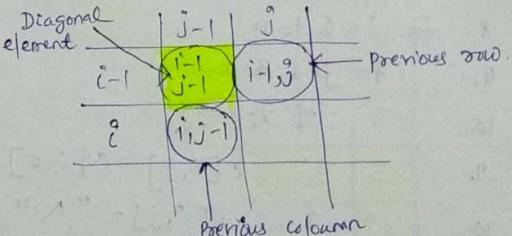
## Dynamic Programming

Dynamic Programming :-

$$LCS[i, j] = \begin{cases} LCS[i-1, j-1] + 1 & \text{if } i=0 \text{ or } j=0 \\ \max(LCS[i-1, j], LCS[i, j-1]) & i, j > 0 \text{ and } A_i = B_j \\ & i, j > 0 \text{ and } A_i \neq B_j \end{cases}$$

diagonal element  
Previous row  
Previous column

|   | a   | b | c   | d |
|---|-----|---|-----|---|
| 0 | 0   | 1 | 2   | 3 |
| b | 0   | 0 | 1   | 1 |
| d | 2   | 0 | 1   | 1 |
|   | (b) |   | (d) |   |



| Example - 1 |   | C (A, B, C) |    |    |    |    |    |
|-------------|---|-------------|----|----|----|----|----|
|             |   | 0           | 0  | 0  | 0  | 0  | 0  |
| 1           | A | 0           | 1↑ | 1  | 1  | 1  | 1  |
| 2           | E | 0           | 1↑ | 1↑ | 1↑ | 1↑ | 1↑ |
| 3           | D | 0           | 1↑ | 1↑ | 1↑ | 2↑ | 2  |
| 4           | H | 0           | 1↑ | 1↑ | 1↑ | 2↑ | 3↑ |
| 5           | F | 0           | 1↑ | 1↑ | 1↑ | 2↑ | 3↑ |

$$Y = ABCDH$$

$$X = AE \Delta HF$$

$$LCS = ADH$$

Example - 2

| $i$ | 0 | 1 | 2   | 3   | 4   | 5   | 6   |
|-----|---|---|-----|-----|-----|-----|-----|
| $j$ | 0 | B | D   | C   | A   | B   | A   |
| 0   | x | 0 | 0   | 0   | 0   | 0   | 0   |
| 1   | A | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↓ 1 | ↓ 1 |
| 2   | B | 0 | ↖ 1 | ↖ 1 | ↑ 1 | ↑ 2 | ↖ 2 |
| 3   | C | 0 | ↑ 1 | ↑ 1 | ↖ 2 | ↖ 2 | ↑ 2 |
| 4   | D | 0 | ↑ 1 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 |
| 5   | A | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 3 | ↖ 3 |
| 6   | B | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 |

The running time of the procedure  
 $\cdot O(mn)$ .

Each table entry takes  $O(1)$   
 Time to compute

$$LCS = BCBA$$

This procedure takes i.e. pointing  
 $O(m+n)$  time

Algorithm :

LCS-LENGTH( $X, Y$ )

1.  $m = X.length$
2.  $n = Y.length$
3. let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4. for  $i = 1$  to  $m$
5.      $c[i, 0] = 0$
6. for  $j = 0$  to  $n$
7.      $c[0, j] = 0$
8. for  $i = 1$  to  $m$
9.     for  $j = 1$  to  $n$
10.       if  $x_i == y_j$
11.            $c[i, j] = c[i-1, j-1] + 1$
12.            $b[i, j] = "↖"$
13.       elseif  $c[i-1, j] \geq c[i, j-1]$
14.            $c[i, j] = c[i-1, j]$
15.            $b[i, j] = "↑"$
16.       else
17.            $c[i, j] = c[i, j-1]$
18.            $b[i, j] = "←"$
19. return  $c$  and  $b$

This above algorithm requires  $O(mn)$  time complexity

PRINT-LCS(b, X, i, j)

1. if  $i=0$  or  $j=0$   
return
2. if  $b[i,j] == \text{"\n"}$
3. PRINT-LCS(b, X, i-1, j-1)
4. Point  $X_i$
5. else if  $b[i,j] == \text{"\u2191"}$
6. PRINT-LCS(b, X, i-1, j)
7. else PRINT-LCS(b, X, i, j-1)

The above procedure takes time  $O(m+n)$ , since it decrements at least one of  $i$  and  $j$  in each recursive call.

Example-3

Determine the LCS of  $X = \{P R E S I D E N T\}$  and  $Y = \{P R O V I D E N C E\}$

| $y_j$ | P | R | O | V | I | D | E | N | C | E |
|-------|---|---|---|---|---|---|---|---|---|---|
| $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P     | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| R     | 2 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| E     | 3 | 0 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 |
| S     | 4 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 3 | 3 |
| I     | 5 | 0 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| D     | 6 | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 |
| E     | 7 | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 5 | 5 |
| N     | 8 | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 5 | 6 |
| T     | 9 | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 5 | 6 |

Point-LCS(b, X, 9, 10)  $\uparrow$   
 Point-LCS(b, X, 8, 10)  $\leftarrow$   
 Point-LCS(b, X, 8, 9)  $\leftarrow$   
 Point-LCS(b, X, 8, 8)  $\uparrow$   
 Point-LCS(b, X, 7, 7)  $\uparrow$   
 Point-LCS(b, X, 6, 6)  $\uparrow$   
 Point-LCS(b, X, 5, 5)  $\nwarrow$   
 Point-LCS(b, X, 4, 4)  $\uparrow$

Point-LCS(b, X, 3, 4)  $\uparrow$   
 Point-LCS(b, X, 2, 4)  $\leftarrow$   
 Point-LCS(b, X, 2, 3)  $\leftarrow$   
 Point-LCS(b, X, 2, 2)  $\uparrow$   
 Point-LCS(b, X, 1, 1)  $\nwarrow$   
 Point-LCS(b, X, 0, 0) STOP  
 $Z = \{P R I D E N\}$   
 LCS-LENGTH(9, 10) = 6

## Assembly Line Scheduling

A car factory has 2 assembly lines, each with  $n$  stations. A station is denoted by  $S_{i,j}$  where  $i$  is either 1 or 2 and indicates the assembly line the station is on, and  $j$  indicates the number of station.

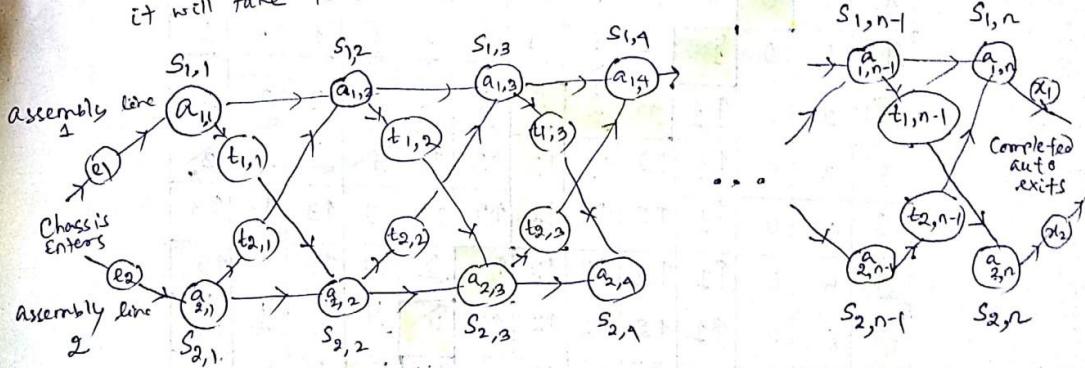
The time taken per station is denoted by  $a_{i,j}$ . Each station is dedicated to some sort of work like engine fitting, body fitting, painting and so on.

A car chassis must pass through each of the  $n$  stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station  $S_{i,j}$ , it will continue to station  $S_{i,j+1}$  unless it decides to transfer to the other line.

Continuing on the same line incurs no extra cost, but transferring from line 1 at station  $j-1$  to station  $j$  on the other line takes time  $t_{i,j}$ .

Each assembly line takes an entry time,  $e_1$ , and exit time  $x_1$  which may be different for the two lines.

So we have to design an algorithm for computing the minimum time it will take to build a car chassis.



Problem: Which stations should be chosen from line 1 and which from line 2 in order to minimize the total time through factory for one car?

✓ If we apply Brute force (enumerate all possibilities for selecting stations) technique then at each station require  $2, 4, 8, 16, \dots, 2^n$  no. of comparisons. So there are  $2^n$  possible ways to choose stations.

Infeasible when  $n$  is large.  
Generalization: An optimal solution to the problem "find the fastest way through  $S_{1,n}$ " contains within it an optimal solution to subproblems:  
"find the fastest way through  $S_{1,j-1}$  or  $S_{2,j-1}$ "  
This is referred to as the optimal substructure property.

Definitions: [Recursive Solution]

$f^*$  - the fastest time to get through the entire factory.

$f_i[j]$  - the fastest time to get from the starting point through  $S_{i,j}$

$$f^* = \min (f_1[n] + x_1, f_2[n] + x_2)$$

Base case:  $j=1, i=1, 2$

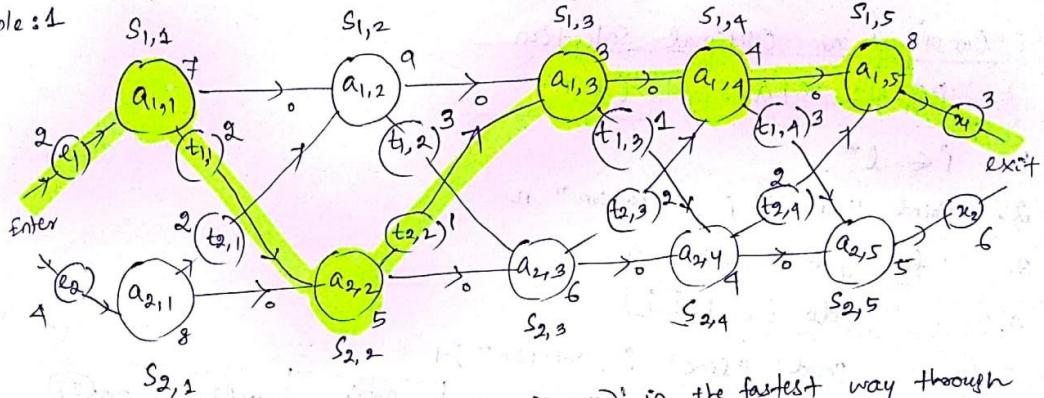
$$f_1[1] = e_1 + a_{1,1} \quad f_2[1] = e_2 + a_{2,1}$$

General case:  $j = 2, 3, \dots, n, i = 1, 2$

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j=1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j=1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

Example: 1



is the line no. whose last station is used in the fastest way through

\* is the line no. where last station is used in the fastest way through the entire factory.

$L_i[j]$  helps us to track in increasing order of  $j$ .

| $f_1[j]$ | 1  | 2      | 3      | 4      | 5      | $f^* = 35 [1]$               |
|----------|----|--------|--------|--------|--------|------------------------------|
| $f_2[j]$ | 9  | 18, 23 | 21, 20 | 21, 28 | 32, 35 | $+3 (\text{exit cost}) = 35$ |
| $f_1[j]$ | 12 | 17, 16 | 22, 27 | 25     | 30, 32 | $+6 (\text{exit cost}) = 36$ |
| $L_1[j]$ | 1  | 2      | 1      | 1      | 2      |                              |
| $L_2[j]$ | 1  | 2      | 1      | 2      |        |                              |

Algorithm:

FASTEST-WAY ( $a, t, e, x, n$ )  $\rightarrow$  complexity  $O(N)$

1.  $f_1[1] \leftarrow e_1 + a_{1,1}$  } compute initial values of  $f_1$  and  $f_2$

2.  $f_2[4] \leftarrow e_2 + a_{2,1}$

3. for  $j \leftarrow 2$  to  $n$

4. do if  $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$

5. then  $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$

6.  $L_1[j] \leftarrow 1$

7. else  $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$

8.  $L_1[j] \leftarrow 2$

} Computer the values of  $f_1[j]$  and  $L_1[j]$

9. if  $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$  } Compute the  
 10. then  $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$  values of  
 11.  $t_2[j] \leftarrow 2$   
 12. else  $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$   $f_2[j]$  and  
 13.  $t_2[j] \leftarrow 1$   $t_2[j]$   
 14. if  $f_1[n] + x_1 \leq f_2[n] + x_2$  } Compute the values of  
 15. then  $f^* = f_1[n] + x_1$  the fastest time through  
 16.  $t^* = 1$  the entire factory.  
 17. else  $f^* = f_2[n] + x_2$   
 18.  $t^* = 2$

### Construct an Optimal Solution

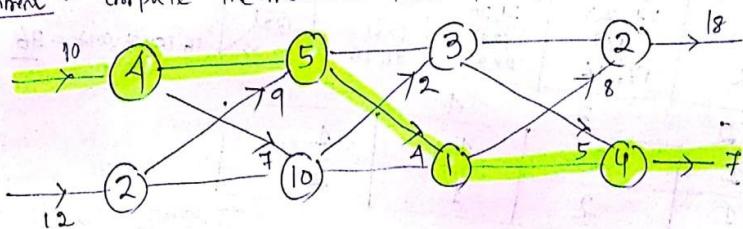
#### PRINT-STATIONS ( $l, n$ )

1.  $i \leftarrow l^*$
2. Point "line"  $i$ , "station"  $n$
3. for  $j \leftarrow n$  down to 2
4. do  $i \leftarrow l_1[i]$

5. Point "line"  $i$ , "station"  $j-1$

| $f_1[i]/t_1[i]$ | 9  | 18 | 20 | 24 | 32 |
|-----------------|----|----|----|----|----|
| $f_2[i]/t_2[i]$ | 12 | 16 | 22 | 25 | 30 |
|                 | 1  | 2  | 3  | 4  | 5  |
|                 | +3 | +3 | +3 | +3 | +6 |

Assignment - compute the minimum time it will take to build a car chassis.



↓ Dry Run.

initially  $l^* = 1$ ,  $i = 1$ , Point line 1 station 5.  
 then  $j = 5$  down to 2,  $i \rightarrow l_1[5]$ , Point line 1 station 4  
 then  $j = 4$ ,  $i \rightarrow l_1[4]$ , Line 1 station 3  
 $j = 3$ ,  $i \rightarrow l_1[3]$ , Line 2 station 2  
 $j = 2$ ,  $i \rightarrow l_2[2]$ , Line 1 station 1

Note: we can compute the fastest way through the factory and the time it takes  $O(n)$  times

## Greedy Algorithms

It always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

- It has only one shot to compute the optimal solution, so that it never goes back and reverses the decision.

- The greedy method is applied to solve the optimization problems that involve searching through a set of configurations to find one that minimize or maximize an objective function defined on these configurations.

### Elements of Greedy strategy

- 1) Determination of the optimal substructure of the problem.
- 2) Development of a recursive solution.
- 3) prove that any stage of the recursion, one of the optimal choice is the greedy choice.
- 4) show that all but one of the sub-problems included by having made the greedy choice are empty.
- 5) Develop a recursive algorithm that implements the greedy strategy.
- 6) convert the recursive algorithm to an iterative algorithm.

### Divide & Conquer

1. Single problem can be divided into 'n' sub problems, these n sub problems can be further divided into 'n' further sub parts.
2. After dividing solve the problem when further division is not possible.

### 3. Examples:

- Quick Sort, Merge sort
- Heap Sort
- Priority Queue

1. Solving starts from bottom.

### Algorithm Greedy (a, n)

```

    { for i=1 to n do
      {
        x = select(a);
        if feasible(x) then
          Solution = Solution+x;
      }
    }
  
```

### Greedy

1. It is based on local optimum solution but it can not give guarantee about the global optimum always.
2. for applying this technique the entities should be arranged in some order.
3. Examples:
  - Activity Selection Problem
  - Fractional Knapsack
  - Huffman code.
  - Job Scheduling with deadline

$n=5$

|   |                |                |                |                |                |
|---|----------------|----------------|----------------|----------------|----------------|
| a | a <sub>1</sub> | a <sub>2</sub> | a <sub>3</sub> | a <sub>4</sub> | a <sub>5</sub> |
| 1 | 2              | 3              | 4              | 5              |                |

### Example: Purchasing Best car

- Select Board
  - Select Top models.
  - Latest & well tested models.
  - Features.
  - Company looking for a suitable candidate
  - Written - GD - Technical - PI
- Known method of selection called greedy.

### Dynamic Programming

1. In this approach there were exist multiple optimal solution.
2. out of multiple solution we choose one optimal solution dynamically.

### 3 Examples:

- Matrix Chain multiplication
- Longest common subsequence
- Assembly line scheduling
- Travelling salesman problem.

4. Solution starts from Top.

## Activity Selection Problem

It is the problem of scheduling several activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Here no conflict occurs.

$$\text{Set of activity} = \{a_1, a_2, a_3, \dots, a_n\}$$

$$\text{Start time} = \{s_1, s_2, s_3, \dots, s_n\}$$

$$\text{Finish time} = \{f_1, f_2, f_3, \dots, f_n\}$$

- Steps:
- 1) Sort the activities according to the finish time i.e.  $f_1 < f_2 < f_3 < \dots < f_n$
  - 2) Plot the graph according to the sorted sequence.
  - 3) Apply the algorithm and find the result.

### Greedy - Activity - Selector ( $A, S, f$ )

It is assumed that the  $n$  activities are ordered by monotonically increasing finish time.  $A$  is an array containing all activities.

1.  $n \leftarrow \text{length}[S] \text{ or } n = S.\text{length}$
2.  $SA \leftarrow \{a_1\}$
3.  $j \leftarrow 1$
4. for  $i \leftarrow 2$  to  $n$
5.     if  $s_i \geq f_j$
6.         then  $SA \leftarrow SA \cup \{a_i\}$
7.          $j \leftarrow i$
8. return  $SA$

### Analysis

→ If the activities are already sorted, so there are one  $O(1)$  operations required per activity. Thus the total running time required for Greedy-activity selector algorithm is  $O(n)$ .

→ If the activities are not sorted, then we have to sort it first by using  $O(n \log n)$  time.

$$\begin{aligned} \rightarrow \text{Total running time} &= O(n \log n) + O(n) \\ &\equiv O(n \log n) \end{aligned}$$

Note: Time complexity for Greedy-activity selector algorithm (un-sorted)  $\rightarrow O(n \log n)$

Let us consider the following set  $S$ .

| $A$   | $a_i$ | 1 | 2 | 3 | 4 | 5 | 6  | 7  | 8  | 9  | 10 | 11 |
|-------|-------|---|---|---|---|---|----|----|----|----|----|----|
| $s_i$ | 1     | 3 | 0 | 5 | 3 | 5 | 6  | 8  | 8  | 2  | 12 | 12 |
| $f_i$ | 4     | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 14 |

We can have several subset of mutually compatible activities like

$\{a_3, a_7, a_{11}\}$ ,  $\{a_1, a_4, a_8, a_{11}\}$ ,  $\{a_2, a_4, a_9, a_{11}\}$  and so on.

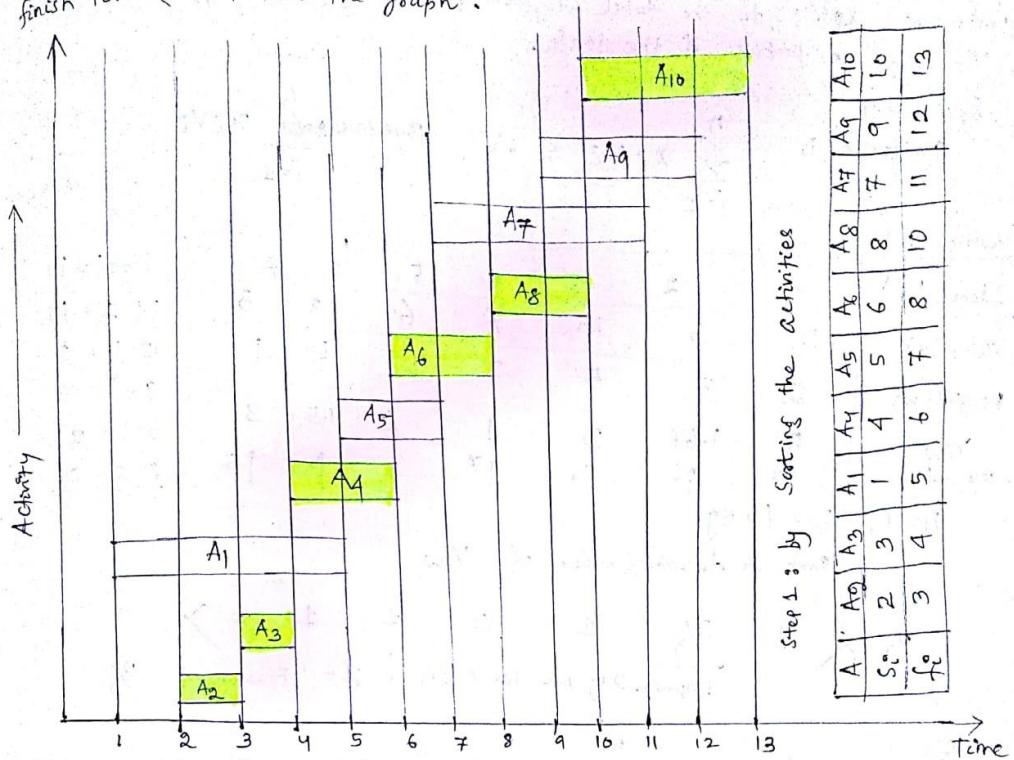
But we consider only those subsets which have maximum no. of activities and in this case it is 4.

Example: Given 10 activities along with their start and finish time as follows:

| $A$   | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ | $A_7$ | $A_8$ | $A_9$ | $A_{10}$ |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|
| $s_i$ | 1     | 2     | 3     | 4     | 5     | 6     | 7     | 8     | 9     | 10       |
| $f_i$ | 5     | 3     | 4     | 6     | 7     | 8     | 11    | 10    | 12    | 13       |

Find the optimal set of activities for the above activity-selection problem using Greedy strategy.

Solution: At first we have to sort the activities in increasing order of their finish time & then draw the graph.



Final Activity Schedule : SA = {A<sub>2</sub>, A<sub>3</sub>, A<sub>4</sub>, A<sub>6</sub>, A<sub>8</sub>, A<sub>10</sub>}

Fractional - Knapsack Problem :

In Fractional Knapsack problem we can even put the fraction of any item in the knapsack if taking the complete item is not possible i.e. here items are divisible.

It is solved using greedy approach.

0-1 Knapsack problem, division not possible & solved by dynamic programming.

Problem: Given a knapsack (shoulder bag) with a limited weight capacity. And few items each having some weight & value, which items should we place in the knapsack so that the value or profit that we obtain by putting the items in knapsack is maximum & the weight limit of the knapsack also does not exceed.

Steps for solving Fractional - Knapsack problem using greedy approach

- 1) For each item, compute its value/weight ratio
  - 2) Arrange all the items in the decreasing order of their value/weight ratios.
  - 3) Start putting the items in the knapsack beginning from the item with the highest ratio. Put as many items as we can in the knapsack.
- n items in a store, for  $i=1, 2, \dots, n$ , item 'i' has weight  $w_i > 0$  & value  $v_i > 0$ ,  $W$  = maximum capacity of the bag or knapsack

only a fraction  $x_i$  of object  $i$  is taken, where  $0 \leq x_i \leq 1$ . Item  $i$  contributes  $w_i x_i$  to the total weight in the knapsack, and  $x_i v_i$  to the value or profit of the load.

Symbolically -

Constraint is  $\sum_{i=1}^n x_i w_i \leq W$  objective  $\max \sum_{i=1}^n x_i v_i$

Example - 1 :-

| Items (i)  | 1            | 2            | 3            | 4            | 5            | 6            | 7            | $15 - 1 = 14$ |
|------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|---------------|
| Value (v)  | 10           | 5            | 15           | 7            | 6            | 18           | 3            | $14 - 2 = 12$ |
| Weight (w) | 2            | 3            | 5            | 7            | 1            | 4            | 1            | $12 - 4 = 8$  |
| $v/w$      | 5            | 1.67         | 3            | 1            | 6            | 4.5          | 3            | $8 - 5 = 3$   |
| Per unit   | $\uparrow_2$ | $\uparrow_6$ | $\uparrow_1$ | $\uparrow_7$ | $\uparrow_1$ | $\uparrow_3$ | $\uparrow_5$ | $3 - 1 = 2$   |
|            |              |              |              |              |              |              |              | $2 - 2 = 0$   |

$$n=7, W=15 \text{ Kg.}$$

Sort in decreasing order of  $v/w$

$$x < 1 \frac{2}{3} 1 0 1 1 1 >$$

Required 2 kg bag having 3 kg weight, Fraction is  $\frac{2}{3}$

$$\sum_{i=1}^n x_i w_i = 1 \times 2 + \frac{2}{3} \times 3 + 1 \times 5 + 0 \times 7 + 1 \times 1 + 1 \times 4 + 1 \times 1 = 15 \text{ Kg}$$

$$\begin{aligned} \text{Profit} \max \sum_{i=1}^n x_i v_i &= 1 \times 10 + \frac{2}{3} \times 5 + 1 \times 15 + 0 \times 7 + 1 \times 6 + 1 \times 18 + 1 \times 3 \\ &= 10 + 3.33 + 15 + 6 + 18 + 3 = \underline{\underline{55.33}} \quad (\text{Ans}) \end{aligned}$$

### Fractional-Knapsack ( $v, w, W$ )

Repeat for  $i \leftarrow 1$  to size( $v$ )

$$\text{do } p[i] \leftarrow v[i]/w[i]$$

[End of for loop]

Sort in decreasing order ( $p$ )

Set  $i \leftarrow 1$

Repeat while ( $W > 0$ )

$$\text{do } s \leftarrow \min(W, w[i])$$

$$\text{solution}[i] \leftarrow s$$

$$W \leftarrow W - s$$

$$i \leftarrow i + 1$$

[End of while loop].

Return (solution)

### Maximum-value (solution, p)

value  $\leftarrow 0$

Repeat for  $i \leftarrow 1$  to size of (solution)

do

$$\text{value} \leftarrow \text{value} + \text{solution}[i] * p[i]$$

[End of for loop]

return (value)

Here inputs =  $i_1, i_2, i_3 \dots$

weights =  $w_1, w_2, w_3 \dots$

value =  $v_1, v_2, v_3 \dots$

$W$  = maximum capacity

Analysis : (Time complexity)

Sort the value  $p = \frac{V_i}{W_i}$  in decreasing order will take  $O(n \log n)$  [using Merge Sort]  
while loop in algorithm require  $O(n)$  time.

$$\text{Total Time complexity} = O(n) + O(n \log n) = O(n \log n)$$

Example-2

| Items ( $I_i$ )       | 1   | 2   | 3   | 4   | 5   |
|-----------------------|-----|-----|-----|-----|-----|
| value ( $V_i$ )       | 30  | 20  | 100 | 90  | 160 |
| weights ( $W_i$ )     | 5   | 10  | 20  | 30  | 40  |
| $p = \frac{V_i}{W_i}$ | 6.0 | 2.0 | 5.0 | 3.0 | 4.0 |

(Sort)

Day Run according to algorithm :  
initial value of  $i=1$ ,

$$\text{Capacity of knapsack } W = 60 > 0$$

$$s = \min(W, w[1]) \Rightarrow \min(60, 5) \Rightarrow 5$$

$$\text{solution}[1] = 5$$

$$W = W - s \Rightarrow 60 - 5 = 55$$

$$i = i+1 = 1+1 = 2$$

Now capacity of knapsack  $W = 55 > 0$

$$s = \min(\dots, 55, 20) \Rightarrow 20, \text{ solution}[2] = 20$$

$$W = 55 - 20 = 35$$

$$i = i+1 = 2+1 = 3$$

Now capacity of knapsack  $W = 35 > 0$

$$s = \min(35, 40) \Rightarrow 35, \text{ solution}[3] = 35$$

$$W = W - s \Rightarrow 35 - 35 = 0$$

$$i = i+1 = 3+1 = 4$$

Now capacity of  $W = 0$ , which terminate the while loop & return solution

Now find the maximum value

initial value = 0

$i = 1$  to size of (solution)

$i = 1 \text{ to } 3$

$$i=1, \text{ value} = \text{value} + \text{solution}[1] * p[1] \Rightarrow 0 + 5 * 6 \Rightarrow 30$$

$$i=2, \text{ value} = \text{value} + \text{solution}[2] * p[2] \Rightarrow 30 + 20 * 5 \Rightarrow 130$$

$$i=3, \text{ value} = \text{value} + \text{solution}[3] * p[3] \Rightarrow 130 + 35 * 4 \Rightarrow 130 + 140 = 270$$

$\therefore$  Thus optimal solution of fractional-Knapsack problem is 270.

Assignment :

| Item   | 1  | 2  | 3  | 4  | 5  |
|--------|----|----|----|----|----|
| Value  | 30 | 40 | 45 | 77 | 90 |
| Weight | 5  | 10 | 15 | 22 | 25 |

$$n = 5$$

$$W = 60 \text{ kg}$$

$$\text{max profit} = 270 \text{ (Ans)}$$

## Huffman Codes : (Dr. David A. Huffman)

Huffman codes are widely used and very effective technique for compressing data. Here we consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string. Used for Lossless (without loss of information) compression.

Message: B C C A B B D D A E C C B B A E D D C C - length = 20.

Cost of message measured by bits.

Characters are represented in ASCII code. i.e 8 bits

$$A = 65 = \underbrace{01000001}_{8\text{ bits}}$$

$$\text{Size of msg} = \frac{20 \times 8 \text{ bits}}{\text{(length)}} = 160 \text{ bits}$$

Sample message sent without encoding require 160 bits.

(1) Fixed length/size code: Each character is represented by a unique binary string i.e equal no. of bits.

If in a file Characters consists of  $m$  different type character then maximum no. of bits used to represent each character is  $n$ , where  $2^n \geq m$

Here 6 different characters are present; at least 3 bits required i.e  $2^3 \geq 6$

| Character | Count/frequency | Code |  |
|-----------|-----------------|------|--|
| A         | 3               | 000  | $\text{Size} = 20 \times 3 \text{ bits} = 60 \text{ bits}$ |
| B         | 5               | 001  |  |
| C         | 6               | 010  | Encode msg + Table set transferred.                        |
| D         | 4               | 011  | $60 \text{ bits} + 55 \text{ bits}$                        |
| E         | 2               | 100  | $\Rightarrow 115 \text{ bits}$                             |

Table size =  $\frac{5 \times 8 \text{ bits}}{\text{ }} + \frac{5 \times 8 \text{ bits}}{\text{ }}$  Size is reduced.

$= 40 \text{ bits} + 15 \text{ bits}$

$= 55 \text{ bits}$

(2) Variable length/size code: Considerably better than a fixed-length code; by giving frequent characters short codewords and infrequent character as long codewords.

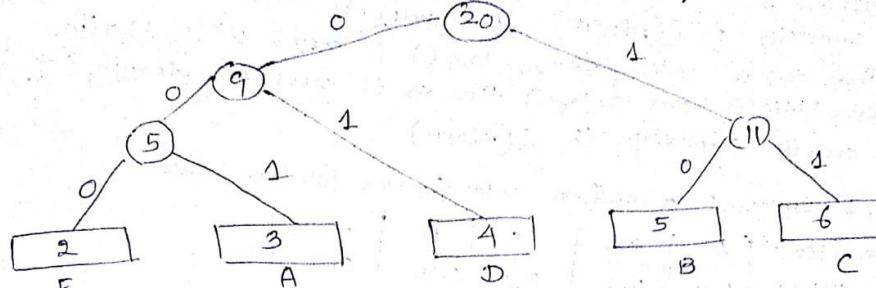
Prefix code is the code in which there is no codeword that is a prefix of other codeword. Decoding process is binary tree whose leaves are characters. We interpret the binary codeword for a character as path from the root to that character, where

$\rightarrow "0"$  means "go to the left child"

$\rightarrow "1"$  means "go to the right child"

It is also called as Optimal merge pattern (Arrange increase order of the count)

take two smallest one to make one node and sum the frequencies.



| Char | Count | Code | Size  |
|------|-------|------|---|
| A    | 3     | 001  | $3 \times 3 = 9$ bits                           |
| B    | 5     | 10   | $5 \times 2 = 10$ bits                          |
| C    | 6     | 11   | $6 \times 2 = 12$ bits                          |
| D    | 4     | 01   | $4 \times 2 = 8$ bits                           |
| E    | 2     | 000  | $2 \times 3 = 6$ bits<br>15 bits (message size) |

$$\text{Table Size} \propto \text{Tree} = 5 \times 8 \text{ bits} + 12 \text{ bits} \Rightarrow 40 \text{ bits} + 12 = 52 \text{ bits}$$

$$\text{Total} = \text{Message} + \text{Table/Tree} = 15 \text{ bits} + 52 \text{ bits} = 67 \text{ bits} \quad (\text{More reduced})$$

formula  $\Rightarrow$  distance \* frequency is Edific (optimal message pattern):

Distance of E = 3, Bcs = 2 similarly for all other character, we got

$$3 \times 2 + 3 \times 3 + 2 \times 4 + 2 \times 5 + 2 \times 6 = 45 \text{ bits}$$

B C C D A C C B . . . . .

10 11 11 01 00 11 11 10 . . . .

Decode : B C C D . . . .

Greedy Algorithm for Huffman code :  
Huffman invented a greedy algorithm that constructs an optimal prefix-code called Huffman code. It begins with a set of |C| leaves and performs a sequence of  $|C|-1$  merging operations to create a final node. To create the Huffman code to a file we have used "min-priority queue" data structure.

We assume that C is a set of n characters and the each character  $c \in C$  is an object with an  $c.freq$  attribute.

Huffman(C)

```

n → |C|
Q → C // min-priority queue
repeat for i → 1 to n-1
    do Z → Allocate-node() // allocate a new node Z
    left[Z] ← x; x ← Extract-MIN(Q)
    right[Z] ← y; y ← Extract-MIN(Q)
    f[Z] = f[x] + f[y]
    Insert(Q, Z)

```

End of loop

return Extract-MIN(Q) // return the root of the tree

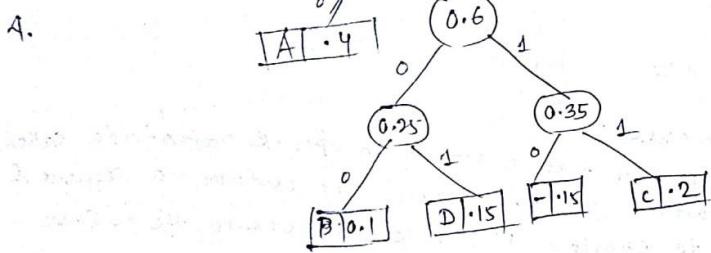
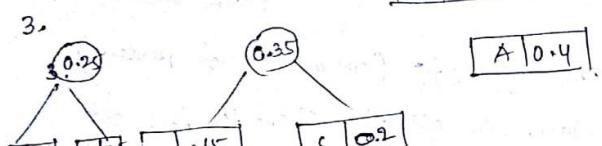
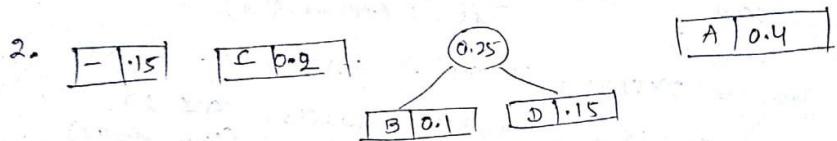
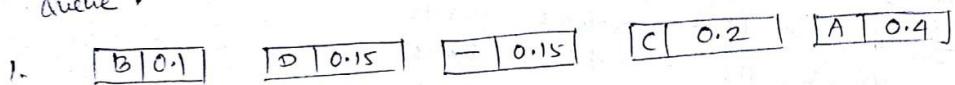
### Analysis

Time complexity is  $O(n \log n)$ ,  $n = \text{number of unique characters}$ .  
 If there are  $n$  nodes,  $\text{extractMin}()$  is called  $2*(n-1)$  times.  
 $\text{ExtractMin}()$  takes  $O(\log n)$  time as it calls  $\text{Min-Heapify}()$ .  
 So overall complexity is  $O(n \log n)$ .

Example: 1 Construct a Huffman code for the following data.

| Character:   | A   | B   | C   | D    | -    |
|--------------|-----|-----|-----|------|------|
| Probability: | 0.4 | 0.1 | 0.2 | 0.15 | 0.15 |

Solution: initially the character & probability are inserted in the priority queue.



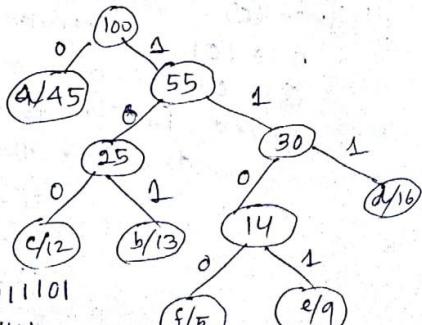
Assign code. Left = 0, Right = 1

The huffman code for  $A = 0, B = 100, C = 111, D = 101, - = 110$

| Character | frequency | Variable length code |   |
|-----------|-----------|----------------------|---|
|           |           | 0                    | 1 |
| a         | 45        | 0                    |   |
| b         | 13        | 101                  |   |
| c         | 12        | 100                  |   |
| d         | 16        | 111                  |   |
| e         | 9         | 1101                 |   |
| f         | 5         | 1100                 |   |

Ans:

String is 001011101  
 as 0.0.101.1101  
 decodes a|a|b|e (parsing uniquely)



## UNIT: 3

### Disjoint Set Data Structure

In computing, a **disjoint-set data structure**, also called a **union–find data structure** or **merge–find set**, is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non overlapping) subsets.

It supports the following useful operations:

- **Find:** Determine which subset a particular element is in. *Find* typically returns an item from this set that serves as its "representative"; by comparing the result of two *Find* operations, one can determine whether two elements are in the same subset.
- **Union:** Join two subsets into a single subset.
- **Make Set,** which makes a set containing only a given element (a singleton). **Applications:**
- partitioning of a set
- implementing Kruskal's algorithm to find the minimum spanning tree of a graph.
- Determine the connected components of an undirected graph. **Disjoint Set Operations, Linked list Representation**
- A disjoint-set is a collection  $S=\{S_1, S_2, \dots, S_k\}$  of distinct dynamic sets.
- Each set is identified by a member of the set, called *representative*. **Disjoint set operations**
  - **MAKE-SET( $x$ ):** create a new set with only  $x$ . Assume  $x$  is not already in some other set.
  - **UNION( $x,y$ ):** combine the two sets containing  $x$  and  $y$  into one new set. A new representative is selected.
  - **FIND-SET( $x$ ):** return the representative of the set containing  $x$ .

Application: Determine the connected components of an undirected graph.

### Linked list Representation

- Each set as a linked-list, with head and tail, and each node contains value, next node pointer and back-to-representative pointer.

## 21.2 Linked-list representation of disjoint sets

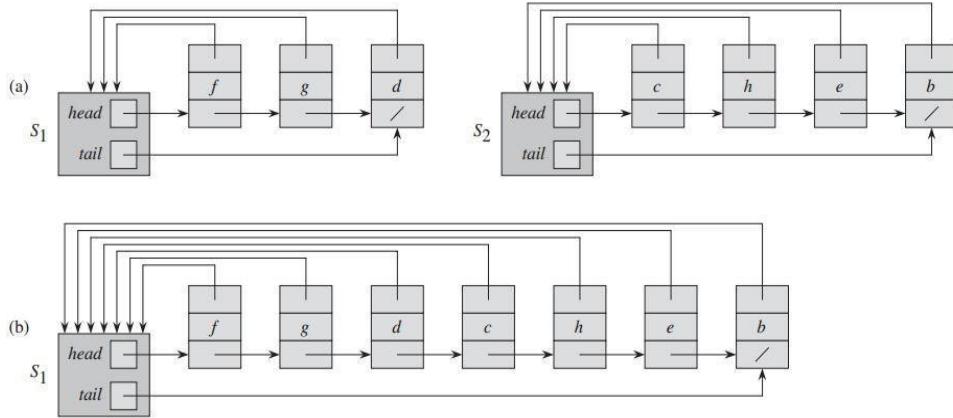
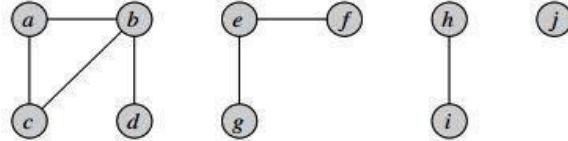


Figure 21.2 (a) Linked-list representations of two sets. Set  $S_1$  contains members  $d$ ,  $f$ , and  $g$ , with representative  $f$ , and set  $S_2$  contains members  $b$ ,  $c$ ,  $e$ , and  $h$ , with representative  $c$ . Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers  $head$  and  $tail$  to the first and last objects, respectively. (b) The result of  $\text{UNION}(g, e)$ , which appends the linked list containing  $e$  to the linked list containing  $g$ . The representative of the resulting set is  $f$ . The set object for  $e$ 's list,  $S_2$ , is destroyed.

The total time spent in all UNION operations is thus  $O(n \lg n)$ . Each MAKESET and FIND-SET operation takes  $O(1)$  time, and there are  $O(m)$  of them. The total time for the entire sequence is thus  $O(m + n \lg n)$ .

## 21.1 Disjoint-set operations



(a)

| Edge processed | Collection of disjoint sets |       |     |     |         |     |     |       |     |     |
|----------------|-----------------------------|-------|-----|-----|---------|-----|-----|-------|-----|-----|
|                | {a}                         | {b}   | {c} | {d} | {e}     | {f} | {g} | {h}   | {i} | {j} |
| initial sets   | {a}                         |       |     |     |         |     |     |       |     |     |
| (b,d)          | {a}                         | {b,d} | {c} |     | {e}     | {f} | {g} | {h}   | {i} | {j} |
| (e,g)          | {a}                         | {b,d} | {c} |     | {e,g}   | {f} |     | {h}   | {i} | {j} |
| (a,c)          | {a,c}                       | {b,d} |     |     | {e,g}   | {f} |     | {h}   | {i} | {j} |
| (h,i)          | {a,c}                       | {b,d} |     |     | {e,g}   | {f} |     | {h,i} |     | {j} |
| (a,b)          | {a,b,c,d}                   |       |     |     | {e,g}   | {f} |     | {h,i} |     | {j} |
| (e,f)          | {a,b,c,d}                   |       |     |     | {e,f,g} |     |     | {h,i} |     | {j} |
| (b,c)          | {a,b,c,d}                   |       |     |     | {e,f,g} |     |     | {h,i} |     | {j} |

(b)

**Figure 21.1** (a) A graph with four connected components: {a, b, c, d}, {e, f, g}, {h, i}, and {j}.  
(b) The collection of disjoint sets after processing each edge.

CONNECTED-COMPONENTS(G)

1. for each vertex  $v \in V[G]$
2. MAKE-SET( $v$ )
3. for each edge  $(u,v) \in E[G]$
4. if  $\text{FIND-SET}(u) \neq \text{FIND-SET}(v)$
5. then  $\text{UNION}(u,v)$  SAME-COMPONENT( $u,v$ )
  1. if  $\text{FIND-SET}(u) = \text{FIND-SET}(v)$
  2. then return TRUE
  3. else return FALSE

The procedure CONNECTED-COMPONENTS initially places each vertex in its own set. Then, for each edge  $(u,v)$ , it unites the sets containing  $u$  and  $v$ . By Exercise 21.1-2, after processing all the edges, two vertices are in the same connected component if and only if the corresponding objects are in the same set. Thus,

CONNECTEDCOMPONENTS computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component. Figure 21.1(b) illustrates how CONNECTEDCOMPONENTS compute the disjoint sets.

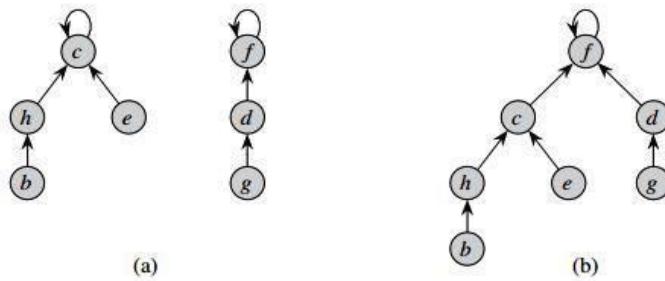
**Disjoint Forests**

In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set. In a disjointset forest, illustrated in Figure 21.4(a), each member points only to its parent. The root of each tree contains the representative and is its own parent.

As we shall see, although the straightforward algorithms that use this representation are no faster than ones that use the linked-list representation, by introducing two heuristics—union by rank $\parallel$  and —path compression $\parallel$ —we can achieve an asymptotically optimal disjoint-set data structure.

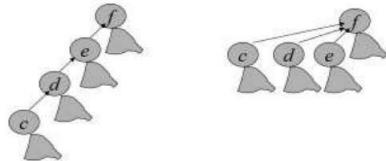
### Union by Rank & Path Compression

- **Union by Rank:** Each node is associated with a rank, which is the upper bound on the height of the node (i.e., the height of subtree rooted at the node), then when UNION, let the root with smaller rank point to the root with larger rank.



**Figure 21.4** A disjoint-set forest. (a) Two trees representing the two sets of Figure 21.2. The tree on the left represents the set  $\{b, c, e, h\}$ , with *c* as the representative, and the tree on the right represents the set  $\{d, f, g\}$ , with *f* as the representative. (b) The result of  $\text{UNION}(e, g)$ .

- **Path Compression:** used in FIND-SET(*x*) operation, make each node in the path from *x* to the root directly point to the root. Thus reduce the tree height.



### Pseudocode for disjoint-set forests

```

MAKE-SET( $x$ )
1  $x.p = x$ 
2  $x.rank = 0$ 

UNION( $x, y$ )
1 LINK(FIND-SET( $x$ ), FIND-SET( $y$ ))

LINK( $x, y$ )
1 if  $x.rank > y.rank$ 
2      $y.p = x$ 
3 else  $x.p = y$ 
4     if  $x.rank == y.rank$ 
5          $y.rank = y.rank + 1$ 

```

The FIND-SET procedure with path compression is quite simple:

```

FIND-SET( $x$ )
1 if  $x \neq x.p$ 
2      $x.p = \text{FIND-SET}(x.p)$ 
3 return  $x.p$ 

```

To implement a disjoint-set forest with the union-by-rank heuristic, we must keep track of ranks. With each node  $x$ , we maintain the integer value  $x.rank$ , which is an upper bound on the height of  $x$ . When MAKE-SET creates a singleton set, the single node in the corresponding tree has an initial rank of 0. Each FIND-SET operation leaves all ranks unchanged. The UNION operation has two cases, depending on whether the roots of the trees have equal rank. If the roots have unequal rank, we make the root with higher rank the parent of the root with lower rank, but the ranks themselves remain unchanged. If, instead, the roots have equal ranks, we arbitrarily choose one of the roots as the parent and increment its rank.

Let us put this method into pseudocode. We designate the parent of node  $x$  by  $x.p$ . The LINK procedure, a subroutine called by UNION, takes pointers to two roots as inputs. The FIND-SET procedure is a twopass method: as it recurses, it makes one pass up the find path to find the root, and as the recursion unwinds, it makes a second pass back down the find path to update each node to point directly to the root. Each call of FIND-SET( $x$ ) returns  $x.p$  in line 3. If  $x$  is the root, then FIND-SET skips line 2 and instead returns  $x.p$ , which is  $x$ ; this is the case in which the recursion bottoms out. Otherwise, line 2 executes, and the recursive call with parameter  $x.p$  returns a pointer to the root. Line 2 updates node  $x$  to point directly to the root, and line 3 returns this pointer

### Graph algorithms

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.

Types of graphs

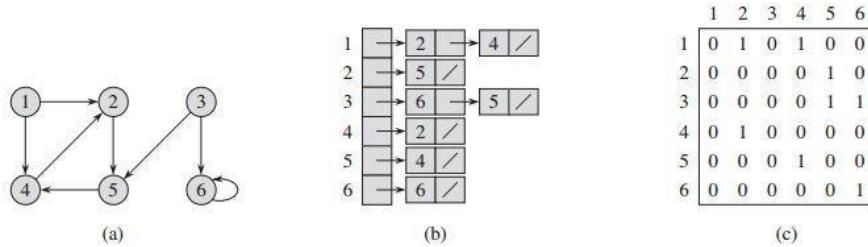
**Undirected:** An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.

**Directed:** A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.

**Weighted:** In a weighted graph, each edge is assigned a weight or cost.

**Cyclic:** A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.

#### Adjacency-list representation of a graph



**Figure 22.2** Two representations of a directed graph. (a) A directed graph  $G$  with 6 vertices and 8 edges. (b) An adjacency-list representation of  $G$ . (c) The adjacency-matrix representation of  $G$ .

#### Graph Algorithm - BFS and DFS

| BFS   | DFS  |
|---|--|
| <b>BFS</b> Stands for — <b>Breadth First Search</b> !.  | <b>DFS</b> stands for — <b>Depth First Search</b> !.   |
| BFS starts traversal from the root node and then explores the search in the level by level manner i.e. as close as possible from the root node. | DFS starts the traversal from the root node and explores the search as far as possible from the root node i.e. depth wise.   |
| Breadth First Search can be done with the help of <b>queue</b> i.e. <b>FIFO</b> implementation.   | Depth First Search can be done with the help of <b>Stack</b> i.e. <b>LIFO</b> implementations.   |
| This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once.                               | This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped-off. |
| BFS is <b>slower</b> than DFS.  | DFS is more <b>faster</b> than BFS.  |
| BFS requires <b>more</b> memory compare to DFS.   | DFS require <b>less</b> memory compare to BFS.   |
| <b>Applications of BFS</b> ><br>To find Shortest path   | <b>Applications of DFS</b><br>> useful in finding spanning trees & forest.   |

BFS( $G, s$ )

```

1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.\text{Adj}[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```



A, B, C, D, E, F

Example :



A, B, D, C, E, F

**Example :**

The procedure BFS works as follows. With the exception of the source vertex  $s$ , lines 1–4 paint every vertex white, set  $u.d$  to be infinity for each vertex  $u$ , and set the parent of every vertex to be NIL. Line 5 paints  $s$  gray,

since we consider it to be discovered as the procedure begins. Line 6 initializes  $s.d$  to 0, and line 7 sets the predecessor of the source to be NIL.

Lines 8–9 initialize  $Q$  to the queue containing just the vertex  $s$ . The while loop of lines 10–18 iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined.

Line 11 determines the gray vertex  $u$  at the head of the queue  $Q$  and removes it from  $Q$ . The for loop of lines 12–17 considers each vertex  $v$  in the adjacency list of  $u$ . If  $v$  is white, then it has not yet been discovered, and the procedure discovers it by executing lines 14–17. The procedure paints vertex gray, sets its distance  $v.d$  to  $u.d+1$ , records  $u$  as its parent  $v.\pi$ , and places it at the tail of the queue  $Q$ . Once the procedure has examined all the vertices on  $u$ 's adjacency list, it blackens  $u$  in line 18. The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted

black (in line 18).

### Analysis of BFS

The operations of enqueueing and dequeuing take  $O(1)$  time, and so the total time devoted to queue operations is  $O(V)$ . Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. The total time spent in scanning adjacency lists is  $O(E)$ . The total running time of the BFS procedure is  $O(V + E)$ .

The following procedure prints out the vertices on a shortest path from  $s$  to  $v$ , assuming that BFS has already computed a breadth-first tree:

```

PRINT-PATH( $G, s, v$ )
1  if  $v == s$ 
2      print  $s$ 
3  elseif  $v.\pi == \text{NIL}$ 
4      print "no path from"  $s$  "to"  $v$  "exists"
5  else PRINT-PATH( $G, s, v.\pi$ )
6      print  $v$ 
```

### Example

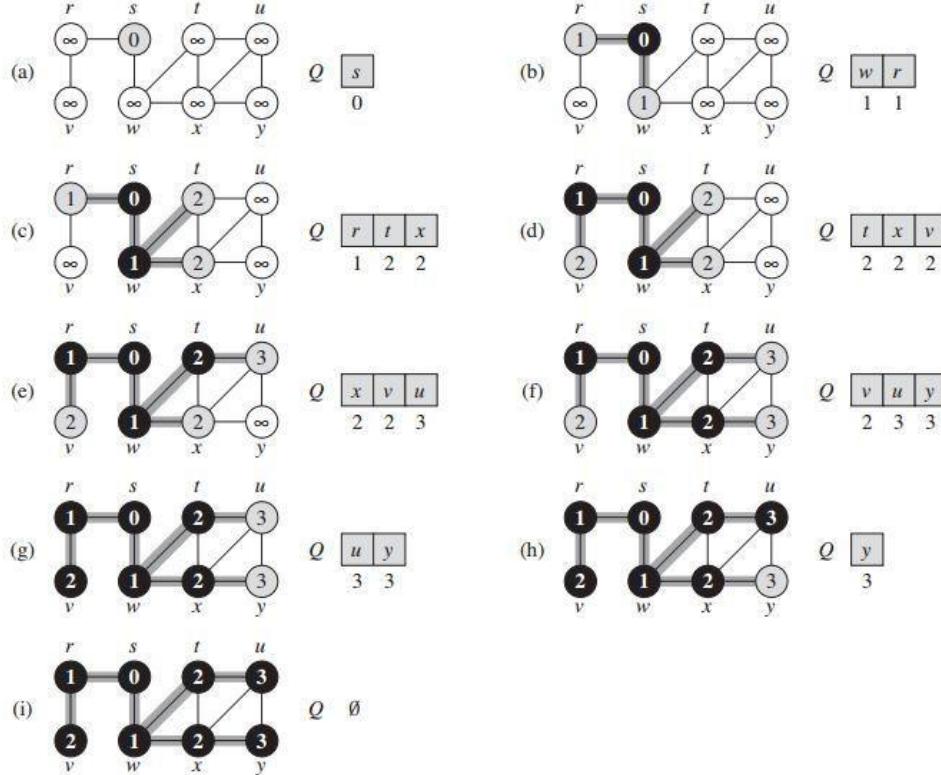


Figure: The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of  $u.d$  appears within each vertex  $u$ . The queue  $Q$  is shown at the beginning of each iteration of the while loop of lines 10–18. Vertex distances appear below vertices in the queue.

### Depth-first search

The strategy followed by depth-first search is, as its name implies, to search —deeper— in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex that still has unexplored edges leaving it. Once all of  $v$ 's edges have been explored, the search —backtracks— to explore edges leaving the vertex from which was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

The following pseudocode is the basic depth-first-search algorithm. The input graph  $G$  may be undirected or directed. The variable  $time$  is a global variable that we use for timestamping.

$\text{DFS}(G)$

```

1 for each vertex  $u \in G.V$ 
2    $u.\text{color} = \text{WHITE}$ 
3    $u.\pi = \text{NIL}$ 
4    $time = 0$ 
5 for each vertex  $u \in G.V$ 
6   if  $u.\text{color} == \text{WHITE}$ 
7      $\text{DFS-VISIT}(G, u)$ 

```

$\text{DFS-VISIT}(G, u)$

```

1  $time = time + 1$            // white vertex  $u$  has just been discovered
2  $u.d = time$ 
3  $u.\text{color} = \text{GRAY}$ 
4 for each  $v \in G.\text{Adj}[u]$     // explore edge  $(u, v)$ 
5   if  $v.\text{color} == \text{WHITE}$ 
6      $v.\pi = u$ 
7      $\text{DFS-VISIT}(G, v)$ 
8    $u.\text{color} = \text{BLACK}$         // blacken  $u$ ; it is finished
9    $time = time + 1$ 
10   $u.f = time$ 

```

Procedure DFS works as follows. Lines 1–3 paint all vertices white and initialize their  $\pi$  attributes to NIL. Line 4 resets the global time counter. Lines 5–7 check each vertex in  $V$  in turn and, when a white vertex is found, visit it using DFS-VISIT. Every time  $\text{DFS-VISIT}(G, u)$  is called in line 7, vertex  $u$  becomes the root of a new tree in the depth-first forest. When DFS returns, every vertex  $u$  has been assigned a discovery time  $u.d$  and a finishing time  $u.f$ . In each call  $\text{DFS-VISIT}(G, u)$ , vertex  $u$  is initially white. Line 1 increments the global variable  $time$ , line 2 records the new value of  $time$  as the discovery time  $u.d$ , and line 3 paints  $u$  gray. Lines 4–7 examine each vertex  $v$  adjacent to  $u$  and recursively visit it if it is white. As each vertex  $v \in \text{Adj}[u]$  is considered in line 4, we say that edge  $(u, v)$  is explored by the depth-first search. Finally, after every edge leaving  $u$  has been explored, lines 8–10 paint  $u$  black, increment  $time$ , and record the finishing time in  $u.f$ .

**Analysis of BFS:** The total running time of the BFS procedure is  $O(V + E)$ .

**Example:**

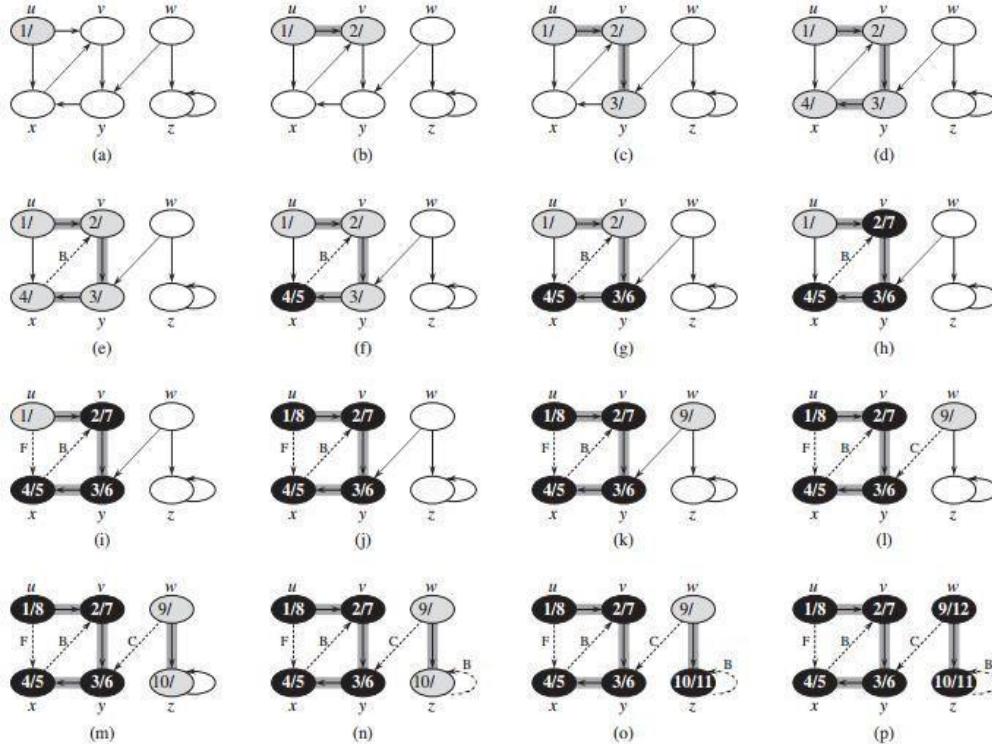


Figure: The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.

### Minimum Spanning Trees

Given an undirected and connected graph  $G = (V, E)$ , a spanning tree of the graph  $G$  is a tree that spans  $G$  (that is, it includes every vertex of  $G$ ) and is a subgraph of  $G$  (every edge in the tree belongs to  $G$ ).

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees. Minimum spanning tree has direct application in the design of networks.

We can model this wiring problem with a connected, undirected graph  $G = (V, E)$ , where  $V$  is the set of pins,  $E$  is the set of possible interconnections between pairs of pins, and for each edge  $(u, v) \in E$ , we have a weight  $w(u, v)$  specifying the cost (amount of wire needed) to connect  $u$  and  $v$ . We then wish to find an acyclic subset  $T \subseteq E$  that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is minimized. Since  $T$  is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it spans the graph  $G$ . We call the problem of determining the tree  $T$  the minimumspanning-tree problem.

Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of  $n$  pins, we can use an arrangement of  $n-1$  wires, each connecting two pins. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

In this chapter, we shall examine two algorithms for solving the minimumspanning-tree problem:

Kruskal's algorithm and Prim's algorithm.

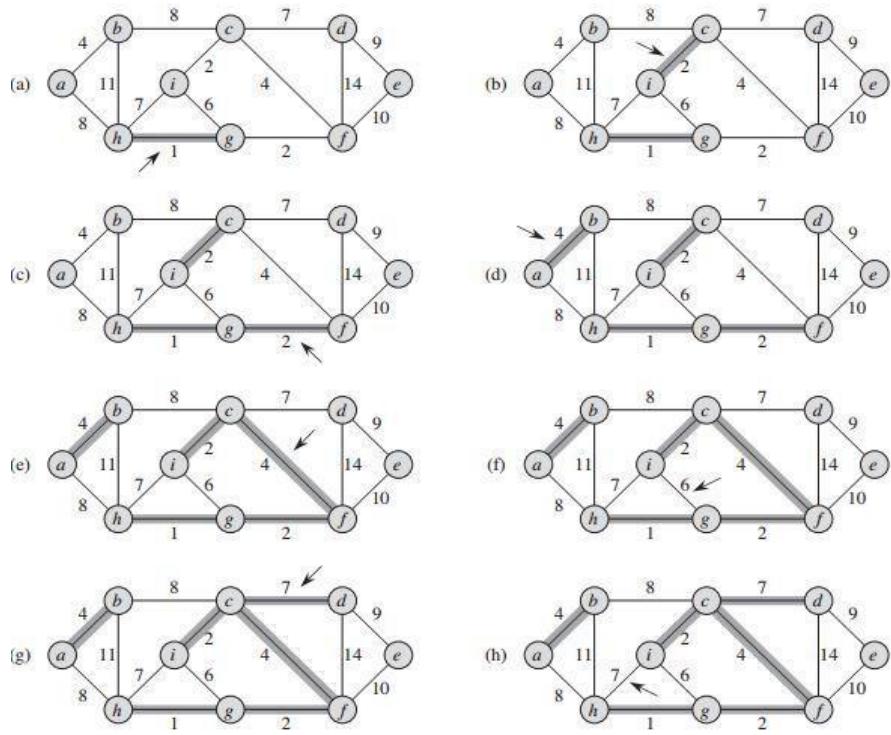
#### **Kruskal's Algorithm**

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and adds it to the growing spanning tree.

#### **Algorithm Steps:**

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle, edges which connect only disconnected components.

Example:



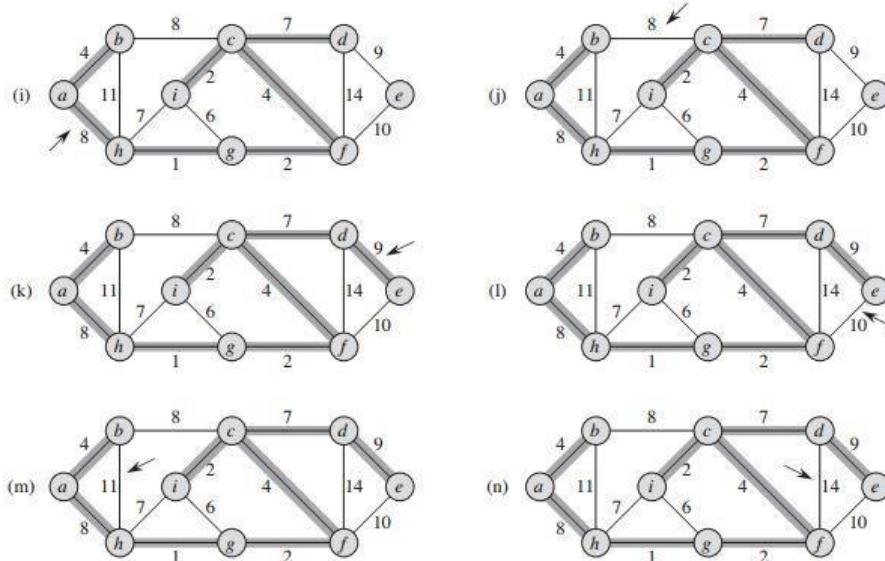


Figure: The execution of Kruskal's algorithm on the graph from Shaded edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Kruskal's uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest. The operation FIND-SET( $u$ ) returns a representative element from the set that contains  $u$ . Thus, we can determine whether two vertices  $u$  and  $v$  belong to the same tree by testing whether FIND-SET( $u$ ) equals FIND-SET( $v$ ). To combine trees, Kruskal's algorithm calls the UNION procedure.

### MST-KRUSKAL( $G, w$ )

- 1  $A = \emptyset$
- 2 **for** each vertex  $v \in G.V$
- 3     **MAKE-SET**( $v$ )
- 4     sort the edges of  $G.E$  into nondecreasing order by weight  $w$
- 5     **for** each edge  $(u, v) \in G.E$ , taken in nondecreasing order by weight
- 6         **if**  $\text{FIND-SET}(u) \neq \text{FIND-SET}(v)$
- 7              $A = A \cup \{(u, v)\}$
- 8             **UNION**( $u, v$ )
- 9 **return**  $A$

Lines 1–3 initialize the set A to the empty set and create  $|V|$  trees, one containing each vertex. The for loop in lines 5–8 examines edges in order of weight, from lowest to highest. The loop checks, for each edge(u, v), whether the endpoints u and v belong to the same tree. If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is discarded. Otherwise, the two vertices belong to different trees. In this case, line 7 adds the edge (u, v) to A, and line 8 merges the vertices in the two trees.

#### Time Complexity:

In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be  $O(E \lg V)$ , which is the overall Time Complexity of the algorithm.

#### Prim's Algorithm

Prim's Algorithm also uses Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm Steps:

- 1) Maintain two disjoint sets of vertices. One is containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- 2) Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices that are connected to growing spanning tree, into the Priority Queue.
- 3) Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

#### MST-PRIM( $G, w, r$ )

```

1  for each  $u \in G.V$ 
2     $u.key = \infty$ 
3     $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7     $u = \text{EXTRACT-MIN}(Q)$ 
8    for each  $v \in G.Adj[u]$ 
9      if  $v \in Q$  and  $w(u, v) < v.key$ 
10         $v.\pi = u$ 
11         $v.key = w(u, v)$ 
```

In the pseudocode, the connected graph  $G$  and the root  $r$  of the minimum spanning tree to be grown are inputs to the algorithm. During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue  $Q$  based on a key attribute. For each vertex  $v$ , the attribute  $v.key$  is the minimum weight of any edge connecting  $v$  to a vertex in the tree; by convention,  $v.key = \infty$  if there is no such edge. The attribute  $v.\pi$  names the parent of  $v$  in the tree.

so that it will be the first vertex processed), set the parent of each vertex to NIL, and initialize the minpriority queue Q to contain all the vertices. Line 7 identifies a vertex  $u \in Q$  incident on a light edge that crosses the cut  $(V - Q, Q)$  (with the exception of the first iteration, in which  $u = r$  due to line 4). Removing  $u$  from the set  $Q$  adds it to the set  $V - Q$  of vertices in the tree, thus adding  $(u, u.\pi)$  to  $A$ . The for loop of lines 8–11 updates the key and  $\pi$  attributes of every vertex  $v$  adjacent to  $u$  but not in the tree.

Lines 1–5 set the key of each vertex to  $\infty$  (except for the root  $r$ , whose key is set to 0

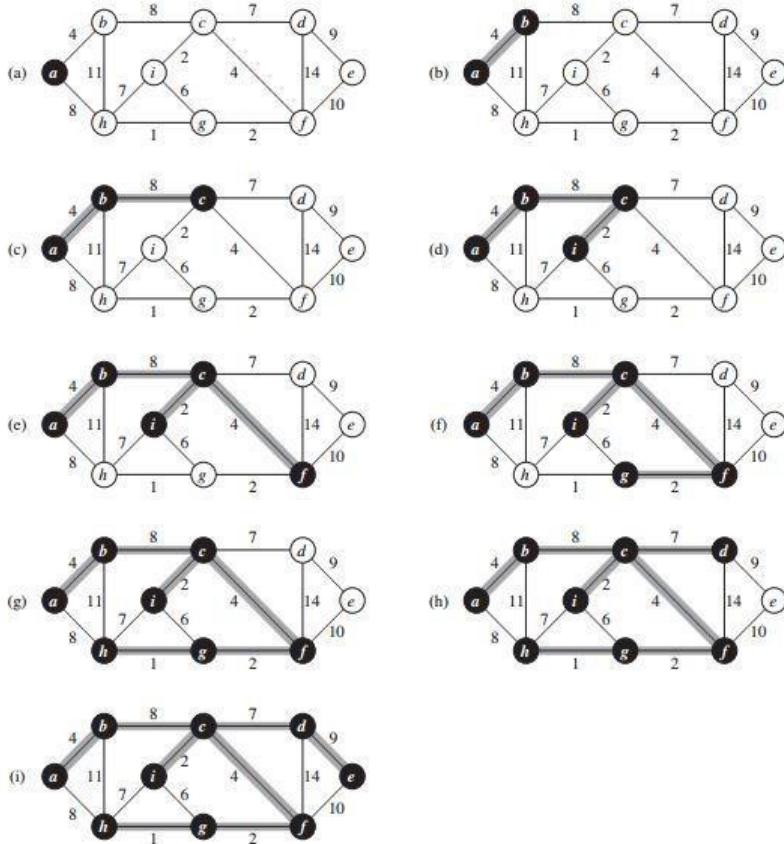


Figure: The execution of Prim's algorithm on the graph. The root vertex is  $a$ . Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge  $(b, c)$  or edge  $(a, h)$  to the tree since both are light edges crossing the cut.

#### Time Complexity:

A binary min-heap supports in  $O(\lg V)$  time.

Thus, the total time for Prim's algorithm is  $O(V \lg V + E \lg V) = O(E \lg V)$ , which is asymptotically the same as for our implementation of Kruskal's algorithm. If we use a Fibonacci heap to implement the min-priority queue  $Q$ , the running time of Prim's algorithm improves to  $O(E + V \lg V)$ .

#### Single-Source Shortest Paths:

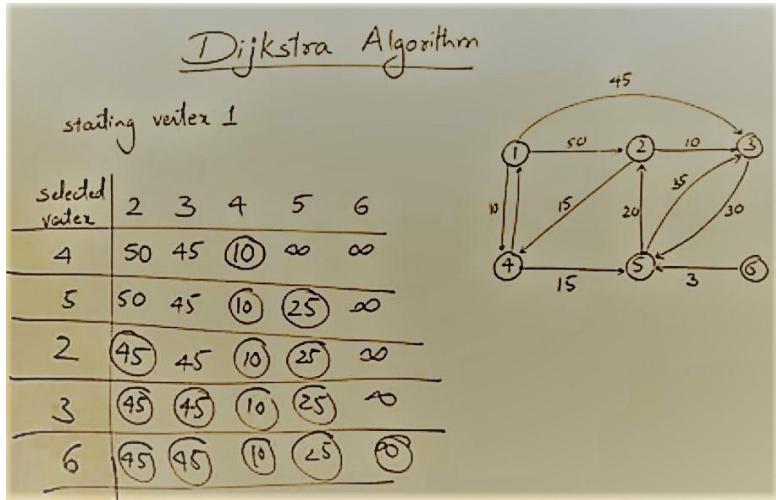
We define the **shortest-path weight**  $\delta(u, v)$  from  $u$  to  $v$  by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

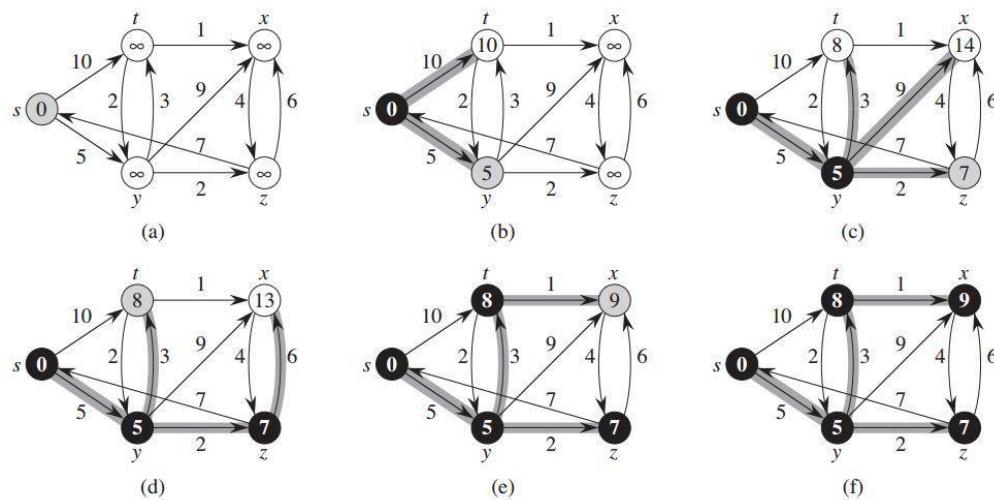
Dijkstra's algorithm maintains a set  $S$  of vertices whose final shortest-path weights from the source  $s$  have already been determined. The algorithm repeatedly selects the vertex  $u \in V - S$  with the minimum shortest-path estimate, adds  $u$  to  $S$ , and relaxes all edges leaving  $u$ . In the following implementation, we use a min-priority queue  $Q$  of vertices, keyed by their  $d$  values.

|   |   |
|---|---|
| <pre>DIJKSTRA(<math>G, w, s</math>) 1  INITIALIZE-SINGLE-SOURCE(<math>G, s</math>) 2  <math>S = \emptyset</math> 3  <math>Q = G.V</math> 4  <b>while</b> <math>Q \neq \emptyset</math> 5    <math>u = \text{EXTRACT-MIN}(Q)</math> 6    <math>S = S \cup \{u\}</math> 7    <b>for</b> each vertex <math>v \in G.\text{Adj}[u]</math> 8      RELAX(<math>u, v, w</math>)</pre> | <pre>INITIALIZE-SINGLE-SOURCE(<math>G, s</math>) 1. <b>for</b> each vertex <math>v \in G.v</math> 2. <math>v.d = \infty</math> 3. <math>v.\pi = \text{NIL}</math> 4. <math>s.d = 0</math></pre><br><pre>RELAX(<math>u, v, w</math>) 1 <b>if</b> <math>v.d &gt; u.d + w(u, v)</math> 2 <math>v.d = u.d + w(u, v)</math> 3 <math>v.\pi = u</math></pre> |
|---|---|

Dijkstra's algorithm relaxes edges as shown in Figure 24.6. Line 1 initializes the  $d$  and  $\pi$  values in the usual way, and line 2 initializes the set  $S$  to the empty set. The algorithm maintains the invariant that  $Q = V - S$  at the start of each iteration of the **while** loop of lines 4–8. Line 3 initializes the min-priority queue  $Q$  to contain all the vertices in  $V$ ; since  $S = \emptyset$  at that time, the invariant is true after line 3. Each time through the **while** loop of lines 4–8, line 5 extracts a vertex  $u$  from  $Q = V - S$  and line 6 adds it to set  $S$ , thereby maintaining the invariant. (The first time through this loop,  $u = s$ .) Vertex  $u$ , therefore, has the smallest shortest-path estimate of any vertex in  $V - S$ . Then, lines 7–8 relax each edge  $(u, v)$  leaving  $u$ , thus updating the estimate  $v.d$  and the predecessor  $v.\pi$  if we can improve the shortest path to  $v$  found so far by going through  $u$ . Observe that the algorithm never inserts vertices into  $Q$  after line 3 and that each vertex is extracted from  $Q$  and added to  $S$  exactly once, so that the while loop of lines 4–8 iterates exactly  $|V|$  times.



Example:

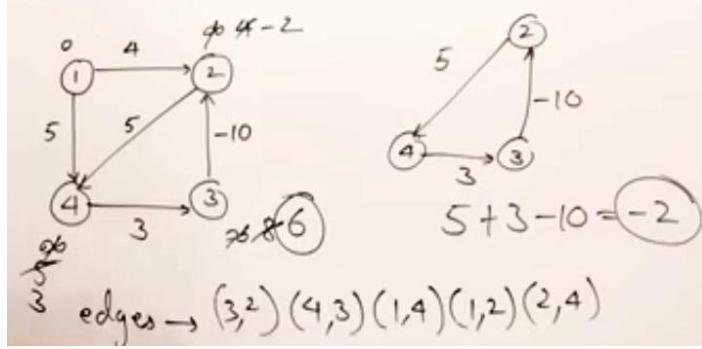


**Figure** The execution of Dijkstra's algorithm. The source  $s$  is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set  $S$ , and white vertices are in the min-priority queue  $Q = V - S$ . (a) The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum  $d$  value and is chosen as vertex  $u$  in line 5. (b)–(f) The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex  $u$  in line 5 of the next iteration. The  $d$  values and predecessors shown in part (f) are the final values.

**Time Complexity:**

Time Complexity of the implementation is  $O(V^2)$ . If the input graph is represented using adjacency list, it can be reduced to  $O(E \log V)$  with the help of binary heap.

**Note:** Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, Bellman–Ford algorithm can be used.

**Bellman–Ford algorithm****The Bellman–Ford algorithm**

solves the single-source shortest-paths problem in which edge weights may be negative. It returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, it produces the shortest paths and their weights.

## 24.1 The Bellman-Ford algorithm

The **Bellman-Ford algorithm** solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph  $G \in D(V; E)$  with source  $s$  and weight function  $w: W \rightarrow \mathbb{R}$ , the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

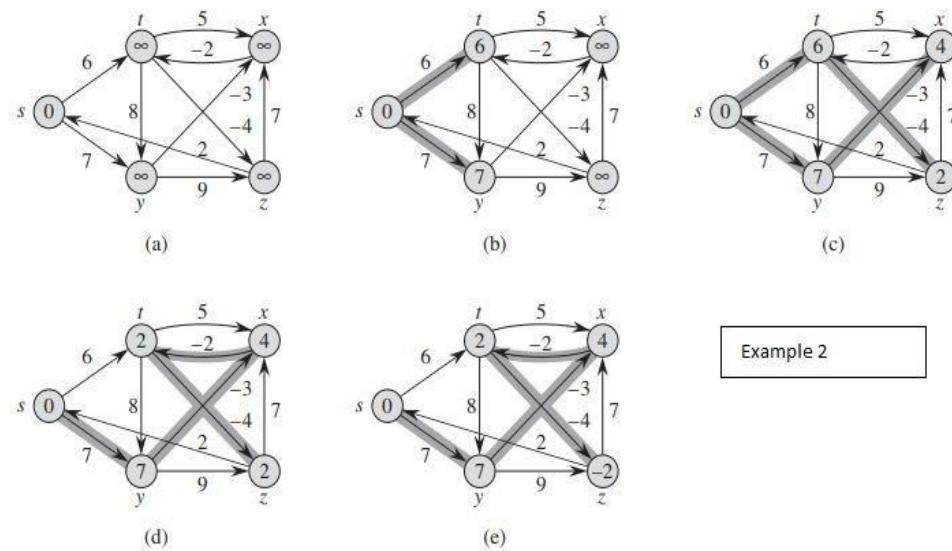
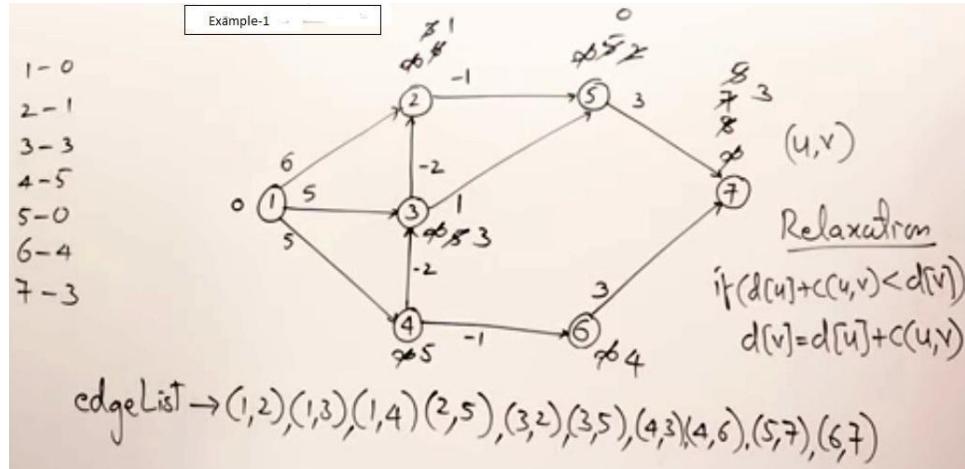
The algorithm relaxes edges, progressively decreasing an estimate  $d$  on the weight of a shortest path from the source  $s$  to each vertex  $v \in V$  until it achieves the actual shortest-path weight  $\text{dist}_s(v)$ . The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD.G;w;s/
Time Complexity O(V.E)
1      INITIALIZE-SINGLE-SOURCE.G;S/
2      for i = 1 to jG:Vj - 1
3          for each edge .u; / 2 G:E
4              RELAX.u;;w/
5          for each edge .u; / 2 G:E
6              if :d > u:d C w.u;/
7          return FALSE
8      return TRUE
```

Figure 24.4 shows the execution of the Bellman-Ford algorithm on a graph with 5 vertices. After initializing the  $d$  and  $\text{dist}_s$  values of all vertices in line 1, the algorithm makes  $jV - 1$  passes over the edges of the graph. Each pass is one iteration of the **for** loop of lines 2–4 and consists of relaxing each edge of the graph once. Figures 24.4(b)–(e) show the state of the algorithm after each of the four passes over the edges. After making  $jV - 1$  passes, lines 5–8 check for a negative-weight cycle and return the appropriate boolean value. (We'll see a little later why this check works.)

The Bellman-Ford algorithm runs in time  $O(VE)$ , since the initialization in line 1 takes  $V$  time, each of the  $V - 1$  passes over the edges in lines 2–4 takes  $E$  time, and the **for** loop of lines 5–7 takes  $O(E)$  time.

To prove the correctness of the Bellman-Ford algorithm, we start by showing that if there are no negative-weight cycles, the algorithm computes correct shortest-path weights for all vertices reachable from the source.



**Figure —** The execution of the Bellman-Ford algorithm. The source is vertex  $s$ . The  $d$  values appear within the vertices, and shaded edges indicate predecessor values: if edge  $(u, v)$  is shaded, then  $v.\pi = u$ . In this particular example, each pass relaxes the edges in the order  $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$ . (a) The situation just before the first pass over the edges. (b)-(e) The situation after each successive pass over the edges. The  $d$  and  $\pi$  values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

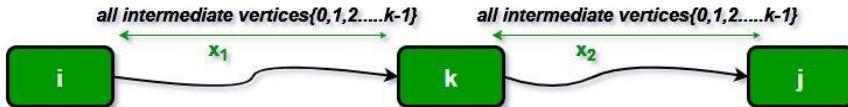
### The Floyd-Warshall algorithm

The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number  $k$  as an intermediate vertex, we already have considered vertices  $\{0, 1, 2, \dots, k-1\}$  as intermediate vertices. For every pair  $(i, j)$  of the source and destination vertices respectively, there are two possible cases. 1)  $k$  is not an intermediate vertex in shortest path from  $i$  to  $j$ . We keep the value of  $dist[i][j]$  as it is.

2)  $k$  is an intermediate vertex in shortest path from  $i$  to  $j$ . We update the value of  $dist[i][j]$  as  $dist[i][k] + dist[k][j]$  if  $dist[i][j] > dist[i][k] + dist[k][j]$

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



Example 1

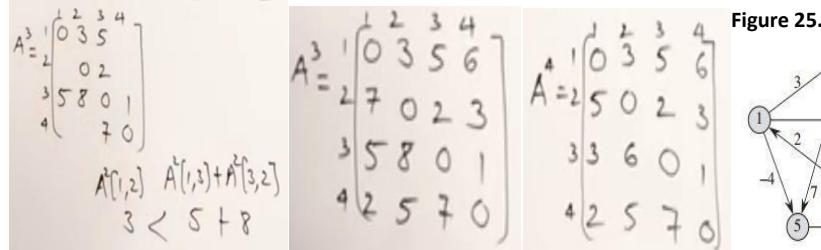
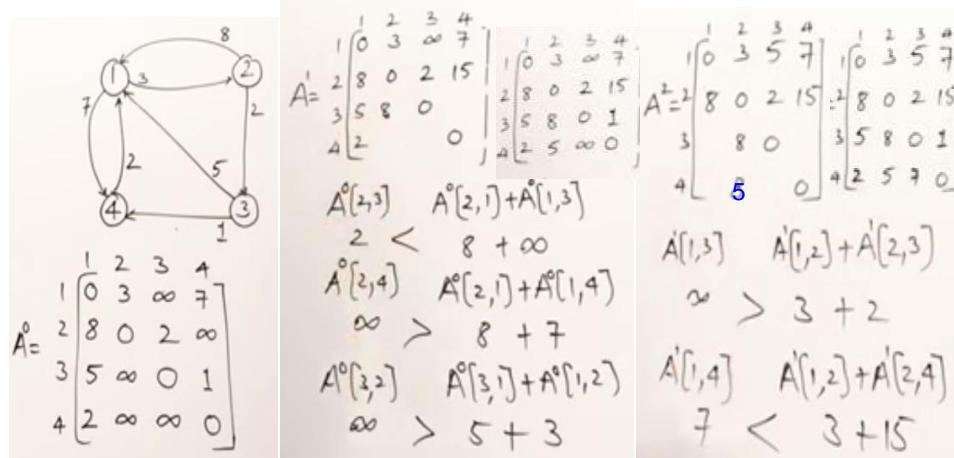
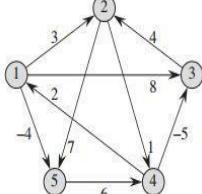


Figure 25.1 Example 2



FLOYD-WARSHALL(W /

1 n D W:rows

2 D.0/ D W

3 **for** k D 1 to n

4 let  $D^{k/}$  D  $d_{ij}^{k/}$  be a new  $n \times n$  matrix

**PRINT-ALL-PAIRS-SHORTEST-PATH** ( $\pi, i, j$ )

if  $i == j$

print i

elseif  $\pi_{ij} == NIL$

print "no path from" i "to" j "exists"

else **PRINT-ALL-PAIRS-SHORTEST-PATH** ( $\pi, i,$

$\pi_{ij})$  print j

5 **for** i D 1 to n

6 **for** j D 1 to n

7  $d_{ij}^{k/} \leftarrow \min(d_{ij}^{k/}; d_{ik}^{k/} + d_{kj}^{k/})$

8 **return**  $D^{n/}$

Figure 25.4 shows the matrices  $D^{k/}$  computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

The running time of the Floyd-Warshall algorithm is determined by the triply nested **for** loops of lines 3–7. Because each execution of line 7 takes  $O(1)$  time, the algorithm runs in time  $O(n^3)$ . As in the final algorithm in Section 25.1, the code is tight, with no elaborate data structures, and so the constant hidden in the  $O$ -notation is small. Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

### Constructing a shortest path

There are a variety of different methods for constructing shortest paths in the Floyd-Warshall algorithm. One way is to compute the matrix  $D$  of shortest-path weights and then construct the predecessor matrix ... from the  $D$  matrix. Exercise 25.1-6 asks you to implement this method so that it runs in  $O(n^3)$  time. Given the predecessor matrix ..., the **PRINT-ALL-PAIRS-SHORTEST-PATH** procedure will print the vertices on a given shortest path.

Alternatively, we can compute the predecessor matrix ... while the algorithm computes the matrices  $D^{k/}$ . Specifically, we compute a sequence of matrices

## 85 | Page

$\dots^{0/}; \dots^{1/}; \dots; \dots^{n/}$ , where  $\dots D \dots^{n/}$  and we define  $i_j^{k/}$  as the predecessor of vertex  $j$  on a shortest path from vertex  $i$  with all intermediate vertices in the set  $f_1; 2; \dots; k$ .

We can give a recursive formulation of  $i_j^{k/}$ . When  $k \leq 0$ , a shortest path from  $i$  to  $j$  has no intermediate vertices at all. Thus,

$$i_j^{0/} D (\text{NIL if } i D j \text{ or } w_{ij} D 1 ; (25.6) i \text{ if } i \neq j \text{ and } w_{ij} < 1 :$$

For  $k \geq 1$ , if we take the path  $i \xrightarrow{k} j$ , where  $k \neq j$ , then the predecessor of  $j$  we choose is the same as the predecessor of  $j$  we chose on a shortest path from  $k$  with all intermediate vertices in the set  $f_1; 2; \dots; k-1$ . Otherwise, we

[choose the same predecessor of  \$j\$  that we chose on a shortest path from  \$i\$  with all intermediate vertices in the set  \$\{1, 2, \dots, k-1\}\$ .](#)

696

Chapter 25 All-Pairs Shortest Paths

D.0/D1 4      0 1 1

2 1 5 0 1 1 1 1

6 0

▼

NIL    1    1    NIL    1

0    3    8 1 4

▼

NIL    NIL    NIL    2    2

...<sup>0/</sup>DNIL    3    NIL    NIL    NIL

4    NIL    4    NIL

1 0 1 1    7    NIL NIL    NIL    NIL    5

NIL

▼

0    3    8 1 4    NIL 1    1    NIL    1

▼

1 0 1 1    7 NIL    NIL    NIL    2    2

D.1/D1 4    0 1 1    ...<sup>1/</sup>DNIL    3    NIL    NIL    NIL

2    5 5 0 2 4    1    4    NIL    1

1 1 1 6    0 NIL    NIL    NIL    5    NIL

|                        |             |                      |                           |       |     |           |     |
|------------------------|-------------|----------------------|---------------------------|-------|-----|-----------|-----|
|                        | 03          | 8                    | 44                        | NIL 1 | 1   | 2         | 1   |
| D <sup>2/</sup> D1 4 0 | 10 1 1      | 7NIL                 | NIL                       | NIL   | 2   | 2         |     |
| 5 5                    | 5           | 11                   | ... <sup>2/</sup> D NIL 3 | NIL   | 2   | 2 2       |     |
| 0 24                   | 0 24        | 1                    | 4                         | NIL   | 1   |           |     |
| 1 1 1 6                |             |                      | 0NIL                      | NIL   | NIL | 5         | NIL |
|                        | 03          | 8                    | 44                        | NIL 1 | 1   | 2         | 1   |
| D <sup>3/</sup> D1 4 0 | 10 1 1      | 7NIL                 | NIL                       | NIL   | 2   | 2         |     |
| 0 24                   | 5           | 11                   | ... <sup>3/</sup> D NIL 3 | NIL   | 2   | 2 2 1 5   |     |
| 3                      | 3           | 4                    | NIL                       | 1     |     |           |     |
| 1 1 1 6                |             |                      | 0NIL                      | NIL   | NIL | 5         | NIL |
| 0                      | 3 1         | 4 4 NIL              | 1                         | 4     | 2   | 1 2 1 5 0 |     |
| 2                      | 4 3 4 NIL 1 | 3 0 4 1 14 NIL 4 2 1 | D <sup>4/</sup> D7        | 4     | 0   |           |     |
| 5                      | 3           | ... <sup>4/</sup> D4 | 3                         | NIL   | 2   |           |     |
| 1                      |             |                      |                           |       |     |           |     |
| 8                      | 5           | 1 6                  | 04                        | 3     | 4   | 5         | NIL |
| 0                      | 1 3         | 2 4 NIL              | 3                         | 4     | 5   | 1 2 1 5 0 |     |
| 2                      | 4 3 4 NIL 1 | 3 0 4 1 14 NIL 4 2 1 | D <sup>5/</sup> D7        | 4     | 0   |           |     |
| 5                      | 3           | ... <sup>5/</sup> D4 | 3                         | NIL   | 2   |           |     |
| 1                      |             |                      |                           |       |     |           |     |
| 8                      | 5           | 1 6                  | 04                        | 3     | 4   | 5         | NIL |

**Figure 25.4** The sequence of matrices  $D^{k/}$  and  $\dots^{k/}$  computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

## UNIT: 4

### Difference between Backtracking & Branch and Bound

#### **Backtracking:**

- \* It is used to find all possible solution available to the problem
- \* It traverse tree by DFS (Depth first Search)
- \* It realizes that it has made a bad choice & undoes the last choice by tracking up.
- \* It search the state space tree until it found a solution

Some Problem Solved with Backtracking Technique

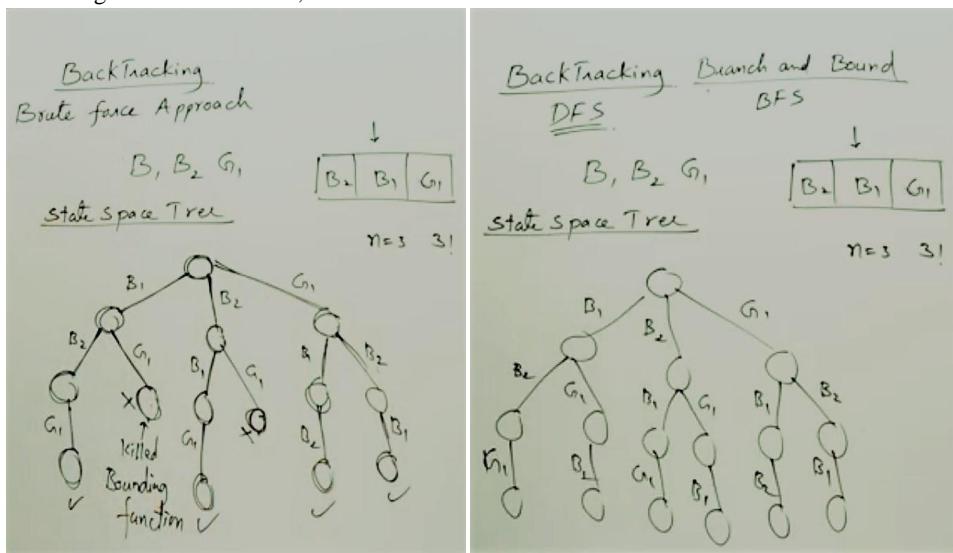
- N- Queens Problem, Sum of Subset, Sudoku Puzzle, Hamiltonian Cycle

#### **(BB):**

- \* It is used to solve optimization problem \*It may traverse the tree by BFS or DFS.
- \* It realizes that is already has a better optional solution that the pre-solution leads to so it abandons that pre-solution.
- \* It completely searches the **state space solution tree** to get optional solution.

Some Problem Solved with Branch and Bound Technique

- Traveling Salesman Problem, Vertex Cover Problem



In the above figure ->**Bounding Function:** No girl sit in the middle of two boys

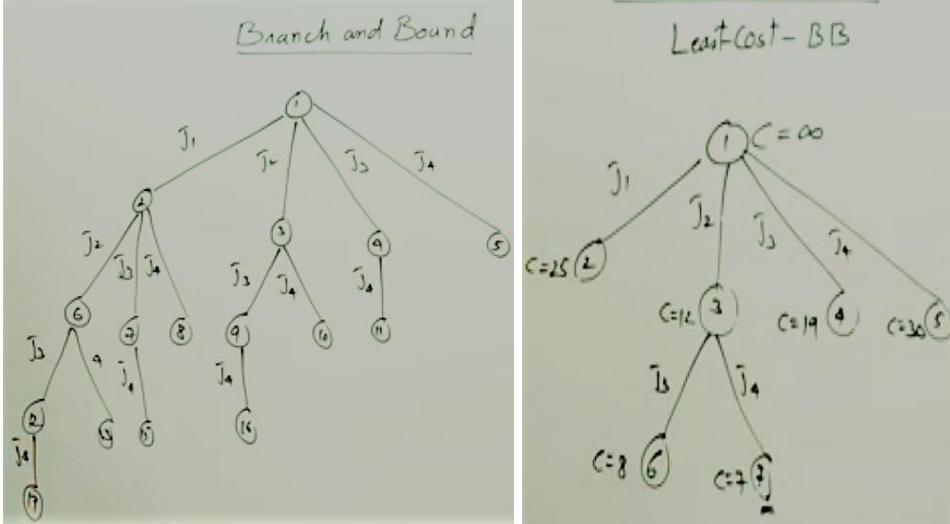
Types of Branch and Bound

**1. FIFO Branch and Bound Search:** This leads to a breadth-first search.

For this we will use a data structure called Queue. Initially Queue is empty.

**2. LIFO Branch and Bound Search:** This leads to a depth-first search For this we will use a data structure called stack. Initially stack is empty.

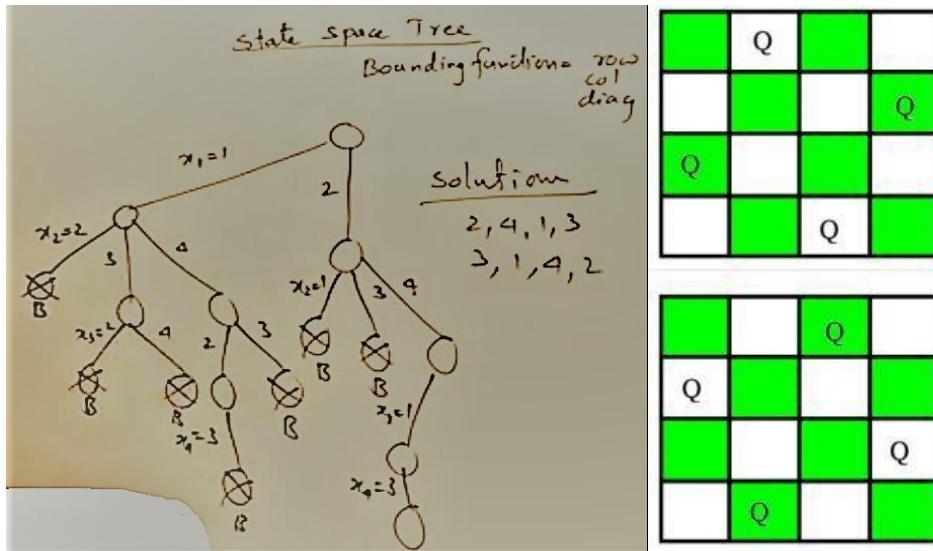
**3. LC (lowest/least cost) Branch and Bound Search:** the branch is extended by the node which adds the lowest additional costs, according to a given cost function.



### N-Queens Problem

Nauck made an 8X8 Chessboard to find the first Feasible Solution. In a NxN square board N – number of queens need to be placed considering three Condition ---

- No two Queens can be placed in same row, same column, and same diagonal. **Time Complexity of N-Queen Problem is O (n!)**



```

1  Algorithm Place( $k, i$ )
2  // Returns true if a queen can be placed in  $k$ th row and
3  //  $i$ th column. Otherwise it returns false.  $x[ ]$  is a
4  // global array whose first  $(k - 1)$  values have been set.
5  //  $\text{Abs}(r)$  returns the absolute value of  $r$ .
6  {
7      for  $j := 1$  to  $k - 1$  do
8          if  $((x[j] = i) \text{ // Two in the same column}$ 
9              or  $(\text{Abs}(x[j] - i) = \text{Abs}(j - k)) \text{ // or in the same diagonal}$ 
10             then return false;
11     return true;
12 }
13 }
```

---

**Algorithm 7.4** Can a new queen be placed?

---

```

1  Algorithm NQueens( $k, n$ )
2  // Using backtracking, this procedure prints all
3  // possible placements of  $n$  queens on an  $n \times n$ 
4  // chessboard so that they are nonattacking.
5  {
6      for  $i := 1$  to  $n$  do
7          {
8              if Place( $k, i$ ) then
9                  {
10                       $x[k] := i;$ 
11                      if  $(k = n)$  then write ( $x[1 : n]$ );
12                      else NQueens( $k + 1, n$ );
13                  }
14          }
15 }
```

---

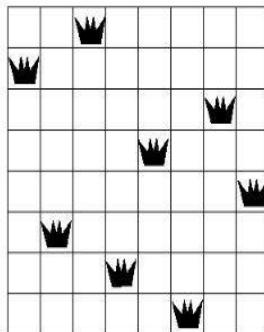
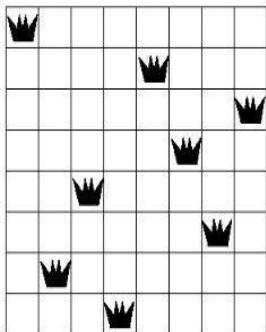
**Algorithm 7.5** All solutions to the  $n$ -queens problem

### Backtracking Algorithm

The idea is to place queens one by one in different columns, starting from the leftmost column.

When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

- 1) Start in the leftmost column
- 2) If all queens are placed return true
- 3) Try all rows 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 solution and recursively check if placing queen here leads to a solution. b) If placing 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) and go to step (a) to try other rows.
- 3) If all rows have been tried and nothing worked, return false to trigger backtracking.



|   |   |   |   |
|---|---|---|---|
| Q |   |   |   |
|   |   | Q |   |
|   |   |   | Q |
| Q |   |   |   |
|   | Q |   |   |
|   |   | Q |   |
|   |   |   | Q |
| Q |   |   |   |

|   |   |   |   |
|---|---|---|---|
| Q |   |   |   |
|   |   | Q |   |
|   |   |   | Q |
| Q |   |   |   |
|   | Q |   |   |
|   |   | Q |   |
|   |   |   | Q |
| Q |   |   |   |

|   |   |   |   |
|---|---|---|---|
| Q |   |   |   |
|   |   | Q |   |
|   |   |   | Q |
| Q |   |   |   |
|   | Q |   |   |
|   |   | Q |   |
|   |   |   | Q |
| Q |   |   |   |

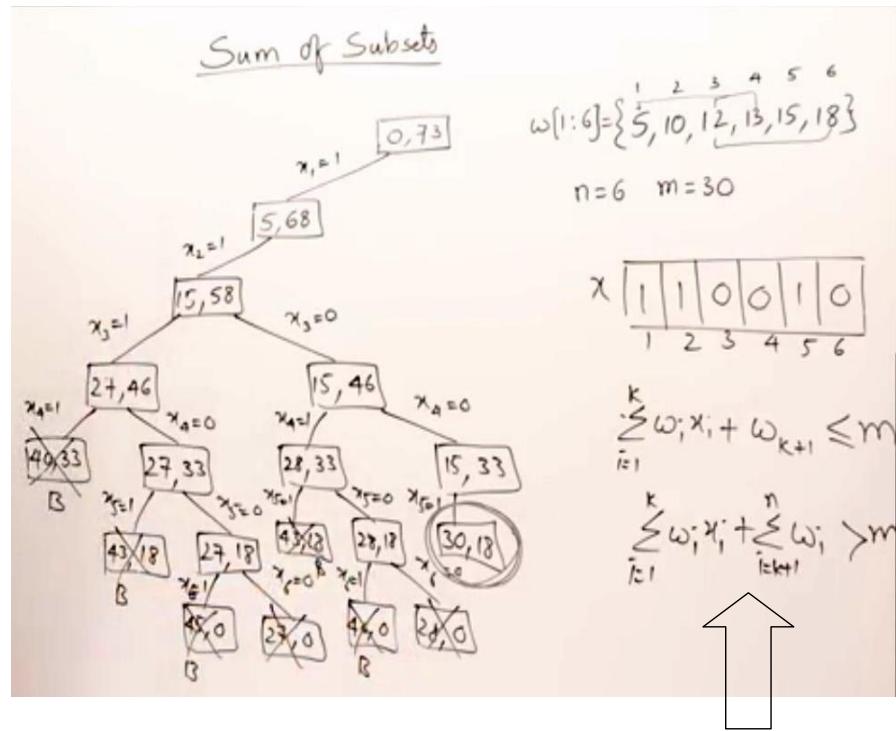
Altogether, **92** solutions to the problem of **EIGHT QUEENS**.

### Sub Set Sum Problem/ Sum of Subset Problem

The Sum of Subset Problem is, there will be a set of distinct positive Numbers X and a given Number N. Now, we have to find all the combination of numbers from X that sum up to N. Make a Set of that Number. **Backtracking Approach:**

First, organize the numbers in non decreasing order. Then generating a tree, we can find the solution by adding the weight of the nodes. Note that, here more than necessary nodes can be generated. The algorithm will search for the weighted amount of the nodes. If it goes beyond given Number N, then it stops generating nodes and move to other parent nodes for finding solution.

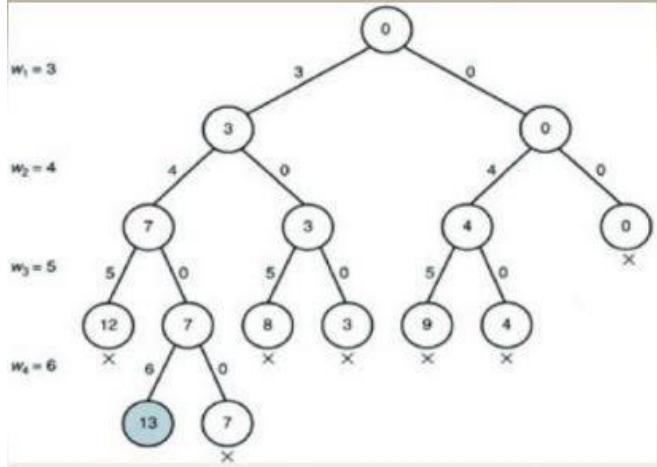
**Example 1:**



**BOUNDING FUNCTION Time**  
**Complexity of Sum of Subset Problem:**  
 $O(2^n)$

**Example: 2**

$W = \{4, 5, 6, 3\}$  and  $M = 13$



```

1 Algorithm SumOfSub( $s, k, r$ )
2 // Find all subsets of  $w[1 : n]$  that sum to  $m$ . The values of  $x[j]$ ,
3 //  $1 \leq j < k$ , have already been determined.  $s = \sum_{j=1}^{k-1} w[j] * x[j]$ 
4 // and  $r = \sum_{j=k}^n w[j]$ . The  $w[j]$ 's are in nondecreasing order.
5 // It is assumed that  $w[1] \leq m$  and  $\sum_{i=1}^n w[i] \geq m$ .
6 {
7     // Generate left child. Note:  $s + w[k] \leq m$  since  $B_{k-1}$  is true.
8      $x[k] := 1$ ;
9     if ( $s + w[k] = m$ ) then write ( $x[1 : k]$ ); // Subset found
10    // There is no recursive call here as  $w[j] > 0$ ,  $1 \leq j \leq n$ .
11    else if ( $s + w[k] + w[k + 1] \leq m$ )
12        then SumOfSub( $s + w[k], k + 1, r - w[k]$ );
13    // Generate right child and evaluate  $B_k$ .
14    if (( $s + r - w[k] \geq m$ ) and ( $s + w[k + 1] \leq m$ )) then
15    {
16         $x[k] := 0$ ;
17        SumOfSub( $s, k + 1, r - w[k]$ );
18    }
19 }
```

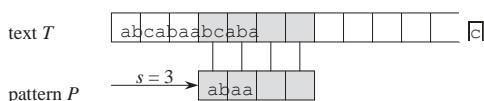
Recursive backtracking algorithm for Sum of Subset Problem

## 32 String Matching

Text-editing programs frequently need to find all occurrences of a pattern in the text. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem—called “string matching”—can greatly aid the responsiveness of the text-editing program. Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences. Internet search engines also use them to find Web pages relevant to queries.

We formalize the string-matching problem as follows. We assume that the text is an array  $T[1:n]$  of length  $n$  and that the pattern is an array  $P[1:m]$  of length  $m$ . We further assume that the elements of  $P$  and  $T$  are characters drawn from a finite alphabet  $\Sigma$ . For example, we may have  $\Sigma = \{f, g\}$  or  $\Sigma = \{a, b, \dots, z\}$ . The character arrays  $P$  and  $T$  are often called *strings* of characters.

Referring to Figure 32.1, we say that pattern  $P$  *occurs with shift s* in text  $T$  (or, equivalently, that pattern  $P$  *occurs beginning at position s*) if  $T[s:n] = P[1:m]$  (that is, if  $T[s+j-1] = P[j]$ , for  $1 \leq j \leq m$ ). If  $P$  occurs with shift  $s$  in  $T$ , then we call  $s$  a *valid shift*; otherwise, we call  $s$  an *invalid shift*. The *string-matching problem* is the problem of finding all valid shifts with which a given pattern  $P$  occurs in a given text  $T$ .



**Figure 32.1** An example of the string-matching problem, where we want to find all occurrences of the pattern  $P = abaa$  in the text  $T = Dabcabaabcabac$ . The pattern occurs only

once in the text, at shift  $s \geq 3$ , which we call a valid shift. A vertical line connects each character of the pattern to its matching character in the text, and all matched characters are shaded.

*Chapter 32 String Matching*

### 32.1 The naive string-matching algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition  $P[1:m] == T[s:s+m]$  for each of the  $n - m + 1$  possible values of  $s$ .

```
NAIVE-STRING-MATCHER(T,P)
1      n D T:length
2      m D P:length
3      for s D 0 to n - m
4          if P[1:m] == T[s:s+m]
5              print "Pattern occurs with shift" s
```

Figure 32.4 portrays the naive string-matching procedure as sliding a “template” containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text. The **for** loop of lines 3–5 considers each possible shift explicitly. The test in line 4 determines whether the current shift is valid; this test implicitly loops to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift  $s$ .

Procedure NAIVE-STRING-MATCHER takes time  $O(n - m + 1)/m$ , and this bound is tight in the worst case. For example, consider the text string  $a^n$  (a string of  $n$   $a$ 's) and the pattern  $a^m$ . For each of the  $n - m + 1$  possible values of the shift  $s$ , the implicit loop on line 4 to compare corresponding characters must execute  $m$  times to validate the shift. The worst-case running time is thus  $(n - m + 1)m$ , which is  $\Theta(n^2)$  if  $m = \Theta(n)$ . Because it requires no preprocessing, NAIVE-STRING-MATCHER's running time equals its matching time.



(a) (b) (c) (d)

**Figure 32.4** The operation of the naive string matcher for the pattern  $P = D\ a a b$  and the text  $T = D a c a a b c$ . We can imagine the pattern  $P$  as a template that we slide next to the text. **(a)-(d)** The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. The algorithm finds one occurrence of the pattern, at shift  $s = 2$ , shown in part (c).

## KMP (Knuth Morris Pratt) Pattern Searching

The [Naive pattern searching algorithm](#) doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

```
txt[] = "AAAAAAAAAAAAAAAB"
pat[] = "AAAAB"

txt[] = "ABABABCABABCABABC"
pat[] = "ABABAC" (not a worst case, but a bad case for Naive)
```

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to  $O(n)$ . The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match. Let us consider below example to understand this.

### Matching Overview

```
txt = "AAAAABAAABA"
pat = "AAAA"
```

We compare first window of `txt` with `pat`

```
txt = "AAAAABAAABA"
pat = "AAAA" [Initial position]
We find a match. This is same as Naive String Matching.
```

In the next step, we compare next window of `txt` with `pat`.

```
txt = "AAAAAABAABA"
pat = "AAAA" [Pattern shifted one position]
This is where KMP does optimization over Naive. In this
second window, we only compare fourth A of pattern
with fourth character of current window of text to decide
whether current window matches or not. Since we know
first three characters will anyway match, we skipped
matching first three characters.
```

### Need of Preprocessing?

An important question arises from the above explanation, how to know how many characters to be skipped. To know this, we pre-process pattern and prepare an integer array `lps[]` that tells us the count of characters to be skipped.

## Preprocessing Overview:

- KMP algorithm preprocesses `pat[]` and constructs an auxiliary `lps[]` of size  $m$  (same as size of pattern) which is used to skip characters while matching.
- **name `lps` indicates longest proper prefix which is also suffix.** . A proper prefix is prefix with whole string **not** allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC" and "ABC".
- We search for `lps` in sub-patterns. More clearly we focus on sub-strings of patterns that are either prefix and suffix.
- For each sub-pattern `pat[0..i]` where  $i = 0$  to  $m-1$ , `lps[i]` stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern `pat[0..i]`.

`lps[i]` = the longest proper prefix of `pat[0..i]`  
which is also a suffix of `pat[0..i]`.

**Note :** `lps[i]` could also be defined as longest prefix which is also proper suffix. We need to use properly at one place to make sure that the whole substring is not considered.

Examples of `lps[]` construction:

For the pattern “AAAA”,  
`lps[]` is [0, 1, 2, 3]

For the pattern “ABCDE”,  
`lps[]` is [0, 0, 0, 0, 0]

For the pattern “AABAACAABAA”,  
`lps[]` is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern “AACACAAAAC”,  
`lps[]` is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pattern “AAABAAA”,  
`lps[]` is [0, 1, 2, 0, 1, 2, 3]

### Searching Algorithm:

Unlike [Naive algorithm](#), where we slide the pattern by one and compare all characters at each shift, we use a value from `lps[]` to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

How to use `lps[]` to decide next positions (or to know a number of characters to be skipped)?

- We start comparison of `pat[j]` with  $j = 0$  with characters of current window of text.
- We keep matching characters `txt[i]` and `pat[j]` and keep incrementing  $i$  and  $j$  while `pat[j]` and `txt[i]` keep **matching**.
- When we see a **mismatch**
  - We know that characters `pat[0..j-1]` match with `txt[i-j+1...i-1]` (Note that  $j$  starts with 0 and increment it only when there is a match).
  - We also know (from above definition) that `lps[j-1]` is count of characters of `pat[0...j-1]` that are both proper prefix and suffix.
  - From above two points, we can conclude that we do not need to match these `lps[j-1]` characters with `txt[i-j...i-1]` because we know that these characters will anyway match. Let us consider above example to understand this.

```
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
lps[] = {0, 1, 2, 3}

i = 0, j = 0
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 1, j = 1
txt[] = "AAAABAAABA"
pat[] = "AAA"
txt[i] and pat[j] match, do i++, j++
```

```

i = 2, j = 2 txt[]
= "AAAABAAABA"
pat[] = "AAAA"
pat[i] and pat[j] match, do i++,
j++ i = 3, j = 3 txt[] =
"AAAABAAABA" pat[] = "AAAA"
txt[i] and pat[j] match, do i++,
j++ i = 4, j = 4
Since j == M, print pattern found and reset j,
j = lps[j-1] = lps[3] = 3

```

Here unlike Naive algorithm, we do not match first three characters of this window. Value of lps[j-1] (in above step) gave us index of next character to match.

```

i = 4, j = 3 txt[] = "AAAAABAAABA"
pat[] = "AAAA" txt[i] and pat[j]
match, do i++, j++ i = 5, j = 4
Since j == M, print pattern found and reset j,
j = lps[j-1] = lps[3] = 3

```

Again unlike Naive algorithm, we do not match first three characters of this window. Value of lps[j-1] (in above step) gave us index of next character to match.

```

i = 5, j = 3 txt[] = "AAAAABAAABA" pat[] = "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only
j j = lps[j-1] = lps[2] = 2
i = 5, j = 2 txt[]
= "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[1] = 1
i = 5, j = 1 txt[]
= "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[0] = 0
i = 5, j = 0 txt[] = "AAAAABAAABA" pat[] =
"AAAA" txt[i] and pat[j] do NOT match and j is 0, we
do i++.
i = 6, j = 0 txt[]
= "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++ and j++
i = 7, j = 1 txt[] = "AAAAABAABA" pat[] = "AAAA" txt[i] and
pat[j] match, do i++ and j++ We continue this way...

```

Knuth–Morris–Pratt algorithm { Worst case Time complexity to  $O(n)$ }

| q        | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----------|---|---|---|---|---|---|---|
| P[q]     | a | b | c | d | a | b | d |
| $\pi[q]$ | 0 | 0 | 0 | 0 | 1 | 2 | 0 |

**COMPUTE-PREFIX-FUNCTION(P)**Input: pattern P of length mOutput: table  $\pi[1,..,m]$ 

```

1  m = P.length
2  let  $\pi[1,..,m]$  be new array
3   $\pi[1] = 0$ 
4  k = 0
5  For q = 2 to m    // Scan the Pattern P[2,,m] from left to right
6    while k > 0 and P[k + 1] ≠ P[q]
7      k =  $\pi[k]$     // The prefix  $P_k$  is not the suffix of  $P_q$ 
8    if P[k + 1] == P[q]
9      k = k + 1    // The longest prefix  $P_k$  is also a proper suffix of  $P_q$       K = 2
10    $\pi[q] = k$ 
11  Return  $\pi$ 

```

**KMP-MATCHER(T,P)**Input: pattern P of length m and text T of length nOutput: list of all numbers s, such that P occurs with shift s in T

```

1. n = T.length
2. m = P.length
3.  $\pi = \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4. q = 0          // number of characters matched
5. for i = 1 to n // scan the text T[1,,n] from left to right
6.   while q > 0 and P[q + 1] ≠ T[i] // when mismatch occurred
7.     q =  $\pi[q]$ 
8.   If P[q + 1] == T[i] //next character matches
9.     q = q + 1
10.  If q == m          // is all of Pattern P[1,,m] matched?
11.    Print "Pattern occurs with shift" i - m
12.    q =  $\pi[q]$           // look for the next match

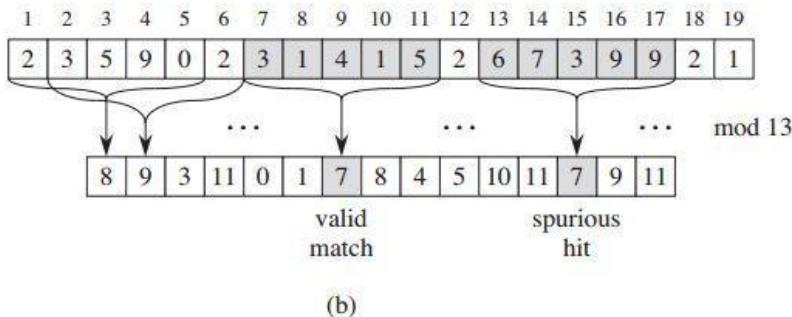
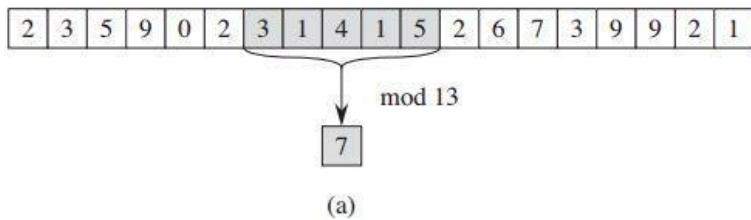
```

### The Rabin-Karp algorithm

Rabin-Karp is another pattern searching algorithm to find the pattern in a more efficient way. It also checks the pattern by moving window one by one, but without checking all characters for all cases, it finds the hash value. When the hash value is matched, then only it tries to check each character. This procedure makes the algorithm more efficient.

The Rabin-Karp algorithm makes use of hash functions and the rolling hash technique. A hash function is essentially a function that maps one thing to a value. In particular, hashing can map data of arbitrary size to a value of fixed size.

**Hashing:** Hashing is a way to associate values using a hash function to map an input to an output. The purpose of a hash is to take a large piece of data and be able to be represented by a smaller form. Example 1:



The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13. (a) A text string. A window of length 5 is shown shaded. The numerical value of the shaded number, computed modulo 13, yields the value 7. (b) The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern  $P = 31415$ , we look for windows whose value modulo 13 is 7, since  $31415 \equiv 7 \pmod{13}$ . The algorithm finds two such windows, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit.

#### Time complexity:

*The time complexity is  $O(m+n)$  [Non Spurious hit], but for the worst case, it is  $O(mn)$  [Spurious hit]*

**Example 2:**

**Text=31415926535 Pattern=26**

Hash function =  $26 \bmod 13 = 0$   
 Window of length 2 taken  
 $31 \bmod 13 = 5$  Slide Window  
 $14 \bmod 13 = 1$   
 $41 \bmod 13 = 2$   
 $15 \bmod 13 = 2$   
 $59 \bmod 13 = 7$   
 $92 \bmod 13 = 1$   
 $26 \bmod 13 = 0$  [Non Spurious hit][valid match]  
 $65 \bmod 13 = 0$  [Spurious hit][not valid match]  
 $53 \bmod 13 = 1$

$35 \bmod 13 = 9$   
**Example 3:**  
 Text = cadba {Take ASCII Value}  
 Text = 99971009897  
 Pattern = db  
 Pattern = 10098  
 Hash Function =  $10098 \bmod 13 = 10$

### Rabin Karp ( $T, P$ )

1.  $n = T.length$
2.  $m = P.length$
3.  $h(P) = \text{hash}(P)$
4.  $h(T) = \text{hash}(T)$
5. for  $s=0$  to  $n-m$
6.    If ( $h(P) = h(T)$ )
7.     If  $[P(0 \dots m-1)] = T[s \dots s+m-1]$   
         point "Pattern found with shift" s
- 8.
9.    If ( $s < n-m$ )
10.     $h(T) = \text{hash}(T(s+1 \dots s+m))$



## NP-Completeness | Set 1 (Introduction)

We have been writing about efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.

**Can all computational problems be solved by a computer?** There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source [Halting Problem](#)).

Status of **NP Complete** problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomialtime algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

### What are **NP**, **P**, **NP-complete** and **NP-Hard** problems?

**P** is set of problems that can be solved by a deterministic Turing machine in **Polynomial** time.

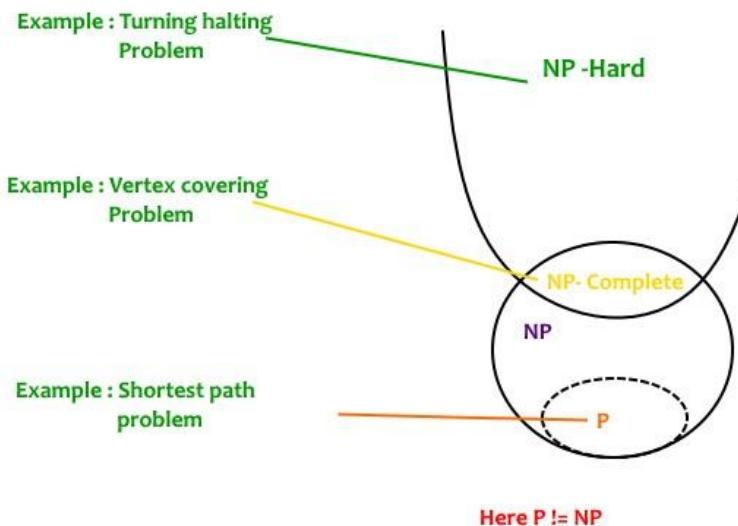
**NP** is set of decision problems that can be solved by a **Non-deterministic Turing Machine** in **Polynomial** time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).

Informally, NP is set of decision problems which can be solved by a polynomial time via a “**Lucky Algorithm**”, a magical algorithm that always makes a right guess among the given set of choices (Source [Ref 1](#)).

**NP-complete** problems are the hardest problems in **NP** set. A decision problem L is NP-complete if: **1)** L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).

**2)** Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is **NP-Hard** if it follows property 2 mentioned above, doesn't need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.



#### Decision vs Optimization Problems

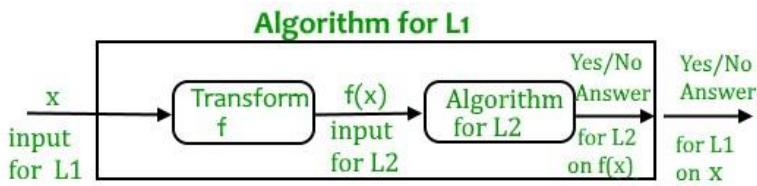
NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source [Ref 2](#)).

For example, consider the **vertex cover problem** (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph  $G$  and  $k$ , is there a vertex cover of size  $k$ ?

#### What is Reduction?

Let  $L_1$  and  $L_2$  be two decision problems. Suppose algorithm  $A_2$  solves  $L_2$ . That is, if  $y$  is an input for  $L_2$  then algorithm  $A_2$  will answer Yes or No depending upon whether  $y$  belongs to  $L_2$  or not.

The idea is to find a transformation from  $L_1$  to  $L_2$  so that the algorithm  $A_2$  can be part of an algorithm  $A_1$  to solve  $L_1$ .



Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra's algorithm to find shortest path, we can take log of all weights and use Dijkstra's algorithm to find the minimum product path rather than writing a fresh code for this new problem.

#### **How to prove that a given problem is NP complete?**

From the definition of NP-complete, it appears impossible to prove that a problem L is NPComplete. By definition, it requires us to show that every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

#### **What was the first problem proved as NP-Complete?**

There must be some first NP-Complete problem proved by definition of NP-Complete problems. SAT (**Boolean satisfiability problem**) is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don't know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem is NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

We will soon be discussing more NP-Complete problems and their proof for NPCompleteness.

#### **References:**

MIT Video Lecture on Computational Complexity

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E.

### P, NP-Complete, NP, and NP-Hard (in Short)

NP problems have their own significance in programming, but the discussion becomes quite hot when we deal with differences between NP, P, NP-Complete and NP-hard.

**P**- Polynomial time solving: Problems which can be solved in polynomial time, which take time like  $O(n)$ ,  $O(n^2)$ ,  $O(n^3)$ . E.g.: finding maximum element in an array or to check whether a string is palindrome or not. So there are many problems which can be solved in polynomial time. **NP**- Non deterministic Polynomial time solving: Problem which can't be solved in polynomial time like TSP (travelling salesman problem) or an easy example of this is subset sum problem: given a set of numbers, does there exist a subset whose sum is zero? But NP problems are checkable in polynomial time means that given a solution of a problem, we can check that whether the solution is correct or not in polynomial time.

*Now we will discuss about NP-Complete and NP-hard.*

Take two problems A and B both are NP problems.

**Reducibility-** If we can convert one instance of a problem A into problem B (NP problem) then it means that A is reducible to B.

**NP-hard--** Now suppose we found that A is reducible to B, then it means that B is at least as hard as A.

**NP-Complete --** The group of problems which are both in NP and NP-hard are known as NPComplete problem.

Now suppose we have a NP-Complete problem R and it is reducible to Q then Q is at least as hard as R and since R is an NP-hard problem. Therefore Q will also be at least NP-hard, it may be NP-complete also.

### Approximation Algorithm

Given an optimization problem P, an algorithm A is said to be an approximation algorithm for P, if for any given instance I, it returns an approximate solution that is a feasible solution.

Approximation algorithms

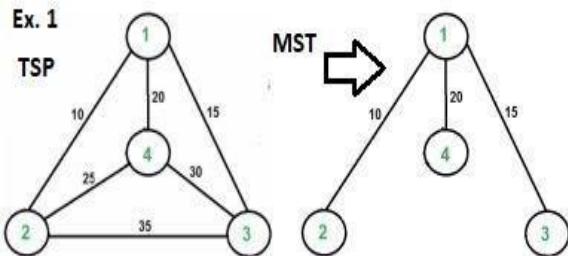
1. Guaranteed to run in polynomial time.
2. Guaranteed to get a solution which is close to the optimal solution.
3. Obstacle: need to prove a solution's value is close to optimum value, without even knowing what the optimum value is!

### Types of approximation

P -An optimization problem, A -An approximation algorithm, I -An instance of P,

$A^*(I)$  -Optimal value for the instance I,  $A(I)$  -Value for the instance I generated by A

1. **Absolute approximation:** A is an absolute approximation algorithm if there exists a constant k such that, for every instance I of P,  $A^*(I) - A(I) \leq k$ . Example: Planar graph coloring.
2. **Relative approximation:** A is an relative approximation algorithm if there exists a constant k such that, for every instance I of P,  $\max\{ A^*(I) / A(I), A(I) / A^*(I) \} \leq k$ . Example: Vertex cover.

**Ex. 1****TSP**

The full walk of above tree would be 1-2-1-4-1-3-1. Apply Hamiltonian cycle so that the output is 1-2-4-3-1  
 $\text{Cost} = 10 + 25 + 30 + 15 = 80$

In this case, the approximate algorithm produces the optimal tour, but it may not produce optimal tour in all cases.

A full walk lists all vertices when they are first visited in preorder, it also lists vertices when they are returned after a subtree is visited in preorder. The full walk of above tree would be 1-2-1-4-1-3-1.

Following are some important facts that prove the 2-approximateness.

- 1) The cost of best possible Travelling Salesman tour is never less than the cost of MST.  
 (The definition of MST says, it is a minimum cost tree that connects all vertices).
- 2) The total cost of full walk is at most twice the cost of MST (Every edge of MST is visited at most twice)
- 3) The output of the above algorithm is less than the cost of full walk. In above algorithm, we print preorder walk as output. In preorder walk, two or more edges of full walk are replaced with a single edge. For example, 2-1 and 1-4 are replaced by 1 edge 2-4. So if the graph follows triangle inequality, then this is always true.

From the above three statements, we can conclude that the cost of output produced by the approximate algorithm is never more than twice the cost of best possible solution.

**A Hamiltonian cycle**, also called a Hamiltonian circuit, Hamilton cycle, or Hamilton circuit, is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NPcomplete.

### Traveling-salesman problem

Imagine we are a salesman, and we need to visit  $n$  cities. We want to start a *tour* at a city and visit every city *exactly one time*, and finish the tour at the city from where we start. There is a non-negative cost  $c(i, j)$  to travel from city  $i$  to city  $j$ . The goal is to find a tour (which is a Hamiltonian cycle) of minimum cost. We assume every two cities are connected. Such problem is called *Traveling-salesman problem (TSP)*.

We can model the cities as a complete graph of  $n$  vertices, where each vertex represents a city.

It can be shown that TSP is NPC.

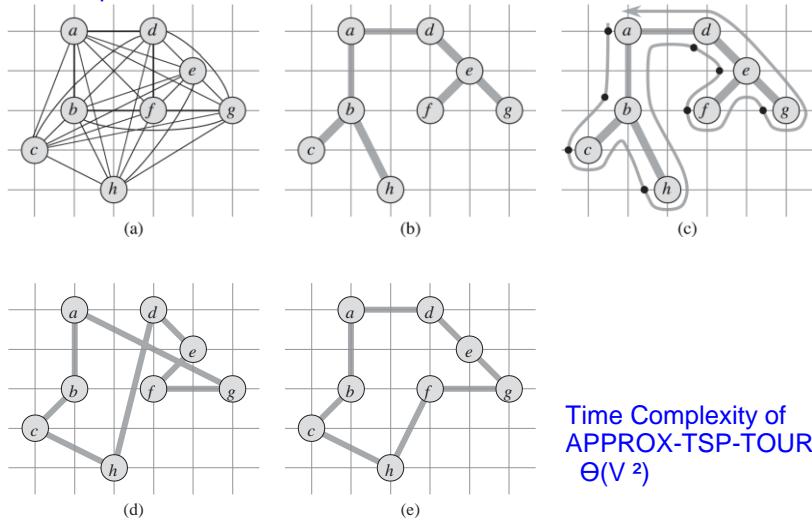
If we assume the cost function  $c$  satisfies the *triangle inequality*, then we can use the following approximate algorithm.

*Triangle inequality.* Let  $u, v, w$  be any three vertices, we have

$$c(u, w) \leq c(u, v) + c(v, w)$$

One important observation to develop an approximate solution is if we remove an edge from  $H^*$ , the tour becomes a *spanning tree*.

```
Approx-TSP ( $G = (V, E)$ ) {
1. compute a MST  $T$  of  $G$  ;
2. select any vertex  $r$  be the root of
   the tree ;
3. let  $L$  be the list of vertices
   visited in a preorder tree walk
   of  $T$  ;
4. return the hamiltonian cycle  $H$  that
   visits the vertices in the order  $L$  ;
}
```

**Example:2**

**Time Complexity of APPROX-TSP-TOUR**  
 $\Theta(V^2)$

**Figure 35.2** The operation of APPROX-TSP-TOUR. (a) A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary Euclidean distance. (b) A minimum spanning tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order. (c) A walk of T, starting at a. A full walk of the tree visits the vertices in the order a; b; c; b; h; b; a; d; e; f; g; e; d; a. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a; b; c; h; d; e; f; g. (d) A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074. (e) An optimal tour H for the original complete graph. Its total cost is approximately 14.715.

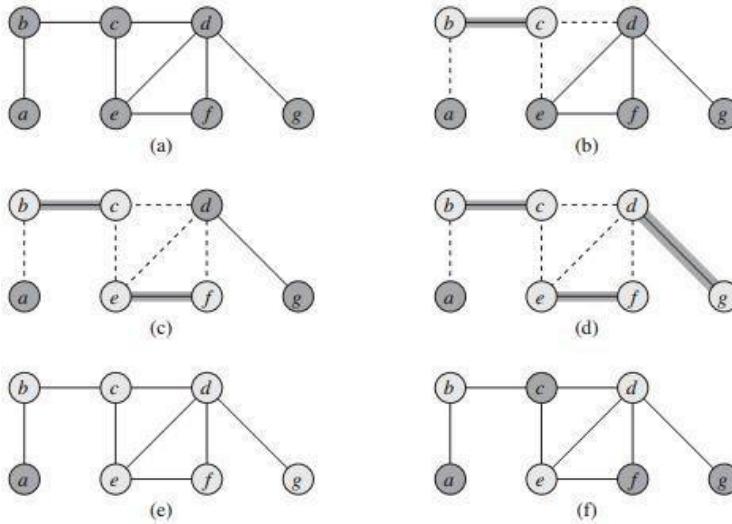
Recall from Section 12.1 that a preorder tree walk recursively visits every vertex in the tree, listing a vertex when it is first encountered, before visiting any of its children.

Figure 35.2 illustrates the operation of APPROX-TSP-TOUR. Part (a) of the figure shows a complete undirected graph, and part (b) shows the minimum spanning tree  $T$  grown from root vertex  $a$  by MST-PRIM. Part (c) shows how a preorder walk of  $T$  visits the vertices, and part (d) displays the corresponding tour, which is the tour returned by APPROX-TSP-TOUR. Part (e) displays an optimal tour, which is about 23% shorter.

### Vertex Cover Problem

**vertex cover** of an undirected graph  $G = (V, E)$  is a subset  $V' \subseteq V$  such that if  $(u, v)$  is an edge of  $G$ , then either  $u \in V'$  or  $v \in V'$  (or both). The size of a vertex cover is the number of vertices in it.

The **vertex-cover problem** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an **optimal vertex cover**. This problem is the optimization version of an NP-complete decision problem.



**Figure 35.1** The operation of APPROX-VERTEX-COVER. (a) The input graph  $G$ , which has 7 vertices and 8 edges. (b) The edge  $(b, c)$ , shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices  $b$  and  $c$ , shown lightly shaded, are added to the set  $C$  containing the vertex cover being created. Edges  $(a, b)$ ,  $(c, e)$ , and  $(c, d)$ , shown dashed, are removed since they are now covered by some vertex in  $C$ . (c) Edge  $(e, f)$  is chosen; vertices  $e$  and  $f$  are added to  $C$ . (d) Edge  $(d, g)$  is chosen; vertices  $d$  and  $g$  are added to  $C$ . (e) The set  $C$ , which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices  $b, c, d, e, f, g$ . (f) The optimal vertex cover for this problem contains only three vertices:  $b, d$ , and  $e$ .

APPROX-VERTEX-COVER( $G$ )

```

1   $C = \emptyset$ 
2   $E' = G.E$ 
3  while  $E' \neq \emptyset$ 
4      let  $(u, v)$  be an arbitrary edge of  $E'$ 
5       $C = C \cup \{u, v\}$ 
6      remove from  $E'$  every edge incident on either  $u$  or  $v$ 
7  return  $C$ 
```

endpoints  $u$  and  $v$  to  $C$ , and deletes all edges in  $E'$  that are covered by either  $u$  or  $v$ . Finally, line 7 returns the vertex cover  $C$ . The running time of this algorithm is  $O(V + E)$ , using adjacency lists to represent  $E'$ .

**Notes Prepared by  
Prof. Chandrakanta Mahanty**