|| Om Shri Ganeshay Namah ||

- What is algorithm ?

- An algorithm is a well-defined computational procedure that takes some value, or set of values, as inputs and produces some value, or set of values, as output.
- An algorithm is thus a sequence of computational steps that transform the input into output.
- It is a tool for solving a well specified computational problem.

Some examples :-

i) Sorting algorithms (used – E-commerce, ranking)
ii) Pagerank algorithm (which web-page to show)
iii) Routing iv) Resource allocation v) Convex hull

- What are the characteristic features of algorithms ?

i) Input (zero or more inputs) (fibonacci, pattern printing)
ii) Output (at least one output)
iii) Correctness :-

An algorithm is said to be correct if, for every input instance, it halts with correct output. A correct algorithm solves the given computational problem. An incorrect algorithm might not halt at all on some I/P instances, or it might halt with incorrect answer.

Insight : Incorrect algorithms can sometimes be useful, if we can control their error rate.

iv) Finiteness :-

The algorithm should end after a finite amount of time and it should have a limited/finite no. of instructions. A finite algorithm ensures that it will eventually stop executing and produce a result (expected o/p or a response that no solution is possible).

e.g that cause infiniteness —

1. infinite loops
2. Recursive functions with no base conditions

v) Definiteness :-

All instructions in the algorithm must be unambiguous, precise and easy to interpret. By referring to any of the instruction in the algorithm one can clearly understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity.

vi) Effectiveness :-

An algorithm must be developed by using very basic, simple and feasible operations so that one can trace it out using just pencil & paper. All the steps that are required to get the output must be feasible with available resources. (Hardware)

vii) Efficiency :-

Algorithm must make efficient use of resources (memory and Time).

- Space complexity
- Time complexity

Other characteristics —

viii) Language independent

Since algorithm should be language independent, then how to express algorithms ?

Ans — Pseudocodes

- What is Pseudocode ?
- What is not algorithm ? — Mechanism, method

- Pseudocode : Pseudo : pretentious

In pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm (sometimes English).
- Pseudocode is not concerned with issues of S/W engg (for e.g - data abstraction, error handling, modularity)

Conventions of Pseudocode :-

i) Indentation indicates block structure.

ii) Looping constructs & conditional constructs have interpretations similar to those in C, C++, Java etc.. loop counter retains the value after exiting the loop. (To & downto for for loop, by for step)

iii) // — comment

iv) i = j = e equivalent as i = e; j = e; / j = e; i = j;

v) Variables are local to given procedure. We shall not use global variables without explicit indication.

vi) Compound data is organized into objects, which are composed of attributes. To access data, we use object name . attribute. A . length

vii) Parameters to a procedure are passed by value

When value is passed :-

a = b — it is not reflected to calling function

When object is passed (eg - array)

A[10].value = b — it is visible to calling procedure

viii) Return statement immediately transfers control back to point of calling. It may return multiple values at once.

ix) Boolean operators "and" and "or" are short circuiting

x AND y — y will be only evaluated if x is TRUE

x OR y — y will only be evaluated if x is FALSE

x) Keyword error represents error. No need for error handling in pseudocode.

Examples -

① Maximum of n numbers

```
max = a1                          a[1]
for t = 2 to n
    if at > max then              a[t]
        set max = at              a[t]
    endif
endfor
```

(2) n^{th} fibonacci number

```
func" fibonacci (n)
  if n<=0
    return "invalid I/P"
  else if n=1
     return 0
  else if n=2
     return 1
  else
  return fibonacci (n-1) + fibonacci (n-2)
```

```
if n<=0
   return "invalid I/P"
else if n=1
     return 0
elseif n=2
     return 1
else fib_0 = 0
     fib_1 = 1
     fib_n = 0
for i=3 ton
  fib_n = fib_0
        + fib_1
  fib_0 = fib_1
  fib_1 = fib_n
return fib_n
```

(3) factorial of a no. $0! = 1$

```
if n<0
   return "invalid I/P"
else
   result = 1
   for i=1 to n
      result = result * i
   return result
```

```
func" fact (n)
 if n<0
    return "invalid =I/P"
 else if n=0
     return 1
 else
     return n* fact(n-1).
```

(4) Linear search

```
i = NIL
for j = 1 to A.length do
   if A[j] = v then
      i = j
      return i
   endif
endfor
return i
```

- Key characteristics of Pseudocode -

1. Informality.
2. Abstraction
3. Clarity
4. Flexibility.
5. Use of keywords & constructs
6. Indentation

Basic aspects of Algorithm

Correctness, Design and Analysis

Example 1 - Stable matching problem

Computational Tractability

How resource requirements (time & space) scale with I/P size ?

Efficiency :-

An algorithm is efficient if, when implemented, it runs quickly on real input instances.

This definition is incomplete as it does not answers — where & how well

e.g Bad algos run quickly for small I/P on a fast processor

Good algos run slowly when they are sloppily coded.

Real I/P meaning — 100, 10^6, 10^9 — what ?

So, above definition of efficiency does not consider - how well or badly, algorithm may scale as problem size grows big to unexpected levels.

Efficiency — platform independent
instance independent
of predictive value w.r.t. increasing I/P size

e.g stable matching
No. of men = no. of women = n
I/P preference list of $2n$ people having n preferences
So, I/P size $2n^2$.

- Worst case running times and Brute force :-

Worst case - bound on the largest possible running time the algo could have over all I/Ps of a given size N, and how this scales with N.

Average case — random cases
good for getting insights
but often unaguine.

Ques - what is a reasonable analytical benchmark that can tell us whether a running-time bound is impressive or weak?

Ans. Comparison with brute-force search over the search space of possible soln.

Stable matching Example (using Brute force)
Even when size of I/P instance is relatively small, the search space it defines is enormous

Search space — n! possible perfect matching b/w n men & n women

Matching must be stable

m_1 w_1
m_2 w_2
m_2 w_3
2!

possible perfect matchings
(m_1, w_1) (m_2, w_2) (m_3, w_3)
(m_1, w_1) (m_2, w_3) (m_3, w_2)
(m_1, w_2) (m_2, w_1) (m_3, w_3)
(m_1, w_2) (m_2, w_3) (m_3, w_1)
(m_1, w_3) (m_2, w_2) (m_3, w_1)
(m_1, w_3) (m_2, w_1) (m_3, w_2)

Brute force will search for all possible combinations for stability.

Efficiency :-

An algorithm is efficient if it achieves qualitatively better worst case performance, at an analytical level, than brute force search.

Algorithms having better efficiency than brute force nearly always contain a valuable heuristics data that make them work; as they tell about intrinsic structure and computational tractability of the underlying problem.

- Polynomial Time as a definition of efficiency :-

Discussion of functions - n, $\lg n$, n^2, n^3, $n \log n$, 2^n, $n!$

Formulation of scaling behavior.

Suppose an algorithm has following property -

There are absolute constants $c > 0$ & $d > 0$ so that on every I/P instance of size N, its running time is ~~at most proportional to N^d~~
→ bounded by cN^d primitive computational steps.
i.e. running time is at most proportional to N^d.

for a model where each step corresponds to a single assembly language instruction of a standard processor or a one line of Std. programming language C++, Java etc. In such cases, we say, if this running time bound holds, for some c & d, then we say that the algorithm has a polynomial running time or, it is a polynomial time algorithm.

→ It follows scaling property

for I/P size N cN^d
I/P size 2N $c(2N)^d = c\,2^d N^d$
slow down by a factor of 2^d.

• Asymptotic Lower Bound :- (Best case)

$T(n) = \Omega(f(n))$ $\exists\ \epsilon > 0, n_0 \geq 0$ $n \geq n_0$

$T(n) \geq \epsilon f(n)$

ϵ - fixed, independent of n

O - Landau's symbol
Ω -
θ -

Eg $T(n) = pn^2 + qn + r$

$T(n) = pn^2 + qn + r \geq pn^2$

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

Problems ① $T(n) \leq (p+q+r)n^2 \leq (p+q+r)n^3$

② $T(n) \geq pn^2 \geq pn \qquad n \geq 1$

~~Solve~~ Asymptotically Tight Bounds

$T(n) = \theta(f(n))$: $T(n)$ is asymptotically tight bounded by θ.

∴ for above example

$T(n) = pn^2 + qn + r - \theta(n^2)$

If $T(n) = O(f(n))$ and

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

i.e. $T(n)$ is both upper & lower bounded by $f(n)$. we call it tight bound.

Ex ① int a, [10], find max element

{

a = max(a[10])

}

max {

≡

}

Minmax array

if i%2 == 0

find max ( )

else

swap ( )

Complexity statements

if. else

loop

nested Loops

~~loop~~ functions/ procedures

- Properties of Asymptotic Growth Rate :-

① Transitivity :-

i) If $f = O(g)$ and $g = O(h)$, then $f = O(h)$

ii) If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$

iii) If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$

Proof

$f(n) = c\, g(n)$ — $g(n) = c'\, h(n)$ ⇒

$f(n) = cc'\, h(n)$ $\quad cc' > 0$

$f(n) = c''\, h(n)$

$f(n) = O\, h(n)$

② Sum of functions :-

(i) $f = O(h)$ and $g = O(h)$

$f + g = O(h)$

Proof

$f \le c \cdot h(n)$ $\quad g \le c'\, h(n)$ $\quad n \ge n_0$

$f + g \le c \cdot h(n) + c'\, h(n)$

$\le (c + c')\, h(n) \le c''\, h(n)$

ii) Let $f_1, f_2 \cdots f_k$ & h be functions

s.t. — $f_i = O(h)$

then, $f_1 + f_2 + f_3 + \cdots f_k = O(h)$

iii) Suppose f and g are two functions such that $g = O(f)$. Then $f + g = \Theta(f)$ i.e. f is asymptotically tight bound for combined function $f + g$.

- Asymptotic bounds for some common functions :-

① Polynomials.

Let f be a polynomial of degree d, in w/c the coefficient of a_d is positive. Then

$f = O(n^d)$

$f = a_0 + a_1 n + a_2 n^2 + \cdots a_d n^d$

s.t. $a_d > 0$

$d \ge 0$

If $f = O(n^d)$

$f = \Omega(n^d)$

then $f = \Theta(n^d)$

Also

$n^{1.5}$

$n \log n$

$\log n$

② Logarithms — slowly growing functions

Properties of Asymptotic Example

		Reflexive	Symmetric	Transitive
Big (O)	$f(n) \leq c \cdot g(n)$	✓	×	✓
Ω	$f(n) \geq c \cdot g(n)$	✓	×	✓
Θ	$c \cdot g(n) \leq f(n) \leq c' g(n)$	✓	✓	✓

Practice Question

i) $f(n) = n \log n \quad g(n) = \frac{n\sqrt{n}}{2}$

ii) $f(n) = 2 (\log n)^2 \quad g(n) = \log n + 1$

True / False

i) $2n^2 + 1 = O(n^2)$

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

iii) $n^2 (1 + \sqrt{n}) = O(n^2 \log n)$

iv) $3n^2 + \sqrt{n} = O(n + n\sqrt{n} + \sqrt{n})$

Some other

To Prove

i) $f_1(n) = \Omega(g_1(n))$

$f_2(n) = \Omega(g_2(n))$

Then - $f_1(n) + f_2(n) = \Omega(g_1(n) + g_2(n))$

ii) Also - $f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$

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

$T(n) = O(n^2)$ & $g(n) = O(n^3)$

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

Ques Problem P I/P size n

3 algos
- A — 5 SP of size n/2 + n
- B — 2 SP of size n-1 + c
- C — 9 SP size $\frac{n}{3}$ + n^2

Compare

Recurrences

Recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs.

Recurrence go hand-in-hand with divide & conquer approach.

Recurrences can take many form.

- It might divide subproblems into equal parts.

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

- or unequal parts

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

- can have one less element than original problem

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

- **Methods for solving Recurrences :-**
  - i) Iteration / Substitution (Brute force method) Back substitution
  - ~~ii)~~ Substitution (By guessing)
  - iii) Recursion Tree
  - iv) Master method

i) Substitution Method for Solving recurrence :-

It takes two steps -

1. Guess the form of solution
2. Use mathematical induction to find the constants and show that the solution works

This method can be used to establish either upper or lower bounds on a recurrence.

① Example $T(n) = 2T\left(\lfloor \frac{n}{2} \rfloor\right) + n$ —①

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

Guess – Let the soln be $O(n \lg n)$

To prove $T(n) \leq c \cdot n \lg n$ for an appropriate choice of constant $c>0$

Let this bound holds for all positive $m<n$. st for $m = \lfloor \frac{n}{2} \rfloor$, yielding

$$T\left(\lfloor \tfrac{n}{2} \rfloor\right) \le c \lfloor \tfrac{n}{2} \rfloor \lg \lfloor \tfrac{n}{2} \rfloor$$ — substitute this to $T(n)$

$$T(n) \le 2\left(c \lfloor \tfrac{n}{2} \rfloor \lg \lfloor \tfrac{n}{2} \rfloor\right) + n \qquad \lg = \log_2$$

$$\le cn \log\left(\tfrac{n}{2}\right) + n$$

$$= cn \log n - cn \log 2 + n$$

$$= cn \lg n - cn + n$$

$$T(n) \le cn \log n \qquad \text{for } c \ge 1$$

Soln-2 $\quad T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$ — (2)

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$$ — (3)

Substitution (2) in (1)

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

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

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

Substitution (3) in (4)

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

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

$$\vdots$$

$$2^4 T\left(\frac{n}{2^4}\right) + 4n, \quad 2^5 T\left(\frac{n}{2^5}\right) + 5n$$

$$\vdots \text{ k times}$$

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

$$\Rightarrow \log n = k$$

$$\Rightarrow n\,T(1) + \log n * n \Rightarrow n + n \log n \Rightarrow O(n \log n)$$

Advantages:-

i) Solves any type of recurrence relation

ii) Always gives the correct answer

Disadvantages:-

i) Slow method due to large no. of mathematical calculations

② eg $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ n * T(n-1) & \text{if } n > 1 \end{cases}$ factorial

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

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

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

$$T(n-2) = (n-2) * T(n-3) \quad \text{——— (3)}$$

Substitution (2) in (1)

$$T(n) = n * (n-1) * T(n-2) \quad \text{——— (4)}$$

Substitution (3) in (4)

$$T(n) = n * (n-1)(n-2)\, T(n-2)$$

$\vdots$ till $(n-1)$ steps

$$T(n) = n * (n-1)(n-2)(n-3) \cdots T(n-(n-1))$$

$$= n * (n-1)(n-2)(n-3) \cdots T(1)$$

$$T(n) = n * (n-1)(n-2) \cdots 3 \cdot 2 \cdot 1$$

$$T(n) = n * n\left(1-\frac{1}{n}\right)n\left(1-\frac{2}{n}\right)n\left(1-\frac{3}{n}\right) \cdots n\left(\frac{3}{n}\right) n\left(\frac{2}{n}\right) \cdot n\left(\frac{1}{n}\right)$$

~~$T(n) = n^n * \text{constants}$~~

$T(n) = O(n^n)$ — factorial complexity

(3) $T(n) = \begin{cases} 1 & , \text{if } n=1 \\ T(n-1) + \log n & , \text{if } n>1 \end{cases}$ ——— (1)

$T(n-1) = T(n-2) + \log(n-1)$ ——— (2)

$T(n-2) = T(n-3) + \log(n-2)$ ——— (3)

Substituting (2) in (1)

$T(n) = T(n-2) + \log n + \log(n-1)$ ——— (4)

Substituting (4) in (3)

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

$= T(n-4) + \log(n-3) + \log(n-2) + \log(n-1) + \log n$

⋮ k Times

$T(n) = T(n-k) + \log(n-(k-1)) + \log(n-(k-2)) + \log(n-(k-3)) + \ldots \log n$

Let $n-k=1$

~~n=k+1~~ $n = k+1$

$T(n) = T(1) + \log(2) + \log(3) + \log(4) + \ldots \log(n)$

$= 1 + \log 1 + \log 2 + \log 3 + \ldots \log n$

$= 1 + \log(1 \cdot 2 \cdot 3 \cdot \ldots n)$

$= 1 + \log(n!)$

$= 1 + \log n^n$

$= 1 + n \log n$

$T(n) = O(n \log n)$

$\log m + \log n = \log(m \times n)$

$n! = n * (n-1) * (n-2) * \ldots$

$n! \simeq n^n$

$n-k=1$

$n=1+k$ $T(n) = T(1) + \log(n \cdot$

$\frac{n-k+1}{1+1} \cdot$

$n! = n^n$

H.W ① $T(n) = 8T\left(\frac{n}{2}\right) + n^2$ — $O(n^3)$

② $T(n) = \begin{cases} T(n/2) + c & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$ — $O(\log_2 n)$

— Binary search

③ $T(n) = n^3 + 2\left(T\left(\frac{n}{2}\right)\right)$, $T(1) = \theta(1)$ — $O(n^3)$

④ $T(1) = \theta(1)$

$T(n) = c + 2(T(n-1))$ — $\theta(2^n)$

⑤ Binary search

Pseudocode -

```
BS(A, i, j, x)
    mid = (i+j)/2
    if mid == x
        return x
    else
        if mid >= x
            BS(A, mid+1, j, x)
        else
            BS(A, i, mid-1, x)
```

Recurrence relation

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

Soln

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

$$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + c$$

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

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

$$\vdots$$

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

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

$n = 2^k$

$\log n = \log_2 2^k = k \log$

$T(n) = T\left(\frac{n}{n}\right) + \log n \cdot c$ $\quad \log_n = k$

$= T(1) + c \cdot \log n$

$= 1 + c \cdot \log n$

$T(n) = O(\log n)$

- **Master Method :-** (Alternate Method)

Pros - faster method

cons - cannot be used for all kinds of recurrence problems.

Only solves problems of type -

$$\boxed{T(n) = aT\left(\frac{n}{b}\right) + f(n)} \qquad a \geq 1, \; b > 1$$

applicability

$T(n) = T(n-1) + 1$ ✗

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

$T(n) = T\left(\frac{n}{2}\right) + c$ ✓

Soln

$$\boxed{T(n) = n^{\log_b a}[U(n)]}$$

- $U(n)$ depends on $h(n)$

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

Based on the value of $h(n)$

i) If $h(n) = n^r$, $r > 0 \Rightarrow U(n) = O(n^r)$

ii) If $h(n) = n^r$, $r < 0 \Rightarrow U(n) = O(1)$

iii) If $h(n) = (\log n)^i$, $i \geq 0 \Rightarrow U(n) = \frac{(\log_2 n)^{i+1}}{i+1}$

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

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

$$u(n) = \frac{f(n)}{n^{\log_b a}} = \frac{n^2}{n^{\log_2 8}} = \frac{n^2}{n^3} = \frac{1}{n} = n^{-1}$$

$u(n) = O(1)$

Soln i

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

$$= n^3 \cdot O(1)$$

$$= n^3 \cdot 1 = O(n^3)$$

Alternate

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

$$= n^{\log_2 8} \cdot U(n)$$

$$= n^3 \cdot U(n)$$

$$h(n) = \frac{f(n)}{n^{\log_b a}} = \frac{n^2}{n^{\log_2 8}} = \frac{n^2}{n^3} = \frac{1}{n} = n^{-1}$$

$h(n) = n^{-1} \Rightarrow r < 0, \quad U(n) = O(1)$

$$T(n) = n^3 \cdot 1 = n^3$$

$$\boxed{T(n) = O(n^3)}$$ Ans.

Ques 2. $T(n) = T\left(\frac{n}{2}\right) + c$

$a = 1$

$b = 2$

$f(n) = c$

Soln - $T(n) = n^{\log_2 1} \cdot U(n)$

$= n^0 \cdot U(n) = 1 \cdot U(n)$

$$h(n) = \frac{f(n)}{n^{\log_b a}} = \frac{c}{n^{\log_2 1}} = \frac{c}{1} = c$$

Case 3

$$h(n) = (\log_2 n)^0 \cdot c$$

$$U(n) = \frac{(\log_2 n)^{0+1}}{0+1} \cdot c = c \frac{\log_2 n}{1} = c \log_2 n$$

$$T(n) = 1 \cdot \log_2 n \cdot c = c \cdot \log_2 n$$

$$\boxed{T(n) = O(\log_2 n)}$$

Ques Example 3 $\quad T(n) = \begin{cases} T(\sqrt{n}) + \log n & \text{if } n \geq 2 \\ O(1) & \text{else} \end{cases}$

Let $n = 2^m$

$$T(2^m) = T(2^{m/2}) + \log 2^m$$

$$= T(2^{m/2}) + m \log_2 2$$

$$T(2^m) = T(2^{m/2}) + m$$

Let $T(2^m) = S(m)$

$$S(m) = S\left(\frac{m}{2}\right) + m$$

Now, $a = 1$, $b = 2$, $f(n) = m$

$$S(m) = m^{\log_2 1} \cdot U(m) = m^0 \cdot U(m) = U(m)$$

$$h(n) = \frac{m}{m^{\log_2 1}} = \frac{m}{1} = m$$

$$U(m) = O(m)$$

$$S(m) = O(m)$$

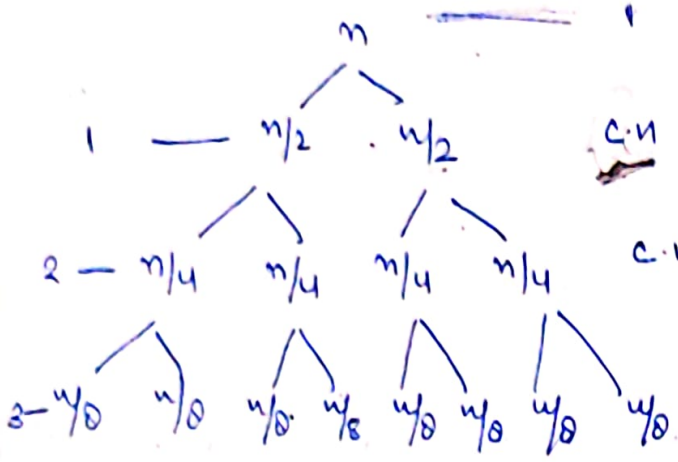
~~$T(2^m) = O$~~

$\log n = m$

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

- **Recursive Tree :-**

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



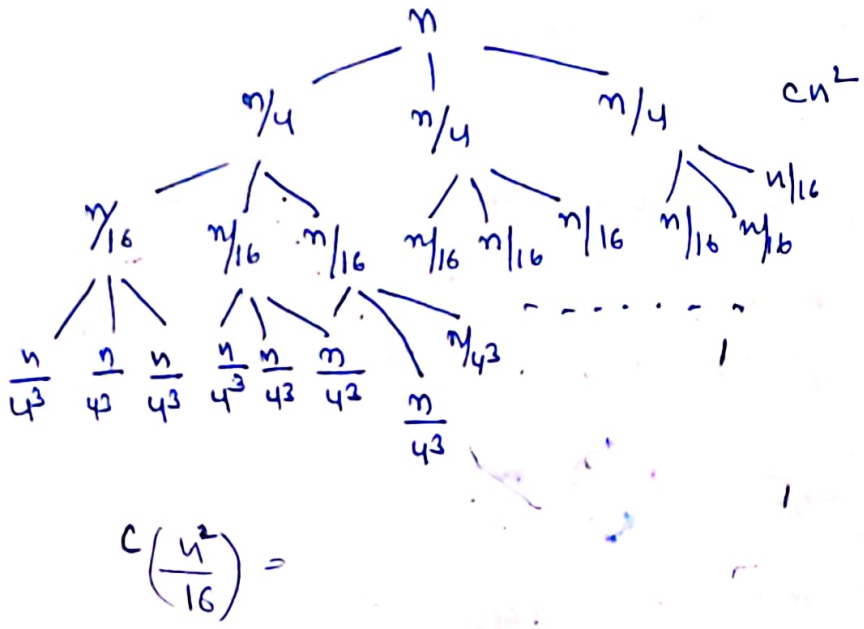
~~for~~ 2 Cost is depending on height of tree.

∴ Total cost = $\log n \cdot c \cdot n$ Height of Binary Tree $\log_2 n$

$= c \cdot n \log n$

$= O(n \cdot \log n)$

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



$3\left(\frac{n^2}{16}\right) = 3 \cdot \frac{n^2}{4^2} \cdot c$

$9 \cdot \left(\frac{n}{4^2}\right)^2 = 9 \cdot \frac{n^2}{4^4} \cdot 1$

$= \left(\frac{3}{16}\right)^2 n^2$

$\left(\frac{3}{16}\right)^k n^2 \cdot c$

$c\left(\frac{n^2}{16}\right) =$

$cn^2 + \frac{3}{16}cn^2 + c\left(\frac{3}{16}\right)^2 n^2 + c\left(\frac{3}{16}\right)^3 n^2$

$cn^2\left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \cdots\right)$ G.P. series

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

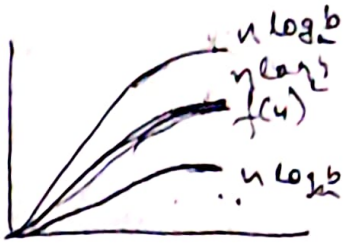
$= cn^2 \frac{1}{\frac{13}{16}}$

$= \frac{16}{13} cn^2$

$= O(n^2)$

$1 + r + r^2 + r^3 \cdots$

$\frac{1}{1-r} \quad r < 1$



Master Method (Contd.)

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

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

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

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

① If $f(n) = O(n^{\log_b a - \epsilon})$ for some positive constant $\epsilon > 0$

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

② If $f(n) = \Theta(n^{\log_b a} \log^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$ $k \geq 0$

③ If $f(n) = \Omega(n^{\log_b a + \epsilon})$ & $af\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for $c < 1$

$T(n) = \Theta(f(n))$

Regularity cond^n

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

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

$n^{\log_b a} = n^{\log_3 9} = n^2$

$f(n) = O(n^{2-\epsilon}) \quad \epsilon = 1$

$\therefore$ Case 1 is applied

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

Ans.

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

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

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

$Q(y) \leq 2Q\left(\frac{y}{4}\right) + y^{0.5}$

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

Ex. 2 $T(n) = T\left(\frac{2n}{3}\right) + 1$

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

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

$f(n) = O(n^{\log_b a}) = O(n^0) = O(1) = 1$

Case 2 – $T(n) = \Theta(\lg n)$

Ex.3 $T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$

$a=3,\ b=4,\ f(n) = n \lg n$

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

$f(n) = \Omega\left(n^{\log 0.793 + 0.2}\right)^{\epsilon}$ [Case 3]

and

$a \cdot f\left(\frac{n}{b}\right) \le c \cdot f(n)$

$3 \cdot f\left(\frac{n}{4}\right) \lg\left(\frac{n}{4}\right) \le c \cdot f(n)$ for $c = \frac{3}{4}$

Case 3

$T(n) = \Theta(n \lg n)$

Ex.4 $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

$a=2,\ b=2,\ f(n) = \Theta(n)$

$n^{\log_b a} = n^{\log_2 2} = n^1$

$\because f(n) = \Theta(n^{\log_a b}) = \Theta(n)$ [Case 2]

$\therefore T(n) = \Theta(n \lg n)$.

Ex.5 $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$

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

$n^{\log_a b} = n^{\log_2 8} = n^3$

$\because f(n) = O(n^{3-1})$ $\epsilon = 1$ [Case 1]

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

$2.8 < \log_2 7 < 2.81$

Ex.6 $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$

$a=7,\ b=2,\ f(n) = \Theta(n^2)$

$n^{\log_b a} = n^{\log_2 7} = n^{2.00}$

$\because f(n) = O(n^{2.8 - 0.8})$ [Case 1]

$\therefore T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7}) = \Theta(n^{2.8})$

Priority Queue Implementation using Heap Data Structures

Goal of this subject : approach towards better soln than brute force (polynomially solvable approach)

Why are we studying priority queues — useful while implementing google algorithms.

Priority queue implemnt. using arrays = O(n)

" " " " heaps = O(log n)

Some applications may have priorities - for e.g — stable matching

---

In other copy

- Some representative problems –

1. Stable Matching Problem :- David Gale – 1962
Lloyd Shapley
(Two mathematical economists)

Ques – could one design a college admission process, or a job recruiting process, that was self-enforcing?

Each student has a preference order for companies
Applicants choose which offer to accept

Each company make a preference list as per the applications received and then extend offers

Problems observed by Gale and Shapley :-

(1) S_1 accepts offer by C_1. Later C_2 extends offer to S_1. S_1 chooses C_2 & declines C_1. C_1 will now extend offer to waitlisted S_2. S_2 accepts offer & declines its previously accepted offer by C_3. This goes on.
Ques – ~~but~~ How do we get the stability?

(2) If S_4 destined to go to C_3 calls C_2 & say I want to join you & C_2 also find S_4 suitable. So, if C_2 is less scrupulous company, then it will retract offer from S_1 & give it to S_4 instead.
Ques – How do control this chaos?

Outcomes of chaos –
i) People not happy
ii) Right people at wrong company & vice-versa

System is not self-enforcing (means people are allowed to

act in their self interest, thus making breaking down

One solution –

① S_1 offered at C_1 calls C_2 for offer, but C_2 declines saying we prefer our selected candidates.

② C_1 calls S_1 offered at C_2 and offers him, S_1 declines saying I am happy where I am.

Here, self-interest itself prevents offers from being retracted or redirected. So, it gives stable outcomes.

Problem Statement by Gale & Shapley :-

Given a set of preferences among employers and applicants, can we assign applicants to employers so that for every employer E, and every applicant A who is not scheduled to work for E, at least one of the following two things is the case ?

i) E prefers every one of its accepted applicants to A ; or

ii) A prefers her current situations over working for employer E.

If this holds, the outcome is stable.

Similar problems : National Resident matching program (match residents to hospitals)

Problem formulation :

Problem has various distracting asymmetries like

- Each applicant is looking for single company but company looks for multiple applicants
- There may be more applicants than no. of available internship slots.
- Each applicant usually does not applies for every company.

Problem Simplification :-

Each of n applicant applies to each of n companies and each company wants to accept a single applicant.

Problem Resemblance :- Stable marriage

n men n women

want to marry & stable union

Problem statement for stable marriage :-

Let us consider

Let $M = \{m_1, m_2 \ldots m_n\}$ of n men & $W = \{w_1, w_2 \ldots w_n\}$ of n women. Let $M \times W$ denotes set of all ordered pairs of the form (m, w). A matching S is a set of ordered pairs, each from $M \times W$, with the property that each member of M and each member of W appears in atmost one pair in S. A perfect matching S' is a matching with the property that each member of M and each member of W appears in exactly one pair in S'.

Perfectly matching - Each person is married (nobody is unmarried, nobody is in polygamy)

Setting up the preferences - Each man ranks women as per his preference. Similarly each women also ranks men as per her preference.

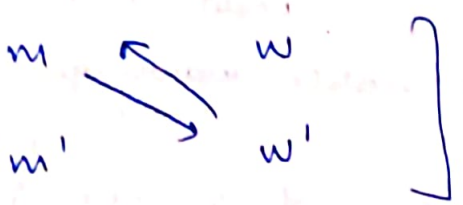
m 'prefers w to w' if w ranks higher than w'

Ranks are increasing / decreasing but not non descending or non ascending.

Now, if S is perfect matching. What problems may arise ?

1. Two pairs (m, w) & (m', w') in S are there st m prefers w' to w & w' prefers m to m'.

So



Marriage is unstable

(m, w') (m', w) — not present in S. So unstable matching

Goal of the problem : set of marriages with no instability

A matching S is stable if -

i) It is perfect matching

ii) There is no instability w.r.t. S.

Two questions arise -

1. Does there exist stable matching for every set of preference list ?
2. Given a set of preference lists, can we efficiently construct a stable matching if there is one ?.

e.g

$M = \{m, m'\}$ $W = \{w, w'\}$

Preference list

m prefers w to w'
m' prefers w to w'
w prefers m to m'
w' prefers m to m'

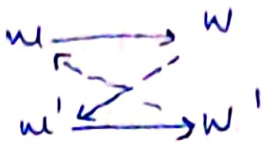
There is complete agreement.

(m, w) (m', w')

Unstable if (m', w) & (m, w')

Other case

m prefers w to w'
m' prefers w' to w
w prefers m' to m
w' prefers m to m'



(m,w) (m', w') – stable : men are happy

(m',w) (m,w') – also stable as women are happy

Designing the Algorithm for stable matching

Assumption –

- Initially everyone is ~~~~ unmarried
- Introducing an intermediate state - engagement
  i.e. m proposes w & w accepts will not mean (m,w) should be a pair as in future she may accept proposal ranked higher to m.

- suppose, ~~now in some state~~ after a period of Time, some men & some women are not paired
  m proposes to w
  if w is free - she accepts - engagement
  if w is not free - she declines

- Finally, algo will terminate if there is no free men or women. At this Time, all engagements are made final and we get perfect matching

Algorithm

Initially all $m \in M$ and $w \in W$ are free

while (there is a man m who is free & hasn't proposed to every woman)

choose such a man m

let w be the highest ranked woman in m's preference list to whom m has not yet proposed

if w is free then

(m, w) become engaged

else w is currently engaged to m'

if w prefers m' to m then

m remains free

else w prefers m to m' then

(m, w) become engaged

m' becomes free

end if

end if

end while

return the set S of engaged pairs

Analyzing the Algorithms :— Algo will terminate (upper bound) — correctness of stable matching

(1) From woman perspective.

(1.1) w remains engaged from the point at which she receives her first proposal; and the sequence of partners to which she is engaged gets better and better (in terms of her preference list)

(2) From man perspective -

1.2 The sequence of women to whom m proposes gets worse and worse (in terms of his preference list)

Now we show that algo will terminate, and give a bound on the max^m. no. of iterations needed for termination

1.3 "The Gale-Shapley algorithm terminates after at most n^2 iterations of the while loop."

Proof -

The strategy for upper bounding the running time of an algorithm we will use here is to find a measure of progress.

In this algorithm, some man proposing (for the only time) to a woman he has never proposed to before. So if let $P(t)$ denote the set of pair (m, w) s.t. m has proposed to w by the end of iteration t, we see that for all t, the size $P(t+1)$ is strictly greater than the size of $P(t)$. But there are only n^2 possible pair of men & women in total, so the value of $P(\cdot)$ can increase at most n^2 times over the course of algorithm. It follows that there can be at most n^2 iterations.

⇒ Two worth noting points here -

i) There are executions of the algorithm (with certain preference lists) that can involve close to n^2 iterations, so this analysis is not far from the best possible.

ii) There are many quantities that would not have worked well as a progress measure for the algorithm, since they need not strictly increase in each iteration.

Proof for correctness -

(1.4) i) If m is free at some point in the execution of the algorithm, then there is a woman to whom m has not yet proposed.

ii) Proof - (Proof by contradiction)
Suppose there comes a point when m is free but has already proposed to every woman. Then by (1.1) each of n women is engaged at this point in time. Since the set of engaged pairs forms a matching, there must also be n engaged men at this point in time. But there are only n men total, and m is not engaged. So this is a contradiction.

(1.5) The set S returned at termination is a perfect matching.

Proof.
The set of engaged pair always forms a matching. Let us suppose that algorithm terminates with a free man m. At termination, it must be the case that m had already proposed to every woman, for otherwise while loop could not have exited. But this contradicts (1.4), which says that there cannot be a free man who has proposed to every woman.

1.6 Consider an execution of the G-S algorithm that returns a set of pairs S. The set S is a stable matching.

Proof

From (1.5) S is a perfect matching.

Now, to prove S is stable matching, we will assume that there is an instability w.r.t. S & obtain a contradiction.

Now, such an instability would involve two pairs (m, w) & (m', w') in S with properties that -

- m prefers w' to w &
- w' prefers m to m'

In execution of the algorithm that produced S, m's last proposal was to w. Now we ask, Did he propose to w' at some earlier point in this execution? If he didn't then w must occur higher on m's preference list than w', contradicting the assumption that m prefers w' to w. If he did, he was rejected by w' for some other man m'' whom w' prefers to m. m' is the final partner of w', so either m'' = m' or by (1.1) w' prefers her final partner m' to m''. So either way it contradicts the assumption that w' prefers m to m'.

It follows a stable matching.

- Other extensions

i) women also propose ii) People proposing are given weights (influential will first select)

Discussion

① Earlier system was favouring to men. If each man in exactly first pass gets woman he desires (best case) — choice of woman does not matter here. — unfairness in G-S algorithm.
So if women get worst of their preference, they will end up unhappy.

So, this simple set of preference list compactly summarizes a world in w/c someone is destined to end up unhappy : women are unhappy if men propose, and men are unhappy if women propose.
So, algorithm (G-S) is unfair.

more situations — Initially n men, n women. At iii) some intermediate state, a women died. If this man is influential, we will get one & least influential get zero.

~~xx xx xx~~

② iv) Preference order changes
v) Mixed proposals

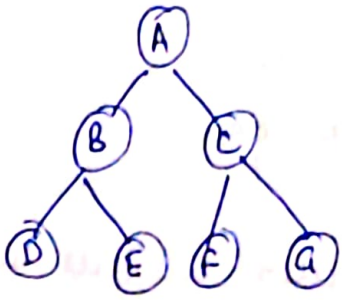
- 5 Representation Problems :-

i) Interval scheduling
ii) weighted interval scheduling
iii) Bipartite matching
} Efficiently solvable by a sequence of increasingly subtle algorithmic techniques

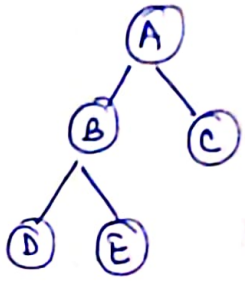
iv) Independent set — problem believed to be unsolvable by any efficient algorithm
v) Competitive facility location — class of problems that is still harder

Priority Queue Implementation using Heaps

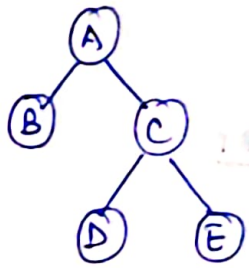
- Binary Tree Representation using Arrays



A	B	C	D	E	F	G



A	B	C	D	E

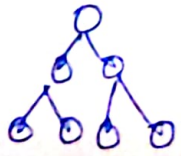


A	B	C	-	-	D	E

Relationship — Parent Node — child
left Right

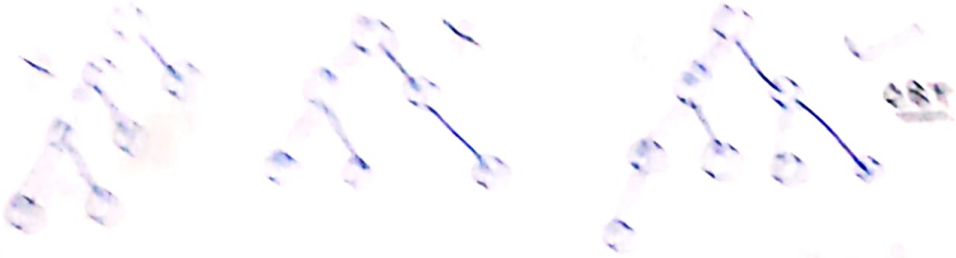
Node — i
left child — $2i$
Right child — $2i+1$
Parent — $\lfloor \frac{i}{2} \rfloor$

→ Full Binary Tree



— If we add one element, height will increase

— If we insert new element, height will not change as there is space for new element.



Height of complete binary tree = $\log n$

- **Heaps**

→ Heap is a complete binary tree.

— Every element is greater (or equal) to its children

Max heap

— every element is less (or equal) to its children

Min heap

- Insert Operation in Max heap.

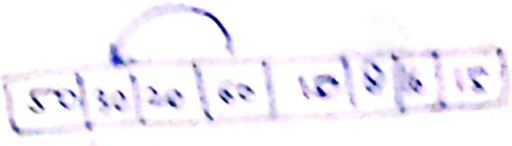
⇒ push insert at root position

Insert 60

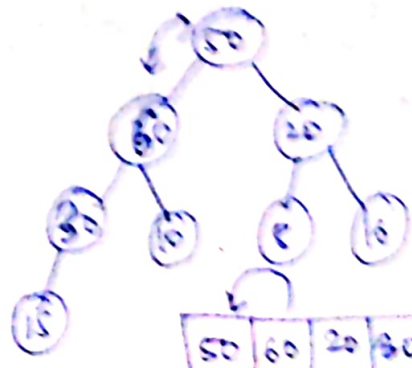
condition violated of heap

swap for position

$\lfloor \frac{8}{2} \rfloor = 4$

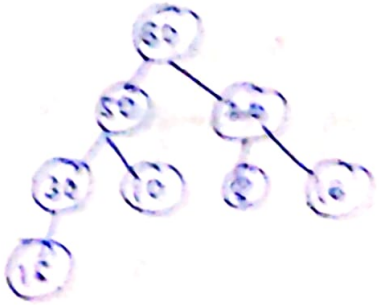


$\lfloor \frac{4}{2} \rfloor = 2$



$\lfloor \frac{2}{2} \rfloor = 1$

⇒



Worst case

No. of swaps = Height of tree = $O(\log n)$

Best case

No. of swaps = $O(1)$

- **Delete operation on max heap**

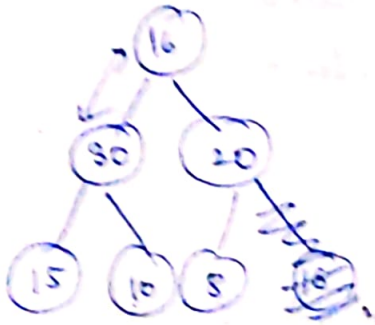
→ Root element is deleted ~~from~~ always

→ Tree has to be adjusted accordingly to retain complete binary tree

To retain CBT, we always replace last element

30	20	15	10	8	16

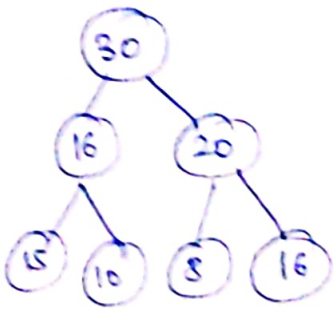
Now, build max. heap



check 16, 30, 20

root left right

greater replaces the root



16	30	20	15	10	8
0	1				

$i \quad i*2 \quad i*2+1$ — check greater

Deletion takes — $O(\log n)$ Time as total no. of swaps is max^m - Height of tree

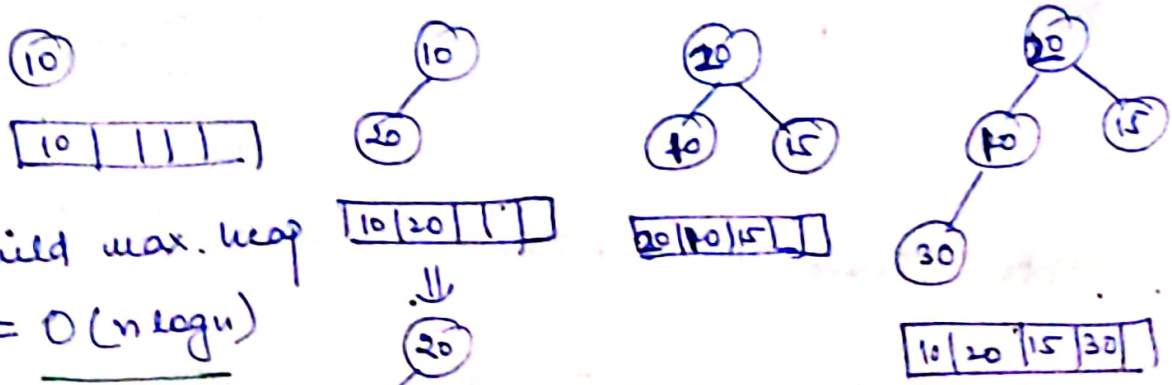
Idea is

If we delete largest element & put it in freed up space
— we get sorted order of elements.

Heap sort -

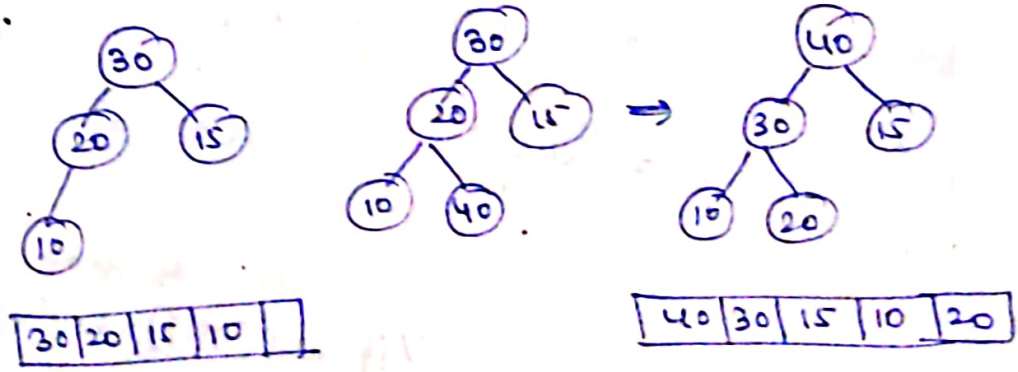
i) Create heap
ii) Delete heap elements one by one

e.g 10, 20, 15, 30, 40

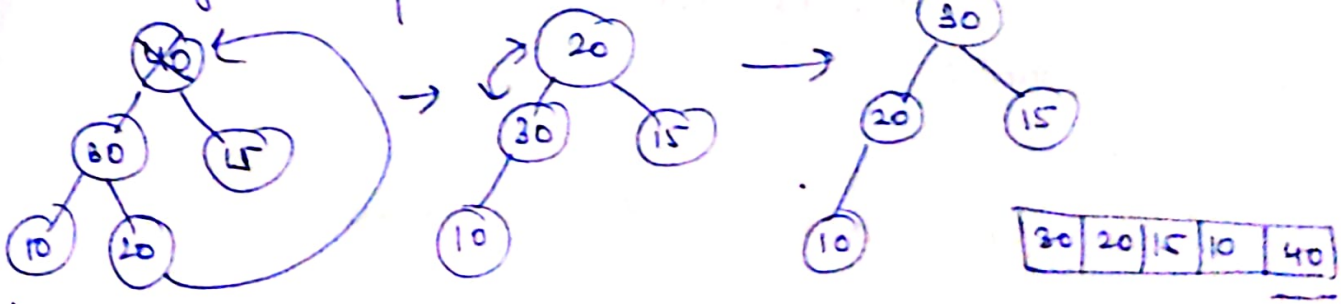


i) Build max. heap
= O(n log n)

n elements moving up to height log n



ii) Delete heap elements one, by one & store in freed up



[illegible]

[illegible]

[illegible]

Heapify-down (H, i) (min heap) — O(log n)

```
n = length(H)
if 2i > n then
    terminate with H unchanged
else if 2i < n then
    left = 2i
    right = 2i + 1
    let j be the index that minimizes / maximizes
    key[H[left]] and key[H[right]]
else if 2i = n then
    j = 2i
endif
if key[H[j]] < key[H[i]] then / key(H[j]) > key[...]
    swap the array entries H[i] & H[j]
    Heapify-down (H, j)
endif
```

Insertion $O(n \log u)$, heapifying (property to attain heap property $- O(u)$

Total complexity $- O(u \log u)$

for insertion

for deletion $- O(u \log u)$

Priority Queue

Elements have priority and they are inserted or deleted based on priority.

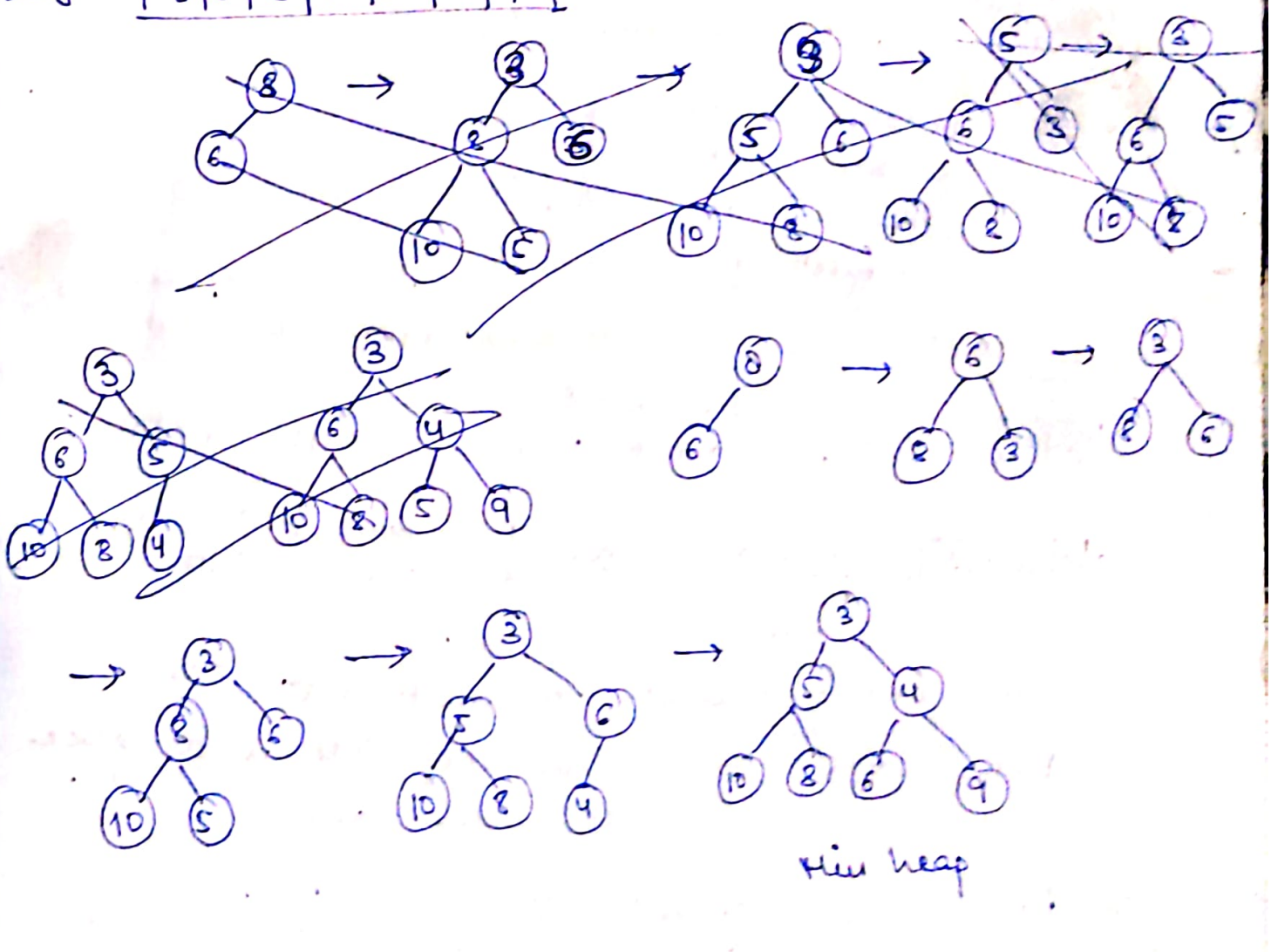
Smaller no. = Higher priority] Min heap

Larger no. = Smaller priority] Max heap.

If we implement priority queues using array
Time to insert or delete 1 elements — O(n)

If we implement priority queue using heap
time to insert or delete 1 element — O(log n)

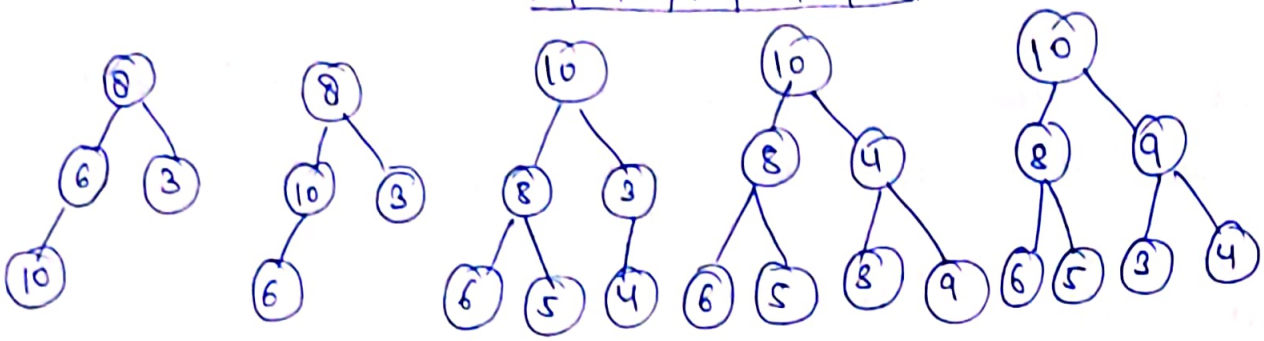
e.g [8|6|3|10|5|4|9]



Min heap

Max. Heap

8	6	3	10	5	4	9



Definition of Priority Queue :-

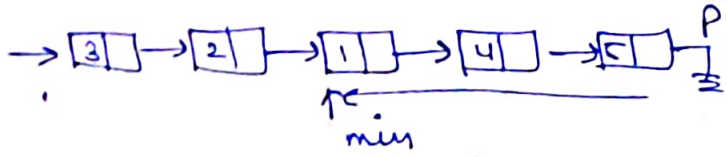
A priority queue is a data structure that maintains a set of elements S, where each element $v \in S$ has an associated value key (v) that denotes the priority of element v. (Smaller keys represent higher value or vice-versa) Priority queues support the addition and deletion of elements from the set, and also the selection of element with smallest key.

e.g Process scheduling
stable matching
railway reservation
counselling

Problem with array or linked list represent°:

3	2	1	4	5

Extract 1
move 4 & 5 (shift — O(n))



Scan full — O(n)
move pointer

If we sort :, I want to insert 8.5

1	2	3	4	5

— Run loop till n to place it, check min

So, in anycase, complexity — $O(n)$

can we reduce it — Yes.

How — Heaps —

How we represented heaps — using arrays as we know

for i^{th} node

left child = $2i$ position

right node = $2i+1$ position

parent node = $\lfloor \frac{i}{2} \rfloor$

Ques Can we represent heap using linked list.

Ans. Yes

Do we prefer representing heap using linked list.

Ans No

—— why ???

Heap-Insert — Insert element at last

fix the heap using heapify-up

Priority Queue Implementation using Heap :-

1. StartHeap (N) :- — O(N)

2. ~~Insert (H,V)~~ Returns an empty heap H that is set to store atmost N elements (initializing the array for holding heap)

2. Insert (H,V) — O(log n)

inserts item v to H. (H currently have n elements)

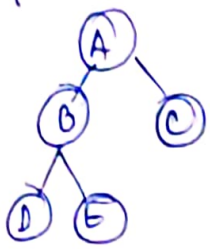
3. FindMin (H) - finding min element of heap - O(1)

4. Delete (H,i) - deletes i in O(logn) time

5. ExtractMin (H) - identifies and deletes an element with min key value from heap

— O(logn)

Heap can store values as well as other data.



— Needs additional array [Position] to store position of each element in heap.

- Delete [H, Position [v]) — delete element v.
  — O(log n)
- changekey (H, v, α) — O(log n)

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

$$<< n! << n^n$$

Assignment-

Q.1 Flowchart

Q.2 Pseudocode

Q.3 Find if $f(n) = O(g(n))$ or $g(n) = O(f(n))$

a. $f(n) = \frac{n(n-1)}{2}$ and $g(n) = 6n \Rightarrow g(n) = O(f(n))$

Method 1 Compare the growth rate of two functions

For $f(n)$ —

highest degree of numerator = 2

denominator = 2

as $n >$, $n^2 \gg$ — n larger, n^2 dominates

$\frac{n^2}{2} - \frac{n}{2} \leq c \cdot 6n$

$c = 2$

$n = 1$

$\frac{n^2}{2} \leq c \cdot 6n$

$\frac{1}{2} \leq 12$

$c = 2$

$n = 10$

$50 \leq 120$

$c = 2$

$n = 10^3$

$\frac{1000000}{2} \leq 12000$ false

$\therefore \frac{n^2}{2}$ is higher order growth than $6n$ as $n \to \infty$

Method.2

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{\frac{n(n-1)}{2}}{\frac{6n}{1}} = \frac{n-1}{12}$$

$$= \infty$$

$\therefore f(n)$ grows faster than $g(n)$

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

(2) $f(n) = n + 2\sqrt{n}$, $g(n) = n^2$

$n + 2\sqrt{n} \leq c \cdot n^2$

$n \leq c \cdot n^2$ — True

$2\sqrt{n} \leq c \cdot n^2$ — True

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

or $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{n^2}{n+2\sqrt{n}} = \frac{n^2}{\frac{n}{1} + \frac{2\sqrt{n}}{1}}$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n+2\sqrt{n}}{n^2} = \frac{1}{n} + \frac{2}{n\sqrt{n}}$

As $n \to \infty$, $\frac{1}{n} \to 0$, $\frac{2}{n\sqrt{n}} \to 0$

(3) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n + \log n}{n\sqrt{n}} = \lim_{n\to\infty} \left(\frac{1}{\sqrt{n}} + \frac{\log n}{n\sqrt{n}} \right)$

~ 0

$f(n) = O(g(n))$ ✓

(4) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n \log n}{\frac{n\sqrt{n}}{2}} = \lim_{n\to\infty} \frac{2\cancel{n}\log n}{\cancel{n}\sqrt{n}}$

~ 0

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

(5) $\lim_{n\to\infty} \frac{f(n)}{g(n)}$

$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{\log n + 1}{2(\log n)^2} = \frac{\log n}{2 \cdot \log n^{\infty}} + \frac{1}{2(\log n)^2}$

As $n \to \infty$ $\frac{1}{2\log n} \sim 0$, $\frac{1}{2(\log n)^2} \sim 0$

$\therefore$ ~~$g(n) \neq O(f(n))$~~ $g(n) = O(f(n))$

or ~~$f(n) = O$~~

Q.4

a) $2n^2+1 = O(n^2)$ — True

$2n^2+1 \leq c \cdot n^2$

for $c = 3$

$2n^2+1 \leq 3n^2$, so true

b) $n^2(1+\sqrt{n}) = O(n^2)$ — false

$n^2 + n^{2.5} = O(n^2)$

$n^2 + n^{2.5} \leq c \cdot n^2$

or $n^2(1+\sqrt{n}) \leq c \cdot n^2$

$n=1 \Rightarrow 1 \cdot (1+1) \leq 2 \cdot 1$

$2 \leq 2$

$n=4 \Rightarrow 16(1+\sqrt{4}) \leq 32$

$16 * 3 \leq 32$

$48 \not\leq 32$

$n=9 \quad 81(1+3) \leq 162$

$81 \times 4 \leq 81 \times 2$

$c = 3$

$n^2 + n^{2.5} \leq 3n^2$

$c = 2$

$n^2(1+\sqrt{n}) \leq 2n^2$

$n \geq 1$

$10^4 * 11 \leq 2 \cdot 10^4$

$11 \leq 2$

c) $n^2(1+\sqrt{n}) = O(n^2 \log n)$ — false

$n^2(1+\sqrt{n}) \leq c \cdot n^2 \log n$

$1+\sqrt{n} \leq c \log n$

$n^2 + n^{2.5} \leq cn^2 \log n$

check for n^2 $\quad n^2 \leq c \cdot n^2 \log n$ — true

$n^{2.5} \leq c \cdot n^2 \log n$

$\sqrt{n} \leq c \cdot \log n$

d) ~~d~~ $3n^2 + \sqrt{n} = O(n + n\sqrt{n}) + \sqrt{n}) = O(n\sqrt{n})$

$3n^2 + \sqrt{n} \leq c\,(n + n\sqrt{n} + \sqrt{n})$

$3n^2 \leq c \cdot (n + n\sqrt{n} + \sqrt{n})$

$3n^2 \leq c \cdot n\sqrt{n}$ — false

e) $\sqrt{n} \log n = O(n)$

$\sqrt{n} \log n \leq c \cdot n$ $\quad c = 2$, $n = 1024$

$10.08 * 10$

$100 \leq 2 \cdot 1024$

True

f) $\log n \in O(n)$ — True

g) $n \in O(n \lg n)$ — True

h) $n \lg n \in O(n^2)$ — True

i) $2^n \in \Omega(6^{\ln n})$

$\because a^{\log_b c} = c^{\log_b a}$

$\therefore 6^{\ln n} = n^{\log_e 6}$ natural log base e

$\because n^{\ln 6}$ — polynomial growth

$\therefore 2^n \in \Omega(n^{\ln 6})$ — True

---

j) ~~log~~ $\lg^3 n \in O(n^{0.5})$ — ~~True~~ false

$\lg^3 n = (\log_2 n)^3$

$(\log_2 n)^3$ grows slower than $n^{0.5}$ as $n \to \infty$.

only true for $n = 2$

else false — $n \geq 2$

1024

4096

~~$(\log)^2$~~

1

$T(n) = 5T(n-1) - 6T(n-2)$ $T(0) = 1$

$T(1) = 2$

$T(n-1) = 5T(n-2) - 6T(n-3)$

$T(n-2) = 5T(n-3) - 6T(n-4)$

$T(n-3) = 5T(n-4) - 6T(n-5)$

$T(n) = 5[5T(n-2) - 6T(n-3)] - 6T(n-2)$

$= 19T(n-2) - 30T(n-3)$

$= 19[5T(n-3) - 6T(n-4)] - 30T(n-3)$

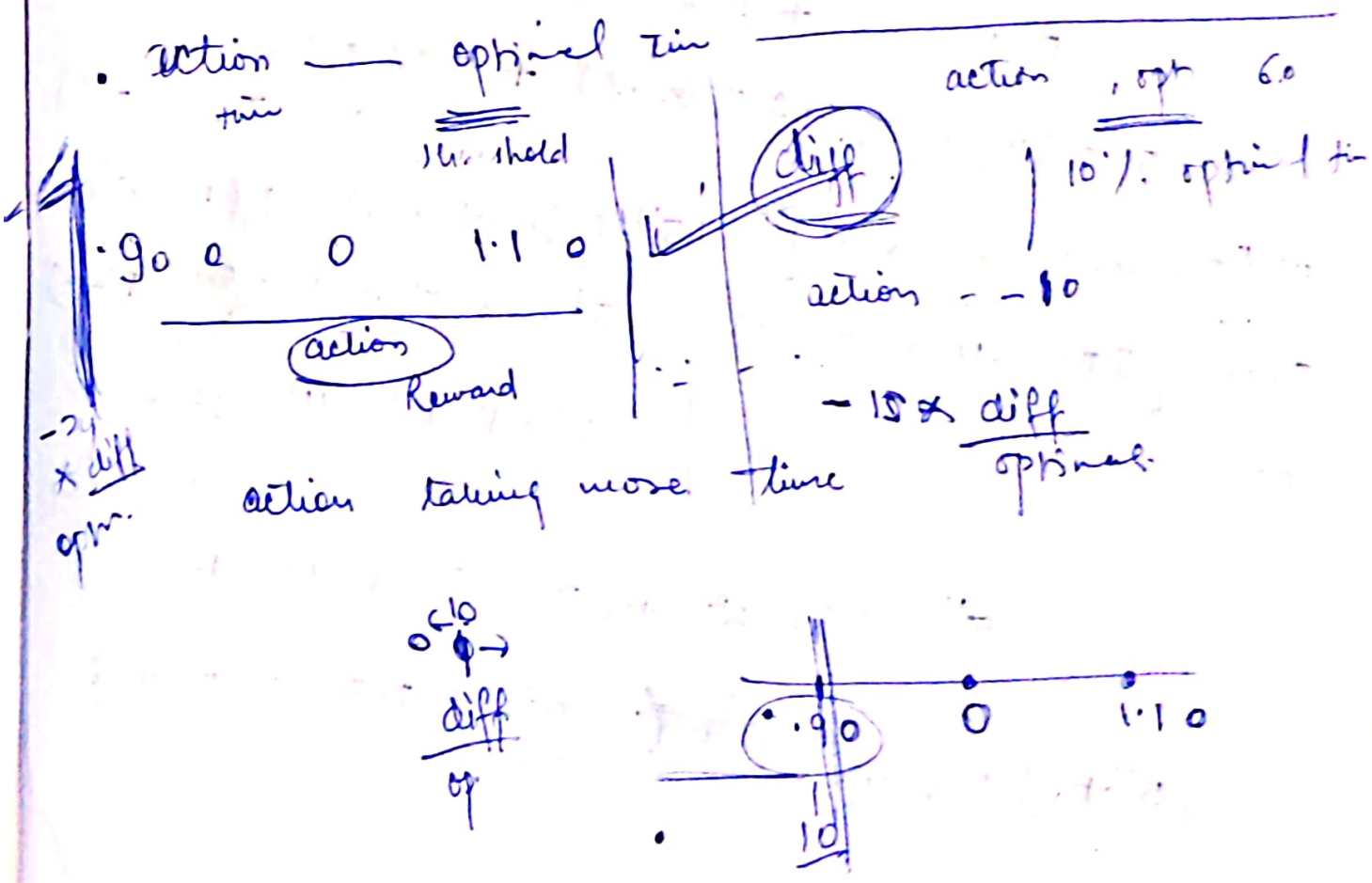
$= 95T(n-3) -$

$= 5^2 T(n-2) - 5*6\,T(n-3) - 6T(n-2)$

$T(n) = 5^2[5T(n-3) - 6T(n-4)] - 5*6\,T(n-3) -$

$= 5^3 T(n-3) - 5^2*6\,T(n-4) - 5*6\,T(n-3)$

-6



~~$T(n) = 5^3 T(n-3) - 5^2 * 6\ T(n-4) - 5 * 6\ T(n-3)$~~

$T(n) = 5T(n-1) + 6T(n-2)$ $\quad T(0) = 1$

$T(n) = 5T(n-1) + 6T(n-2)$ —(1) $\quad T(1) = 2$

$T(n-1) = 5T(n-2) + 6T(n-3)$ —(2)

$T(n-2) = 5T(n-3) + 6T(n-4)$ —(3)

$T(n-3) = 5T(n-4) + 6T(n-5)$ —(4)

$T(n) = 5[5T(n-2) + 6T(n-3)] + 6T(n-2)$

$= 5^2 T(n-2) + 5*6T(n-3) + 6T(n-2)$

$= (5^2+6)T(n-2) + 5*6T(n-3)$

$T(n) = (5^2+6)[5T(n-3) + 6T(n-4)] + 5*6T(n-3)$

~~$= 5^3 T(n-3) + 5 \cdot 6\ T(n-3) + 5$~~

$5^3 T(n-3) + 2 \cdot 5 \cdot 6\ T(n-3) + 5^2 6\ T(n-4) + 6^2 T(n-4)$

$5^3[5T(n-4) + 6T(n-5)]$ $\quad 5^4 T(n-4) + 5^3 \cdot 6\ T(n-5) +$

$5 \cdot 6 \cdot 2 \cdot [5T(n-4) + 6T(n-5)]$ $\quad 5^2 \cdot 6 \cdot 2\ T(n-4) + 2 \cdot 5 \cdot 6^2 T(n-5)$

5^2 $\quad + 5^2 6T\ T(n-4) + 6^2 T(n-4)$

$(5^4 + 5^2 \cdot 6 \cdot 2^2 + 6^2)\ T(n-4) +$

~~$5^3 6$~~ $(5^3 6 + 2 \cdot 5 \cdot 6^2)\ T(n-5)$

$(5^2+6)T(n-4) +$

$=$

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

~~$T(n^2) = n\,T(n) + n^2$~~

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

$$T(n^{1/2}) = n^{1/4}\, T(n^{1/4}) + n^{1/2} \quad \text{——(2)}$$

$$T(n^{1/4}) = n^{1/8}\, T(n^{1/8}) + n^{1/4}$$

$$T(n) = n^{1/2}\left[n^{1/4}\, T(n^{1/4}) + n^{1/2} \right] + n$$

$$= n^{3/4}\, T(n^{1/4}) + 2n$$

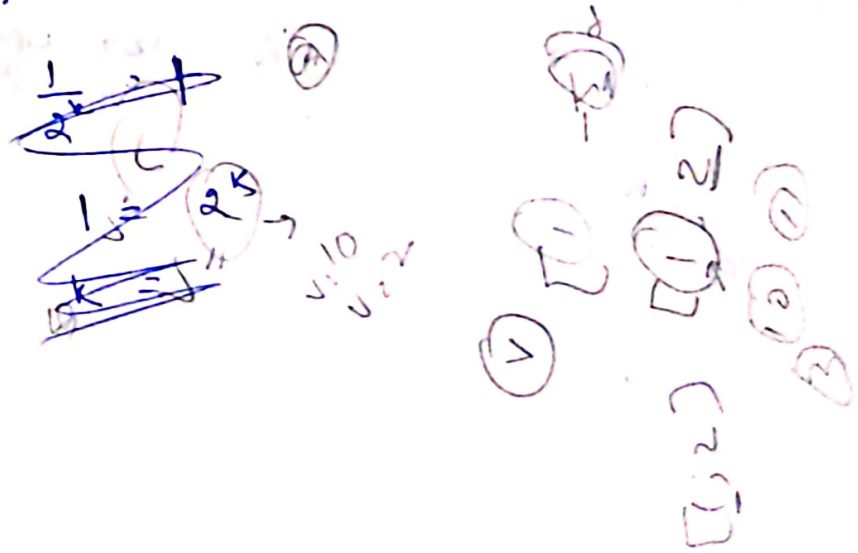
$$T(n) = n^{3/4}\left[n^{1/8}\, T(n^{1/8}) \right] + 2n + n^{1/4}$$

$$= n^{7/8}\, T(n^{1/8}) + 3n$$

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

$$n^{1/2^k} = 1$$

$$n^{-2^k} = 1$$



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

$n^0 = n^{\frac{1}{2^k}}$

$0 = \frac{1}{2^k}$

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

$2^k = \log_2 n$

$T(2) = \sqrt{2}$

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

$T(1) = 2$

$n = 2^k$

$k = \log_2 n$

$n^{1/2^k} = 2^1$

$1 = \log_2 n^{\frac{1}{2^k}}$

$1 = \frac{1}{2^k}\log_2 n$

$2^k = \log_2 n$

$k = \log\log_2 n$

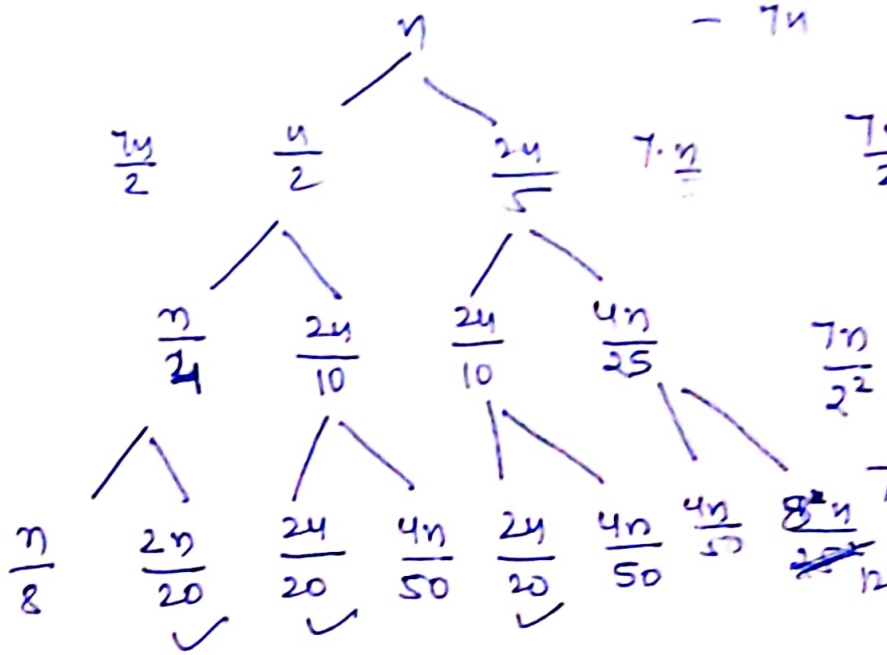
$n^{1-\frac{1}{2^k}}\, T(1) + \log\log n$

$n^{1-\frac{1}{\log_2 n}} \cdot 2 \cdot \log\ldots$

$\frac{n}{n^{1/2^k}} \cdot 2 + \log\log n \; n$

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

n^{-2^k}



$= 7n$

$7n \cdot \frac{n}{n}$

$\frac{7n}{2} + \frac{7n}{5}$

$7n\left(\frac{1}{2} + \frac{1}{5}\right)$

$\frac{7n}{2^2} + \left(\frac{2^2 \cdot 7n}{5^2}\right)$

$7 \times 2\left(\frac{2n}{5}\right) +$

$+ 7\left(\frac{4}{25} \cdot n\right)$

$\frac{n}{8} + \frac{6n}{20} + \frac{12n}{50} + \frac{4^2}{25^2} n$

$7n\left[\frac{1}{2^3} + 3 \cdot \frac{1}{2^2} \cdot \frac{2}{5} + 3 \cdot \frac{1}{2} \cdot \frac{2^2}{5^2} + \left(\frac{2}{5}\right)^3\right]$

$7n\left(\frac{1}{2^2} + 2 \cdot \frac{2}{5} \cdot \frac{1}{2} \cdot \frac{2^2}{5^2}\right)$

$7n\left(\frac{1}{2} + \frac{2}{5}\right)$

$7n\left(\frac{1}{2} + \frac{2}{5}\right)^2$

$7n\left(\frac{1}{2} + \frac{2}{5}\right)^3$

$7n\left[1 + \left(\frac{1}{2} + \frac{2}{5}\right) + \left(\frac{1}{2} + \frac{2}{5}\right)^2 + \left(\frac{1}{2} + \frac{2}{5}\right)^3 + \ldots\right]$

$7n\left[1 + \frac{9}{10} + \left(\frac{9}{10}\right)^2 + \left(\frac{9}{10}\right)^3 + \ldots\right]$

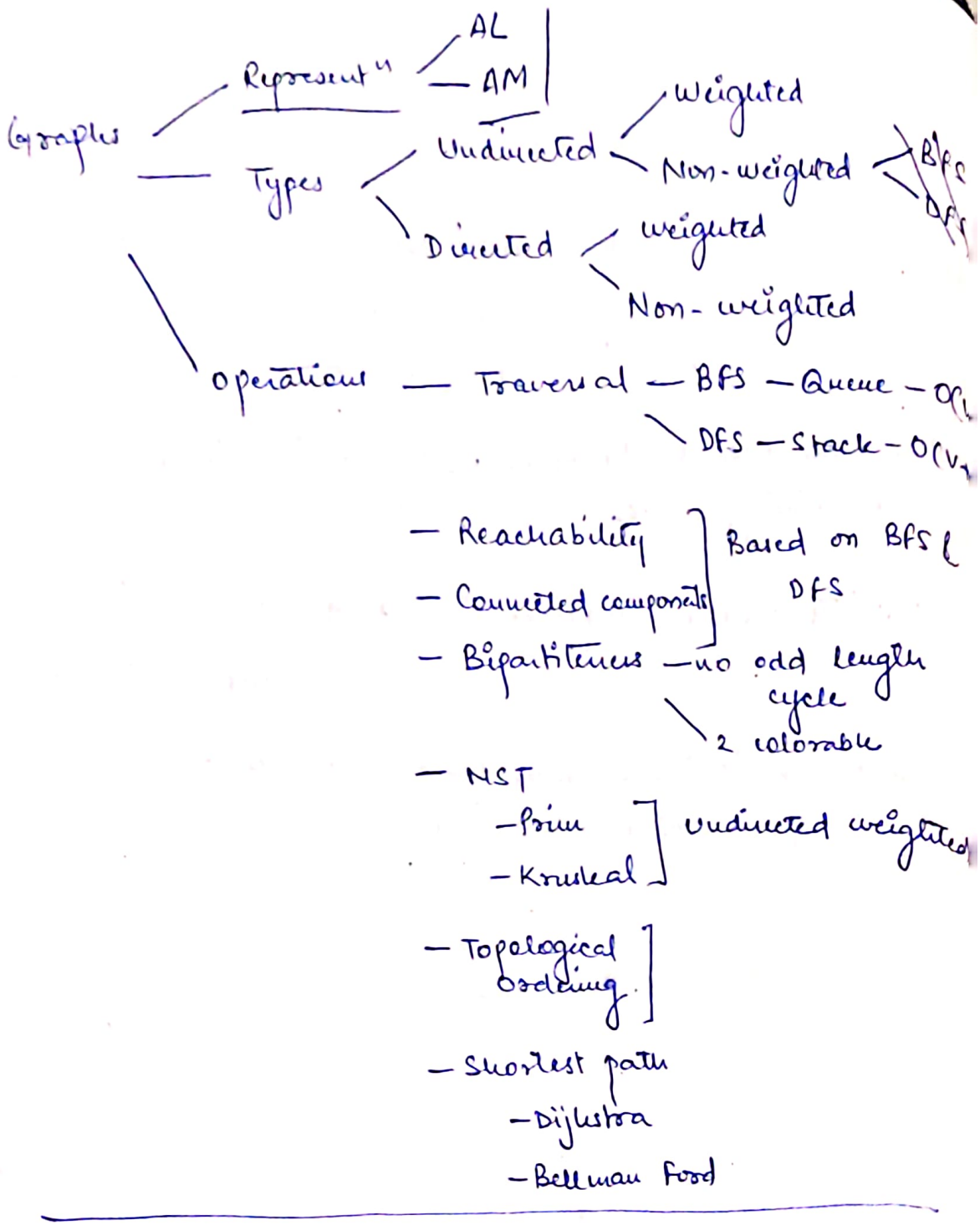
$\frac{5+4}{10} = \frac{9}{10}$

$7n \cdot \frac{1}{1-r}$

$7n \cdot \frac{1}{1-\frac{9}{10}} = 7n \cdot \frac{10}{1}$

$10n$

$= O(n)$


Graphs
Represent"
AL
AM
Types
Undirected
Weighted
Non-weighted
BFS
DFS
Directed
weighted
Non-weighted
operations
Traversal
BFS – Queue – O(
DFS – Stack – O(V
– Reachability
– Connected components
Based on BFS & DFS
– Bipartiteness – no odd length cycle
2 colorable
– MST
–Prim
–Kruskal
undirected weighted
– Topological Ordering
– Shortest path
–Dijkstra
–Bellman Ford

Graphs

Graph is a ~~data structure~~ defined as –

$G = (V, E)$

where V : set of vertices / nodes $(v_1, v_2, v_3 \ldots v_n)$

E : set of edges ($e_{uv} | e_{uv} = 1$ when there is a link b/w node u & node v) & $u, v \in V$

– Undirected Graph – shows symmetric relations

– Directed Graph – shows relation (asymmetric) b/w nodes (directed edges)

So – e_{uv} : asymmetric relation from u to v (where u is head & v is tail)

e leaves node u & enters node v. $e = (u, v)$ ordered pair

– If not mentioned – undirected graph.

• Examples

1. Transportation N/ws.

– Airline N/W

Nodes – airports

Edges – non-stop flight from u to v.

• → Concept of Hubs – nodes having higher traffic

→ It's possible to get b/w any two nodes in the graph via a very small no. of intermediate steps

– Railway N/W

nodes – railway stations

Edges – railway track b/w two railway stations (non-stop)

2. Communication Network
   - ~~Web~~ Computer N/W
3. Information network
   - WWW (directed) (webpage$_1$) ⟶ (webpage$_2$)
4. Social Networks
   Nodes - people
   Edges - relationship b/w people
5. Dependency network
   Directed N/W capturing interdependencies
   e.g - food web

- **Paths and Connectivity :-**

Maximum no. of edges in undirected graph $= \frac{n(n-1)}{2}$

Maximum no. of edges in directed graph $= n(n-1)$

Path -
In an undirected graph $G = (V, E)$, path is a sequence P of nodes $v_1, v_2 \dots v_{k-1} v_k$ such that each consecutive pair v_t, v_{t+1} is joined by an edge in G.

Simple path : All ~~edge~~ nodes are distinct
$v_1 v_2 \dots v_{k-1} v_k$

Cycle : $v_1, v_2 \dots v_{k-1} v_k$ where $v_1 = v_k$

Directed path : $v_1 v_2 \dots v_{k-1} v_k$ st for each consecutive pair v_t and v_{t+1} is joined by a directed edge in G.

Directed cycle :

Connected Graph :-

For undirected graphs, a graph is called connected if for every pair of nodes u and v, there is a path from u to v

For directed graph — we define the term strongly connected i.e. for every two nodes u and v, there is a path from u to v & from v to u.

Short path :-

Distance between two nodes u and v is minimum. If two nodes are not connected, we say they have infinite distance.

Trees –

connected undirected tree not having a cycle] exactly n-1 edges

Rooted tree – rooted at node r

Every n node tree has exactly (n-1) edges.

• Representation of Graph :-

1. Adjacency matrix – $O(n^2)$ – all cases
2. Adjacency list – $O(n+m)$ best case – $O(n+m)$
   $O(n^2)$ – worst case

complexity of checking if an edge is present in adjacency matrix – $O(1)$

complexity of checking if an edge is present in adjacency list – $O(u_v)$ – takes time proportional to degree of v.

Graph Connectivity & Graph Traversal

- Basic algorithm questions:
  - Node to node connectivity
  - Given a source node s, which are the reachable nodes from s.

Breadth First Search (BFS) — O(V+E)

Given $G = (V, E)$

Source vertex : s

→ BFS systematically explores the edges of G to discover every vertex that is reachable from s.

→ It can compute distances (smallest no. of edges) from s to each reachable vertex.

→ Also produces BFS tree.

→ It discovers all vertices at distance k from s before discovering any vertices at distance k+1

Algorithm

```
BFS (G, s)
for each vertex u ∈ G.V - {s}          — O(V)
    u.color = white
    u.d     = ∞
    u.π     = NIL
s.color = GRAY                         — O(1)
s.d = 0                                — O(1)
s.π = NIL                              — O(1)
Q = ϕ                                  — O(1)
ENQUEUE (Q, s)                         — O(1)
```

while $u \neq \phi$

u = DEQUEUE(Q)

for each $v \in G.Adj[u]$

if $v.color == WHITE$

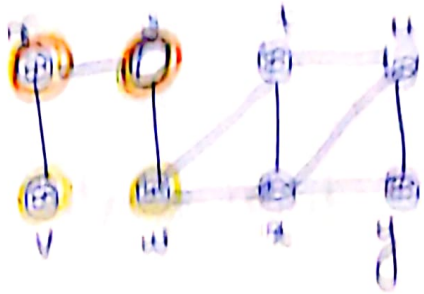
$v.color = GRAY$

$v.d = u.d + 1$

$v.\pi = u$

ENQUEUE(Q, v)

$u.color = BLACK$



A

	s	t	u	v	w	x	y
d	0	∞	∞	∞	∞	∞	∞
π	NIL	NIL	NIL	NIL	NIL	NIL	NIL

d

	s	t	u	v	w	x	y
d	0	1	∞	1	∞	∞	∞
π	NIL	s	NIL	s	NIL	NIL	NIL

Q

s	t	v

(1) $u = s$

$V = \{t, v\}$

(2) $u = t$

$V = \{v\}$

Complexity

Complexity — $O(V+E)$

For adjacency matrix

complexity — $O(V^2)$ for all cases

ex — $V = v$

$E = \frac{v(v-1)}{2}$

For adjacency list

complexity — $O(v)$ — sparse graph

ex $v = v$

$E \sim v$ or $O(v)$

complexity — $O(v^2)$ — dense graph

ex $V = v$

$E = O(v^2)$ or $E = \frac{v(v-1)}{2}$

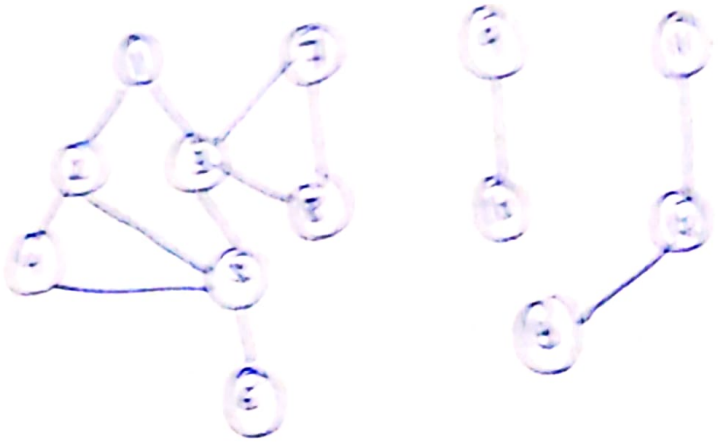
- what do we get from BFS -

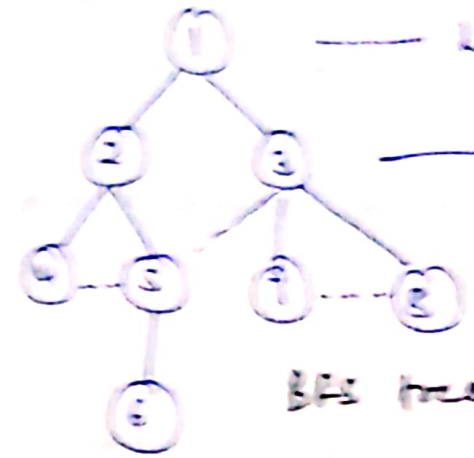
i) Connectivity

ii) Shortest path distance from s to all other vertices

iii) Connected components

Example :





BFS tree

— L_1 contains all neighbors of s.

— L_{j+1} consists of all nodes that are node in L_j and that have an edge in a node in layer L_j

So, for each $j>1$, Layer L_j produced by BFS consists of all nodes at distance exactly j from s. There is a path from s to t if and only if t appears in some layer.

For BFS tree —

Let T be a BFS tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x,y) be an edge of G. Then i & j differ by at most 1.

— proof by contradiction

Exploring a connected component

R: component of graph

Initially
$R = \{s\}$

Build R by finding paths to all nodes from source node s till we don't get any edge leading out of R.

R will consist of nodes to which s has path

Initially $R = \{s\}$

while there is an edge (u,v) where $u \in R$ & $v \notin R$

add v to R

end while

> The set R produced at the end of the algorithm is precisely the connected component of G containing s

Related problem ⇒ # connected components

- **Implementing BFS using Queue :-** $O(n+m)$

BFS (s)

O(1) — set discovered[s] = true and discovered[v] = false for all other v.

O(1) initialise L[0] to consist of single element s

O(1) set the layer counter i = 0

O(1) set the current BFS tree T = ϕ

while L[i] is not empty

Initialise an empty list L[i+1] — O(1)

for each node $u \in L[i]$

consider each edge (u,v) incident to u

if discovered[v] = false then

set discovered[v] = true

add edge (u,v) to the tree T

add i to the list L[i+1]

end if

end for

Increment the layer counter by one

end while

Depth First Search

→ Uninformed search technique
(We have only the present knowledge of graph & not the whole graph)

→ Implemented using stack (LIFO)

→ works till deepest node & after that backtracks

→ Incomplete (It may be possible that we don't get result – e.g. if cycles are present then infinite loop can be possible)

→ Non optimal (BFS is optimal & complete)

→ Time complexity – $O(V+E)$

Algo

```
DFS (v) :
    mark v as "Explored" and add v to R
    for each edge (v,v) incident to v
        if v is not marked "Explored" then
            Recursively invoke DFS (v)
        endif
    endfor
```

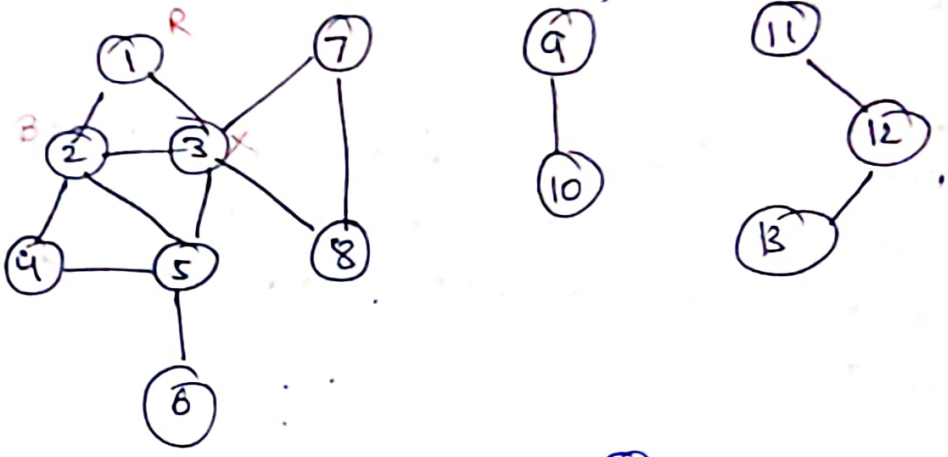
- Similarities

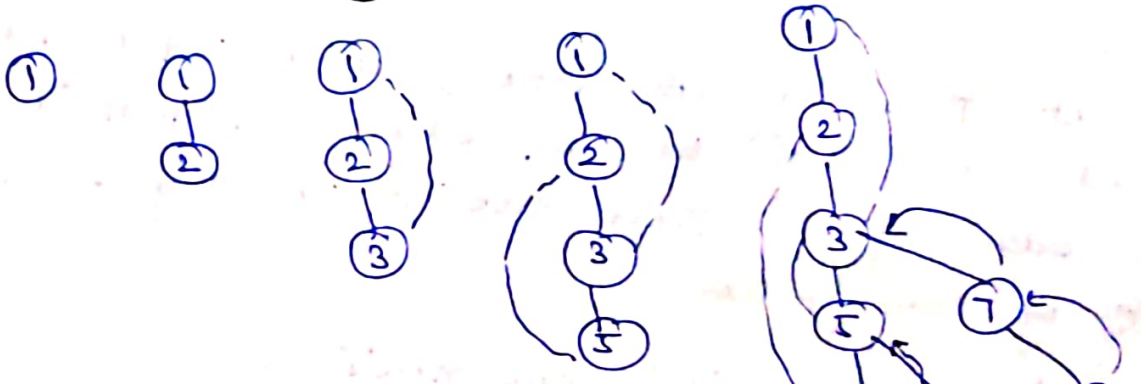
BFS	DFS
① Build the connected component containing s.	Build the connected component containing s.
② Uninformed search technique.	Uninformed search technique

- Differences

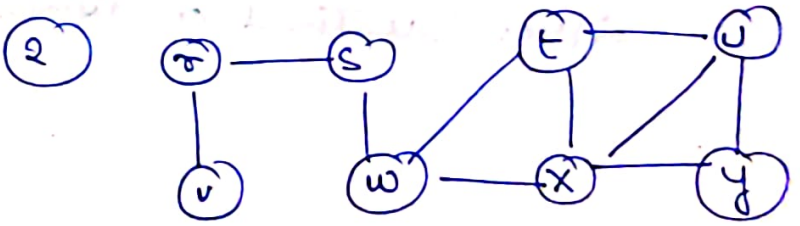
BFS	DFS
(1) level wise search	Depth-wise search
(2) Implemented using queue	Implemented using stack
(3) Always optimal soln.	Not always optimal soln.
(4) Trees of optimal heights	Tree height is not optimal
(5) Finds shortest path from source.	Does not finds shortest path from source

Example



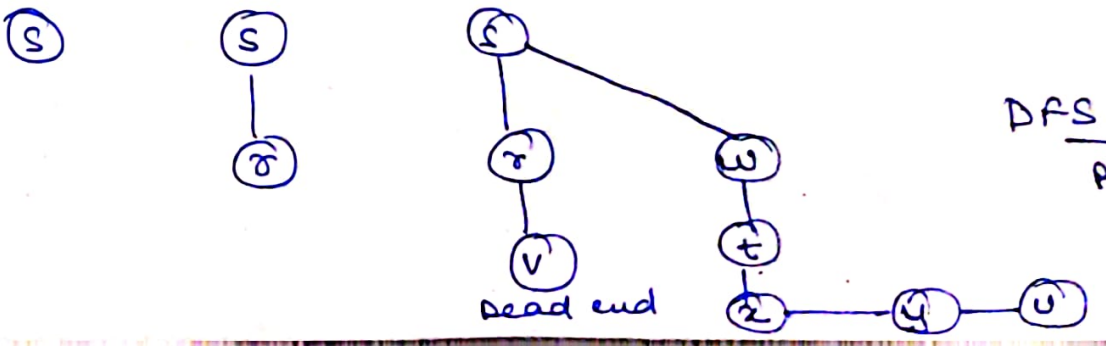


DFS Tree



BF.

FE



DFS Tree

P

Drawbacks

) cannot be used for computing shortest path from s.

DFS gives narrow and deeper trees than BFS.

Lemma –

For a given recursive call DFS(v), all nodes that are marked "explored" b/w the invocation and end of this recursive call are descendants of v in T.

Theorem –

Let T be a DFS tree, let x and y nodes in T, and let (x,y) be an edge of G that is not an edge of T. Then one of x or y is an ancestor of the other.

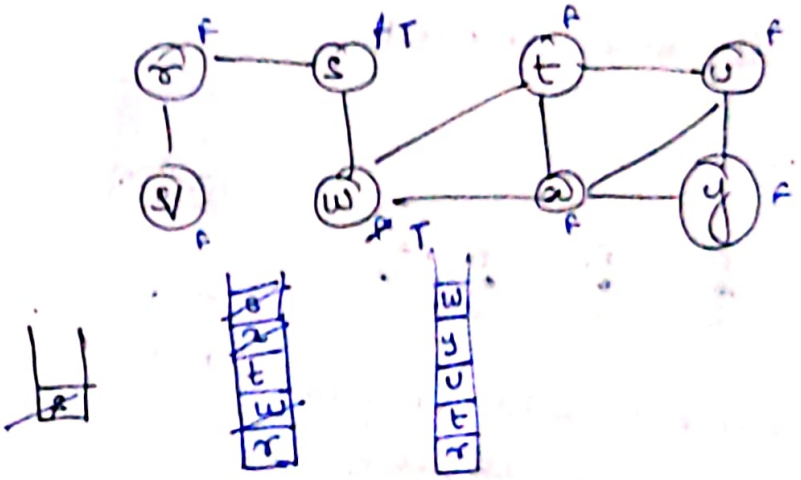
- The set of all connected components :–

For any two nodes s & t in a graph, their connected components are either identical or disjoint.

- DFS implementation using Stacks

DFS (s)
Initialise S to be a stack with one element s
while S is not empty
take a node u from S
if explored[u] = false then
set explored[u] = true
for each edge (u,v) incident to u
& exp[v]==f
add v to stack S
end for
endif
end while

e.g



Time complexity — O(m+n)

- Testing Bipartiteness : An application of DFS

Bipartite graphs

$G = V, E$

where $V = X, Y$ and $X \cap Y = \phi$ (disjoints)

& $\forall e_{ij} \in E$ s.t e_{ij} edge b/w node i & j

where $i \in X$

$j \in Y$

Lets says nodes in X — red colored

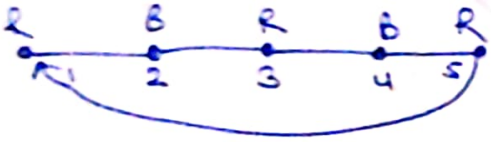
nodes in Y — blue colored

So, e_{ij} — i is red colored

j is blue colored

Problem Statement -

Not bipartite



cycle of odd length

1, 2, 3 . . . 2k, 2k+1

So, If a graph is bipartite, then it cannot contain an odd cycle.

⇒ odd length cycles in a graph are only ~~odd~~ obstacle in a graph to be bipartite.

⇒ Are there any other obstacles — ??

Designing the Algorithm :-

Assumption - Connected graph

```
fun (G, s)
  level = 0
  s.color = red
  ENQUEUE(s)
  for each neighbour of
  while Q ≠ ∅
    u = DEQUEUE(s)
    if level = even    (consider 0 also as even)
      u.color = red
```

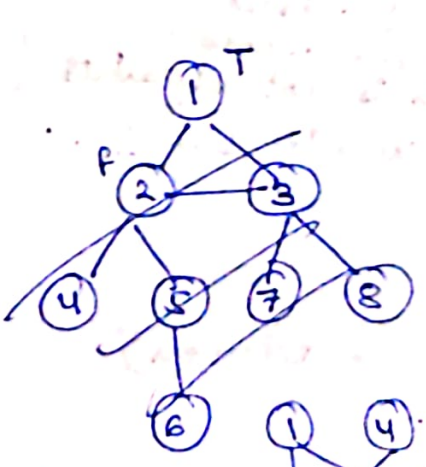
---

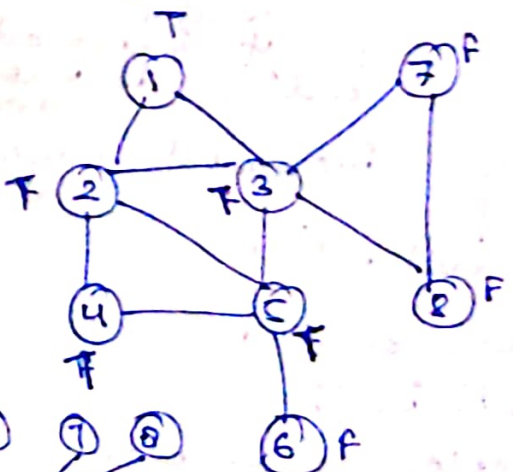
1. Take the source s & color it red
2. s - neighbors colored blue
3. neighbors of these colored red

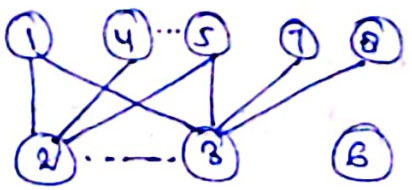
So, all alternate layers are colored red / blue.

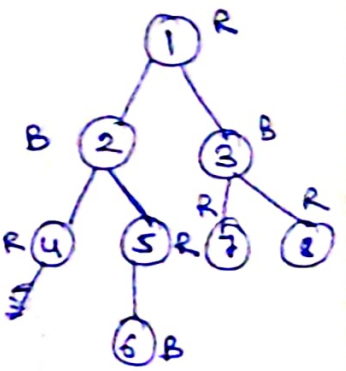
for e.g - Even layers - red

odd layers - blue









bipartite (G, s) ——— O(m+n)

1. set discovered[s] = true & discovered[v] = false
2. for all other nodes v
2. initialise L[0] consisting s ; s.color = "red"
3. set layer counter i = 0
4. set current BFS tree T = ϕ
5. while L[i] not empty

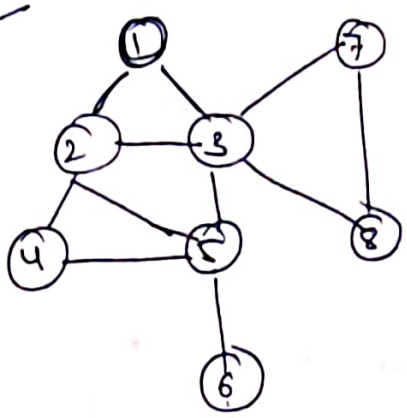
Initialise empty list L[i+1].
for each node u ∈ L[i]
~~if i is even~~
~~u.color = "red"~~
~~else~~
~~u.color = "blue"~~
consider each edge (u,v) incident on u
if discovered[v] = false
set_discovered[v] = true
add_edge (u,v) to tree T.
add v to the list L[i+1]
~~if odd~~
if (i+1) is even
v.color = "red"
else
v.color = "blue"
endif
endif ~~discovered[v] = true~~
if discovered[v] = true
if v.color == u.color
graph is not bipartite
endfor
increment layer counter by 1.
end while

u
u — v1
v2

scan all edges of tree ——— O(m+n)
for all (u,v) in T
if u.color == v.color
print "graph is not bipartite."
else
print "graph is bipartite".

Analyzing the algorithm :-

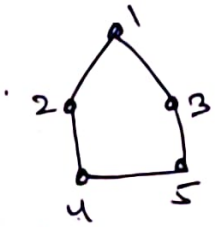
Ex-1



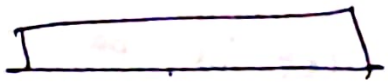
Queue = | 1 | 3 | 7 | 8 | 4 | 5 | 2 | 5 | 7 | 8 |

color = | R | B | B | | | | | |
1 2 3 4 5 6 7 8

Not bipartite



Not bipartite

color 

Theorem

Let G be a connected graph and let L_1 L_2 .. be the layers produced by BFS starting at node s. Then exactly one of the two things must hold.

(1) There is no edge of G joining two nodes of the same layer. In this case G is a bipartite graph in w/c nodes in even-numbered layers can be red and nodes in odd-numbered layers can be blue.

(2) There is an edge of G joining two nodes of the same layer. In this case, G contains an odd length cycle, and so it cannot be bipartite

i) [illegible] edge for [illegible]

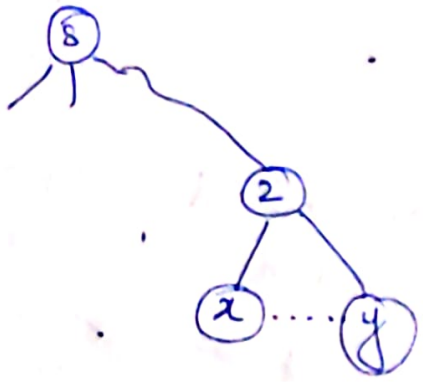
that means edges b/w adjacent layers

By coloring procedure, adjacent layers are differently colored. So each edge has ends with opposite colors.

So G is bipartite.

ii) G contains edge b/w nodes in same layer.

Let it be $e = (x, y)$ with $x, y \in L_j$. Let z be ancestor of nodes x, y



So, we get a cycle z-x-y-z

Length of cycle

$$(j-i) + (j-i) + 1 = 2(j-i) + 1$$

This is an odd no.

So, graph is not bipartite

- **Connectivity in Directed Graphs :-**

→ Asymmetric relation (qualitatively effects structure of graph)

→ E.g - www, citation N/W, SNs, email N/Ws

Representing DAG - indegree outdegree - adjacency list

2 lists < to which / from which

Graph search

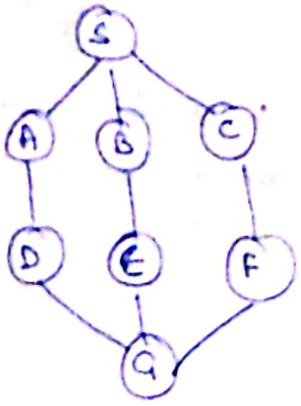
BFS, DFS] considering the directions O(m+n)

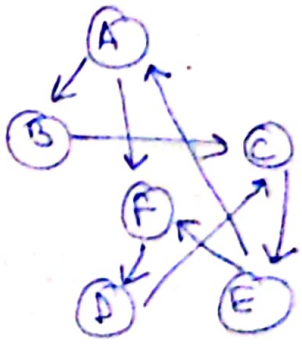
Directed version of BFS

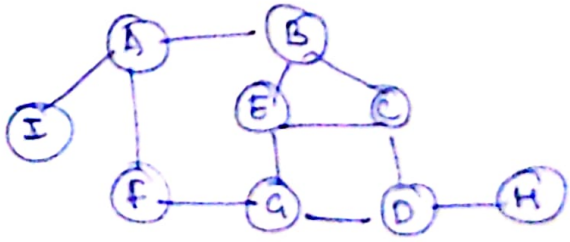
→ from s to t path may exist but vice-versa is not true

→ So for any intermediate node $s \to x \to y \to t$, there is a path to t, but a path to s from x & y is not guaranteed.

Example







- Strong Connectivity

Directed graph is called strongly connected if for every two nodes u & v.

1) If there is a path from u to v, then there is also a path from v to u.

u and v are called mutually reachable. So a graph is strongly connected if every pair of node is mutually reachable.

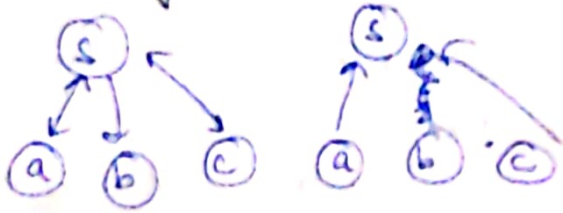
Properties

① If u and v are mutually reachable & v and w are mutually reachable, then u and w are mutually reachable.

$u \longrightarrow v \longrightarrow w$ $\quad u \longleftarrow v \longleftarrow w$

checking for strong connectivity

1. BFS (s)



or

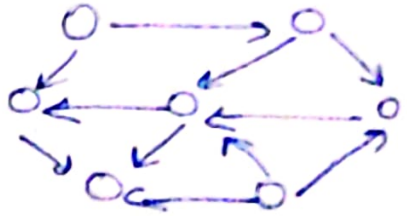
② For any two nodes s and t in a directed graph, their strong components are either identical or disjoint. (this property also shown for undirected graphs)

Total time $O(n+m)$

- **Directed Acyclic Graphs and Topological Sorting :-**

→ for undirected graph, no cycle means tree. # edges = n-1

→ for directed graph, no. of edges can be more as directions are also represented, So for a directed graph, it is very likely that it does not have a cycle but still a rich structure. DAGs can have large no. of edges.

eg 

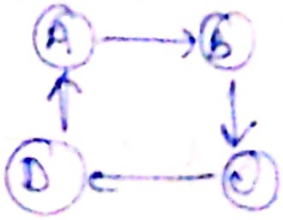
→ If a directed graph has no cycle, it is called directed acyclic graph (DAG)

Problem with DAG :-

→ Most of the dependency N/w are acyclic in nature
eg citation N/w, course prerequisite
So, tasks are arranged such that for two tasks i and j, i must be performed before j.

→ To represent such dependency graphs, we take nodes as task
edges in a precedency manner (A) ⟶ (B) i.e A needs to be done before B.

→ Resulting graph is a DAG. (as no cycle is possible)

X 

Not DAG

as we dont know which action to be performed
Deadlock is there

→ our need is to get a valid order in which the tasks can be performed, so that all dependencies are respected.

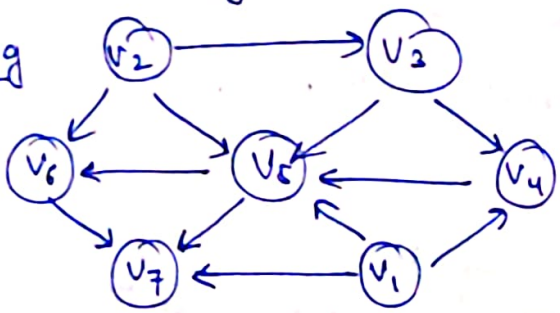
→ So, to get that valid order, we do topological ordering.

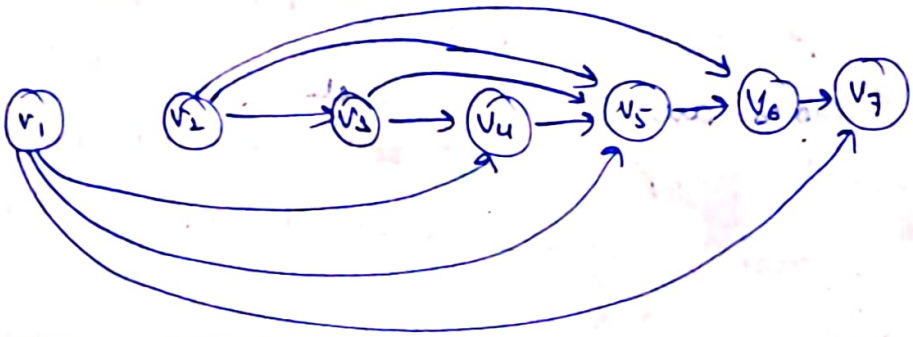
⇒ Topological Ordering :- (for directed graphs)

A topological ordering of G is an ordering of its nodes $v_1, v_2 \ldots v_n$, so that for every edge (v_i, v_j) we have $i < j$.

→ This means all edges are pointing forward in the ordering.

⇒ e.g





So,

> If G has a topological ordering, then G is a DAG.

——— (A)

Proof

Lets say $v_1\ v_2\ v_3 \ldots v_n$ be a topological ordering s.t $i < j$.

If there is a cycle b/w v_i and v_j, then as per defn.

$i < j$ and $j < i$ } this is not possible & contradicts the assumption that $v_1 v_2 \ldots v_n$ is a topological ordering.

Computing a topological ordering :-

Converse of (A) is -

Does every DAG has a topological sorting, if so how to find efficient one.

⇒ **Design and Analysis of Algorithm :-**

First we need to find the initial node. This initial node is such that it is not pointed by any other node.

So, In every DAG G, there is a node v with no incoming edges. ——— (B)

Now by induction

for $n = 1$ (1) topological ordering is there

$n = 2$ (1) → (2) " " " "

Let it be true for any n. i.e. G is a DAG

Then given a DAG G on n+1 nodes, we find a node with no incoming edges (as per statement B).

we first place v in topological ordering (as it is safe as all edges are pointing outwards). Now

G-{v} is a DAG as deleting v will not create any cycles that were not there previously.

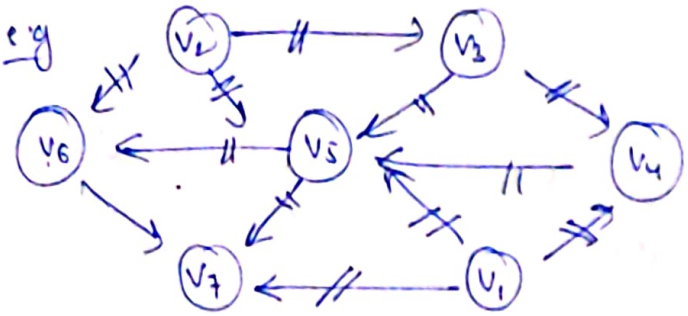
Also G-{v} has n no. of nodes, so we can apply the induction hypothesis on G-{v}. we append nodes of G-{v} in this order after v; this is an ordering of G in w/c all edges point forward & hence it is topological ordering.

Algorithm –

To compute topological ordering of G:

1. find a node v with no incoming edges & order it first
2. delete v from G
3. recursively compute a topological ordering of G - {v} and append this order after v.

e.g



v_1 v_2 v_3 v_4 v_5 v_6 v_7

Running Time :-

Identifying a node with no incoming edge and deleting it — $O(n)$

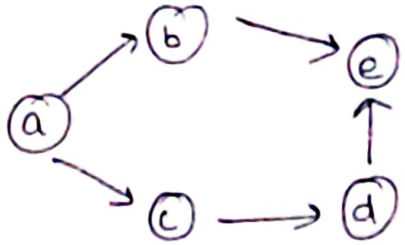
∵ algo runs for n iterations.

∴ total running time $O(n^2)$ — for dense graph.

∴ when m is much less than n^2 — sparse graph.

complexity = $O(m+n)$

Ques

(1) 

How many topological orderings does it have?

Topological ordering $v_1 v_2 \dots v_n$: $i < j$ for each (v_i, v_j)

Brute force

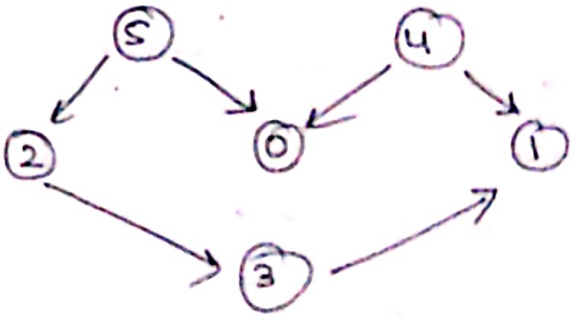
Total order possible $= 5! = 120$

check w/c one are topological

a b c d e

a c b d e

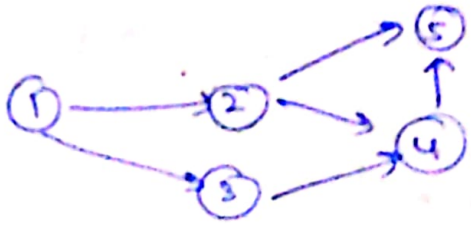
a c d b e

(2) 

5 4 2 0 3 1

4 . . . .

how many - ?

(3) 

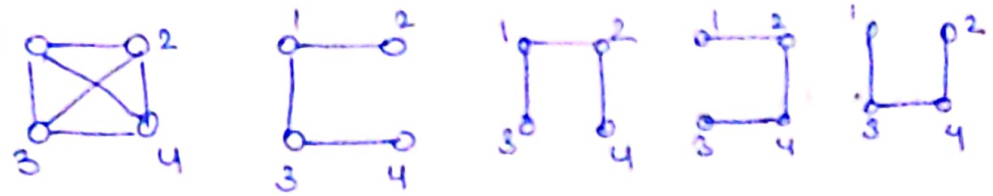
Minimum Spanning Tree

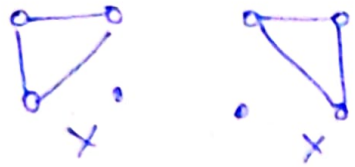
- **Spanning Tree :-**

A spanning tree is a connected subgraph 'S' of graph G(V, E) iff -

i) S contains all the vertices of G.

ii) S contains |V|-1 no. of edges.

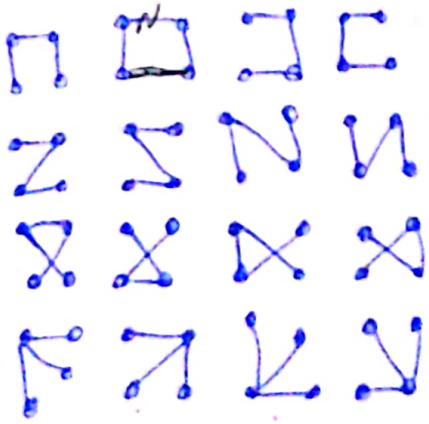
eg 



Ques: for a complete graph, how many spanning trees are possible ?

for K_4 

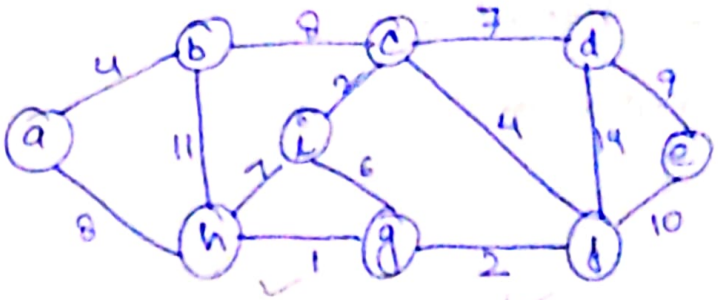
 — total 16 possible

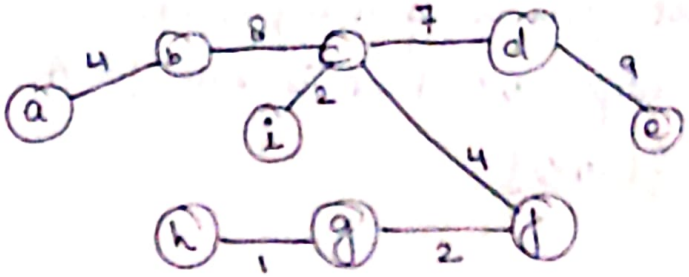


So, formula for no. of spanning tree for $K_n = n^{n-2}$

- **Minimum spanning tree :-**

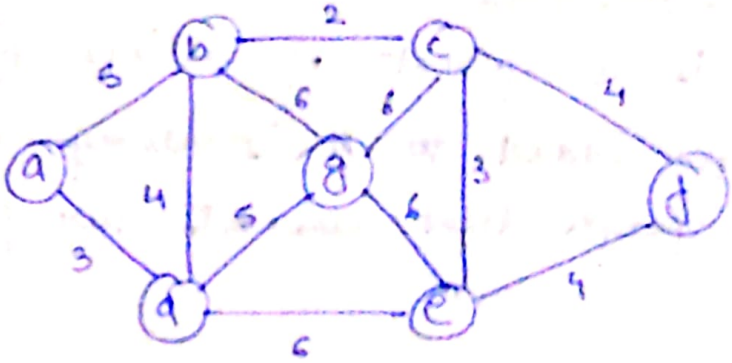
eg ①

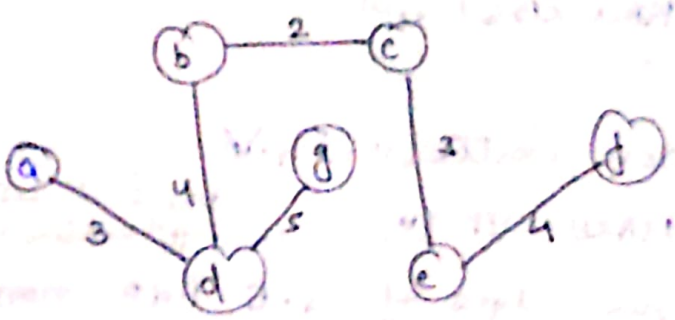




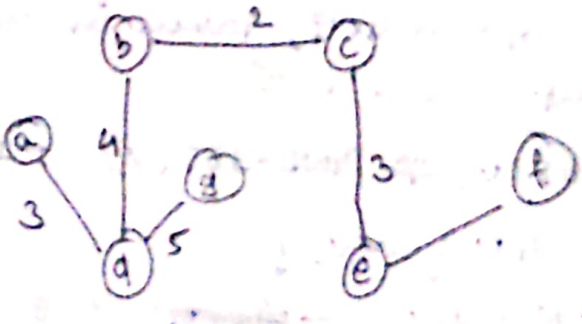
Total cost = 37

eg ②





Total cost = 21



Total cost = 21

- MST Algorithms :
  1. Kruskal's Algorithm
  2. Prim's Algorithm

Generic-MST(G, w)

A = ϕ
while A does not form a spanning tree
  find an edge (u,v) that is safe for A
  A = A ∪ {(u,v)}
return A

① Kruskal's Algorithm :- (Greedy Algorithm)

→ Set A is a forest of vertices given in G.

→ Safe edge added to A is always least-weight edge in graph that connects two components.

Algo -

MST-KRUSKAL(G,w)

O(1) 1. A = ϕ
O(V) 2. for each vertex v ∈ G.V
3. MAKE-SET(v) // Disjoint-set data structure to maintain several disjoint sets of elements
O(ElogE) 4. sort the edges of G.E into non-decreasing order by weight w
O(E) 5. for each edge (u,v) ∈ G.E, taken in nondecreasing order by weight — O(E)
6. if FIND-SET(u) ≠ FIND-SET(v) — O(E)
7. A = A ∪ {(u,v)}
8. UNION(u,v) — O(E)
9. return A

UNION - FIND datastruct

Running Time -

∵ G is connected

∴ $|E| \geq |V| - 1$

Total running time — $O(E \log E)$

∵ $|E| < |V|^2$

∴ $\log |E| = O(\lg V)$

∴ Complexity $= O(E \log V)$

UNION - FIND data structure

→ Allows us to maintain disjoint sets (graph components)

→ operations

FIND(v) : returns name of set containing v.

UNION (u, v) : takes two sets u, v & merge them to a single set.

MAKE-SET (v) : disjoint set

→ This data structure can only be used to maintain components of a graph as we add edges ; it is not designed to handle the effects of edge deletion, which may result in a single component being "split" into two.

② Prim's Algorithm :- (Greedy)

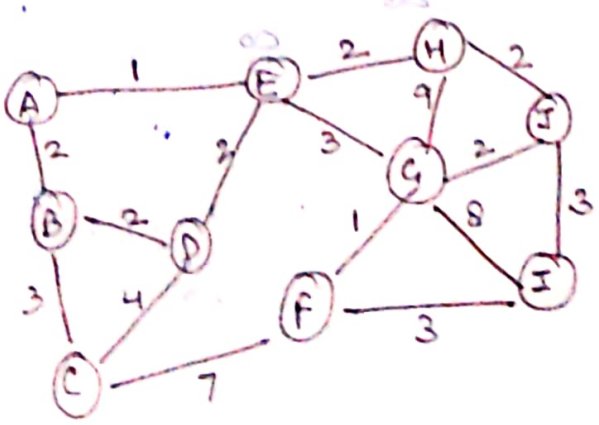
→ Property : edges in set A always form a single tree

→ Tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V.

MST-PRIM (G, w, r) — O(V+E)

```
1.  for each u ∈ G.V
2.      u.key = ∞
3.      u.π = NIL   // Predecessor
4.  r.key = 0
5.  Q = G.V
6.  while Q ≠ ϕ
7.      u = EXTRACT-MIN(Q) // Min priority queue based on key attribute
8.      for each v ∈ G.adj[u]
9.          if v∈Q and w(u,v) < v.key
10             v.π = u
11             v.key = w(u,v)
```

eg 2



cost

Shortest Path Problems

Dijkstra Algorithm

Problem

→ Given a directed graph $G=(V,E)$ with designated start node s.

→ Each edge e has a weight/length $w \geq 0$ associated with it. weight can be cost, distance etc..

→ for a path P
length of P, denoted by $l(P)$ = sum of length of all edges in P

→ Goal is to determine shortest path.

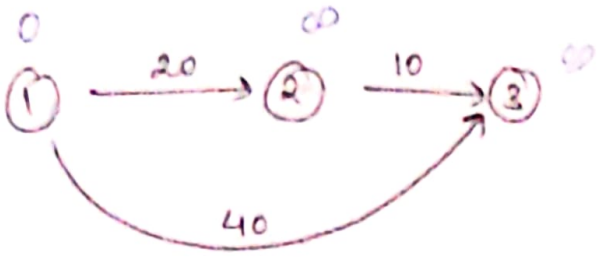
Algorithm (By Edsger Dijkstra in 1959)
— Single source shortest path on a weighted directed graph in which all edge weights are non-negative
— (It also works for undirected graph)

E.g — google maps

DIJKSTRA (G, w, s) — $O(V+E)$

```
1. Initialise-Single-Source (G, s)
2. S = ϕ
3. Q = G.V
4. while Q ≠ ϕ
5.     u = EXTRACT-MIN (Q)
6.     S = S ∪ {u}
7.     for each vertex v ∈ G.Adj[u]
8.         RELAX (u, v, w)
```

Example 1

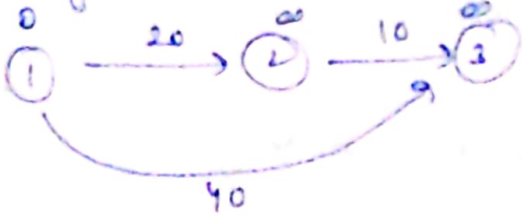


Relaxation

if $d(u) + w(u,v) < d(v)$

$d(v) = d(u) + w(u,v)$

Initially



source = 1

0	∞	∞	d
1	2	3	

u = 1

0	20	∞

1, 2

$d(1) + cost(1,2) < d(2)$

$0 + 20 < \infty$

$20 < \infty$

1, 3

$d(1) + c(1,3) < d(3)$

$0 + 40 < \infty$

$40 < \infty$

0	20	40

u = 2

v = 3

$d(2) + c(2,3) < d(3)$

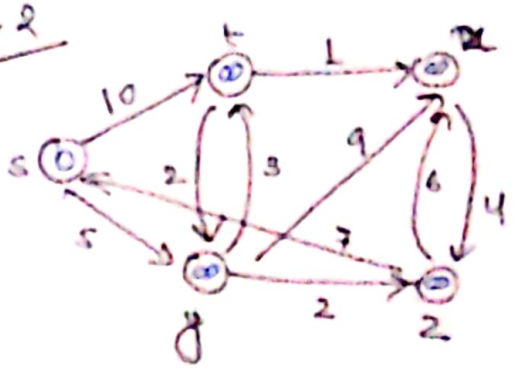
$20 + 10 < 40$

$30 < 40$

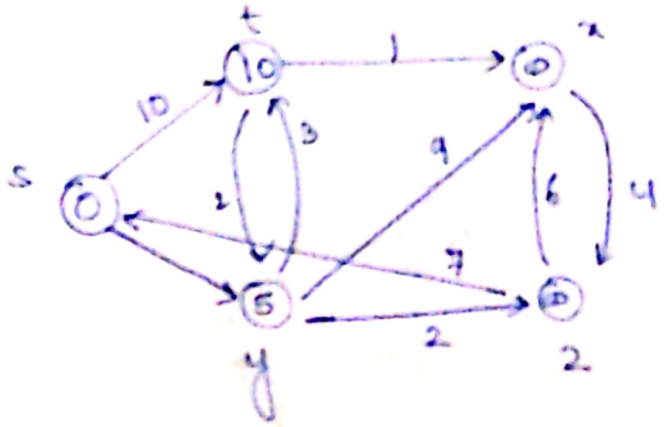
d

0	20	30

Example 2

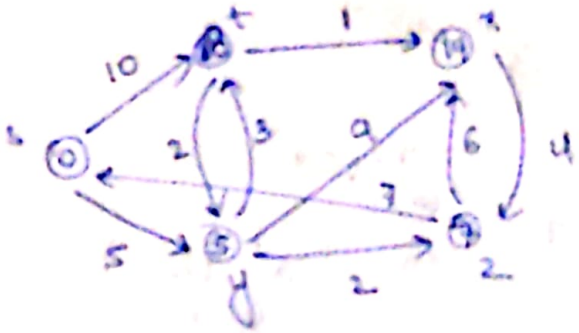


(1)



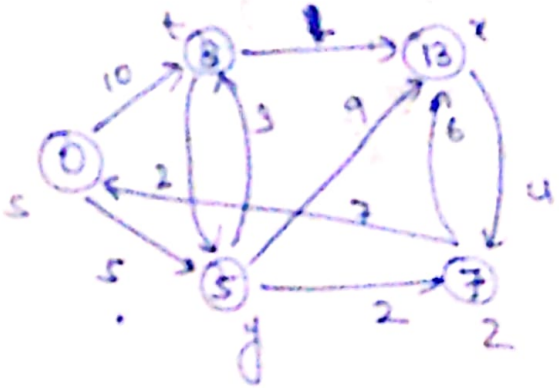
Extract s

(2)



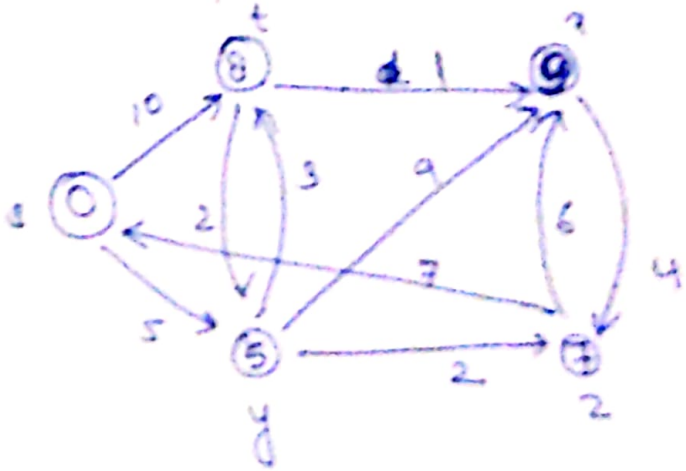
Extract y

(3)



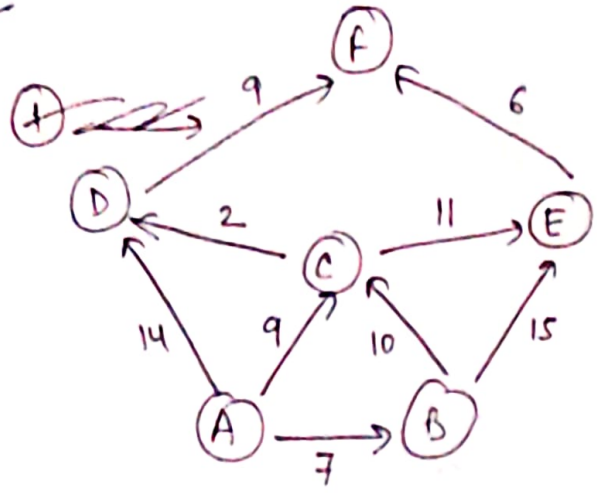
Extract z

(4)

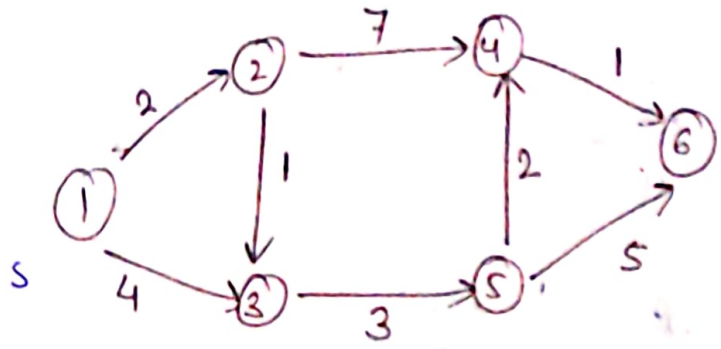


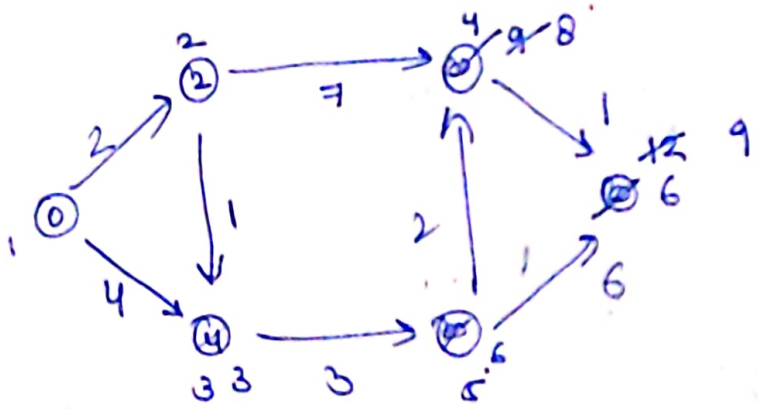
Extract t

Example 3

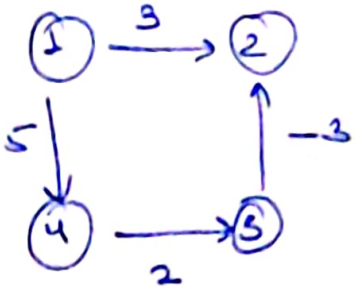


Example 4





→ Drawback of Dijkstra Algorithm :-



for negative weights