W-1

Exception Handling

Page :
Date :

Exception Handling

- Syntax Error → can't handle the error

- NameError - name used before defined
  ZeroDivisionError - division by 0 is an expression
  IndexError - Invalid Indexing.

Handle an Exeption Code

```
try:
    ...     ← code where error may occur
    ...

except IndexError:
    ...     ← Handle it
except (NameError, KeyError):
    ...     ← Handle multiple exceptin types.
except
    ...     ← Handle all other exceptin
else:
    ...     ← execute if try runs without error
```

example:-

```
scores = {"a": [3,22],
          "b": [200,3]}
```

a) updating

if batter [c] exists.
    scores [c].append(s)
if new batter
    scores [c] = [s]

traditional way

```
if c in scores.keys():
    scores [c].append (s)
else :
    scores [b] = [s]
```

Using exceptn

```
try:
    scores[b].append(s)
except KeyError:
    scores [b] = [s] .
```

Scanned by TapScanner

Page :
Date :

GCD - recursion Way -

```
def gcd (m,n):
    (a,b) = (max(m,n), min(m,n))
    if (a%b==0):
        return (b)
    else :
        return (gcd (b, a-b))
```

* This take much longer steps for big no.s

Euclid's Algo

suppose ~~m~~ not divide m (n<m)

now m = qn + r

So if n dividem n gcd (m,n) = n

else compute gcd (n, m mod n)

code :-

```
def gcd(m,n):
    (a,b) = (max(m,n), min(m,n))
    if (a%b==0):
        return b
    else:
        return gcd(b, a%b)
```

• time ∝ no. of digits.

* python → 10^7 operations per sec.

```
import time
start = time.perf_counter()
"
my code
"
end = time.perf_counter()
elapsed_time = end - start
```

Scanned by TapScanner

upper bound

$f(n)$ is said to be $O(g(n))$
if we can find c and n_0 s.t.
$c \cdot g(n)$ is upper bound for $f(n)$ for $n \geq n_0$

W-2

Page :
Date :

- $100n+5$ in $O(n^2)$
  for $n \geq n_0 = (5)$ $c = 101$
  i.e. $100n + 5 \leq 101n^2 \ \forall n \geq 5$.

where,

$100n + 5 \leq 100n + 5n = 105n, \ n \geq 1$

$105n \leq 105n^2$

choose $c = 105, n_0 = 1$.

- $100n^2 + 20n + 5$ is $O(n^2)$
  $100n^2 + 20n + 5 \leq 100n^2 + 20n^2 + 5n^2 = 125n^2$ for $n \geq 1$
  choose $c = 125, n_0 = 1$
- n^3 is not $O(n^2)$
  bcz for every c, $n \geq c$
  $\Rightarrow cn^2 > n^3$

properties :-

1. if $f_1(n)$ is $O(g_1(n))$, $f_2(n)$ is $O(g_2(n))$
   $\Rightarrow f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$.

that means if algorithm has 2 phase A and B with time $O(g_A(n))$ and $O(g_B(n))$, then as a whole algorithm will take $O(\max(g_A(n), g_B(n)))$ time.
i.e. least efficient phase is upper bound of the whole algorithm.

Scanned by TapScanner

Lower bound

Date:

$f(n)$ is $\Omega(g(n))$. if for any c and n_0

$\Rightarrow f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$

n^3 is $\Omega(n^2)$

- $n^3 > n^2$ for all n, $n_0 = 1$, $c = 1$.

y it is used for problems rather than programmes like for sorting an array by comparing and swapping we require a min. of $n \log n$ comparison

So lower bound is $\Omega(n \log n)$.

Tight Bound $\Theta(g(n))$

$f(n)$ is said to be $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$

- constant n_0, c_1, c_2

$$c_1(g(n)) \leq f(n) \leq c_2 g(n) \quad \text{for } \forall n > n_0$$

ex

$n(n-1)/2$ is $\Theta(n^2)$

upperbound

$$n(n-1)/2 \leq n^2/2 \quad \forall n \geq 0$$

Lowerbond

$$n(n-1)/2 \geq n^2/2 - (n/2 \times n/2) \geq n^2/4 \quad \text{for } n \geq 2$$

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

$\hookrightarrow \frac{1}{4}n^2 \leq \frac{n(n-1)}{2} \leq \frac{1}{2}n^2 \quad \forall n \geq 2$

tight bound is the optimal algo. for a problem.

Scanned by TapScanner

Page :
Date :

Calculating Complexity
(recursive / iterative)

- **Tower of Hanoi**

move n-1 from A to C
move large to B
move n-1 from C to B as A (transit peg)

Recurrence Method

$M(n)$ - no. of moves to transfer n disk

$M(1) = 1$

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

$\hookrightarrow M(n-1) = 2(2M(n-2)+1)+1 = 2^2 M(n-2) + (2+1)$

$\hookrightarrow M(n-2) = 2^2(2M(n-3)+1) + (2+1) = 2^3 M(n-3) + (4+2+1)$

$M(1) = 2^{n-1} M(1) + (2^{n-2} + 2^{n-3} + \dots + 4 + 2 + 1)$

$[n-(n-1)] = 2^{n-1}(M(1)) + (2^{n-1} - 1)$

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

$\boxed{M(1) = 2^n - 1}$ → steps required.

- no. of bit in a number:

```
count = 0
while (n>1):
    n = n//2
    count = count + 1.
count += 1.
```

$\log_2(n)$ steps.

$\log_{10}(n) = O(\log_2 n)$

- Matrix Mult :- $O(n^3)$ steps.

Scanned by TapScanner

Date :

Searching in a list:-

```
def naivesearch (v, L):
    for x is L:
        if v == x:
            return(True)
    return(False).
```

worst time complexity - $O(n)$

what if L is sorted:

```
def binarysearch (v, L):
    if L==[]:
        return (False)
    m = len(L) //2
    if v==L[m]:
        return(True)
    if v<L[m]:
        return(binarysearch (v, L[:m]))
    else:
        return (binarysearch (v, L[m+1:]))
```

$O(\log_2 n)$ steps.

$n=0$

$T(0) = 1$

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

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

$= T(n//8) + 3$

$\vdots$

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

$= T(1) + k$, $k = \log_2 n$; $2^k = n$.

$= T(0) + k + 1 = 2 + \log_2 n$.

Scanned by TapScanner

Page :
Date :

Selection_Sort_Technique:-

after sort → Binary search, Median find, Duplicate check, Frequency Table Making.

```
def selectionSort (L):
    n = len(L)
    if n<1:
        return(L)
    for i in range(n):
        mpos = i
        for j in range (i+1, n):
            if (L[j] < L[mpos]):
                mpos = j
        (L[i], L[mpos]) = (L[mpos], L[i])
```

Outer loop n times
inner loop i-n times
$T(n) = (n-1) + (n-2) + \dots (1)$.
$= \frac{n(n-1)}{2}$. $= O(n^2)$.

Insertion_Sort_Technique

create a new list, insert element according to order.

Iterative way

```
def insertion-sort (L):
    n = len(L)
    if n<1:
        return (L).
    for i in range (n).
        # Assume L[:i] is sorted
        # move L[i] to correct pos
        j = i
        while (j>0 && L[j] < L[j-1]):
            (L[j], L[j-1]) = (L[j-1], L[j])
            j = j-1
        # now L[:i+1] is sorted)
    return (L).
```

$T(n) = 0 + 1 + 2 + \dots + (n-1)$
$= n(n-1)/2$
$= O(n^2)$.

Scanned by TapScanner

recursive way:

may arise recursion depth error

```
def Insert(L, v):
    n = len(L)
    if n==0:
        return([v])
    if v >= L[-1]
        return(L+[v])
    else:
        return(Insert(L[:-1], [v])) + L[-1:]
```

$T1(n) = T1(n-1) + 1$

$\Rightarrow T1(n) = n$

```
def Isort(L):
    n = len(L)
    if n<1:
        return(L)
    L = Insert(Isort(L[:-1]), [L[-1]])
    return(L)
```

$TS(n) = TS(n-1) + T1(n-1)$

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

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

$\longrightarrow$ last element $[L[-1]]$

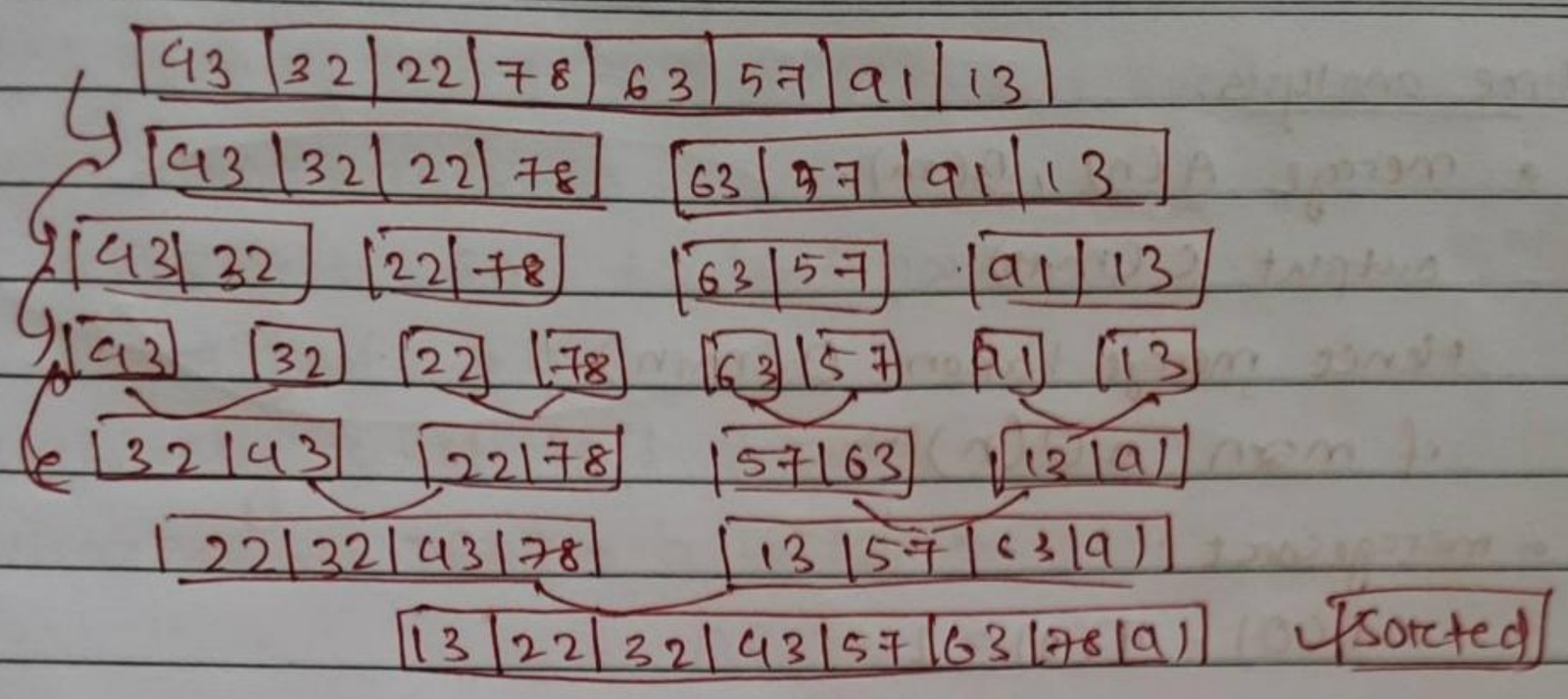
Merege-Sort-Technique (Divide & Conquer)

- Divide the list into two parts. A & B
- Sort both of them ~~A and~~ !
- Combine two sorted list A & B into C:
  - compare first element of A & B
  - Move the smaller of two to C
  - Repeat till no elements left

32 74 89
21 55 64 $\rightarrow$ 21 32 55 64 74 89

Scanned by TapScanner

Page :
Date :



merging part:

```
def merge(A,B):
    (m,n) = (len(A), len(B))
    (c,i,j,k) = ([],0,0,0)
    while (k < m+n):
        if i==m:
            c.extend(B[j:])
            k = k+(n-j)
        elif j==n:
            c.extend(A[i:])
            k = k+(n-i)
        elif A[i]<B[j]:
            c.append(A[i])
            (i,k) = (i+1, k+1)
        else:
            c.append(B[j])
            (j,k) = (j+1, k+1)
    return c.
```

sort part:

```
def mergesort(A):
    n = len(A)
    if n<=1:
        return A.
    L = mergesort(A[:n//2])
    R = mergesort(A[n//2:])
    B = merge(L,R)
    return(B)
```

Scanned by TapScanner

time analysis:

- merge $A(n)$, $B(m)$.

  output $C(m+n)$

  Hence merge takes $O(m+n)$

  if $m \approx n \rightarrow O(n)$.

- mergesort:

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

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

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

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

$= 8T(n/8) + 3n$ $\quad 8(n/8) = 8$

$\vdots$

$2^k T(n/2^k) + kn$

when $k = \log n$

$= 2^{\log_2 n} T(1) + n(\log n)$

$= n + n(\log n) = O(n \log n)$ $\left(< O(n^2)\right)$

Variation

- Union, Intersection, List Difference $(A-B)$

shortcomings
- no obvious way to merge it in that list
- extra storage needed
- more recursion and return

Selection Sort $\rightarrow O(n^2)$ $\rightarrow$ no need of new list

Insertion Sort $\rightarrow O(n^2)$

Merege Sort $\rightarrow O(n \log n)$ $\Big\}$ need a new list.

Scanned by TapScanner

Page :
Date :

Quick-Sort-Technique

→ divide & conquer without merging.

① suppose median is m.
② move all values $\leq m$ to left part L
Right half $\geq m$.
③ recursively sort left & right.
④ Recurrence – $T(n) = 2T(n/2) + n$
$= O(n)$.
• rearrang in single pass

so $T(n) \to O(n \log n)$.

→ To find median we have to sort first. (thats impossible).
Let us take some value in L – pivot
split acc. to pivot.

C.A.R Hoare Algo →

• choose a pivot. (typically the first element).
• do partition acc. of it.

High-level →

less	less	greater	greater	greater	greater	less
43	32	22	78	63	57	91	13

pivot

32	22	13	43	78	63	57	91

sort acc. to recursion.

How to Partition

Four parts :- Pivot Lower Upper Unclassified

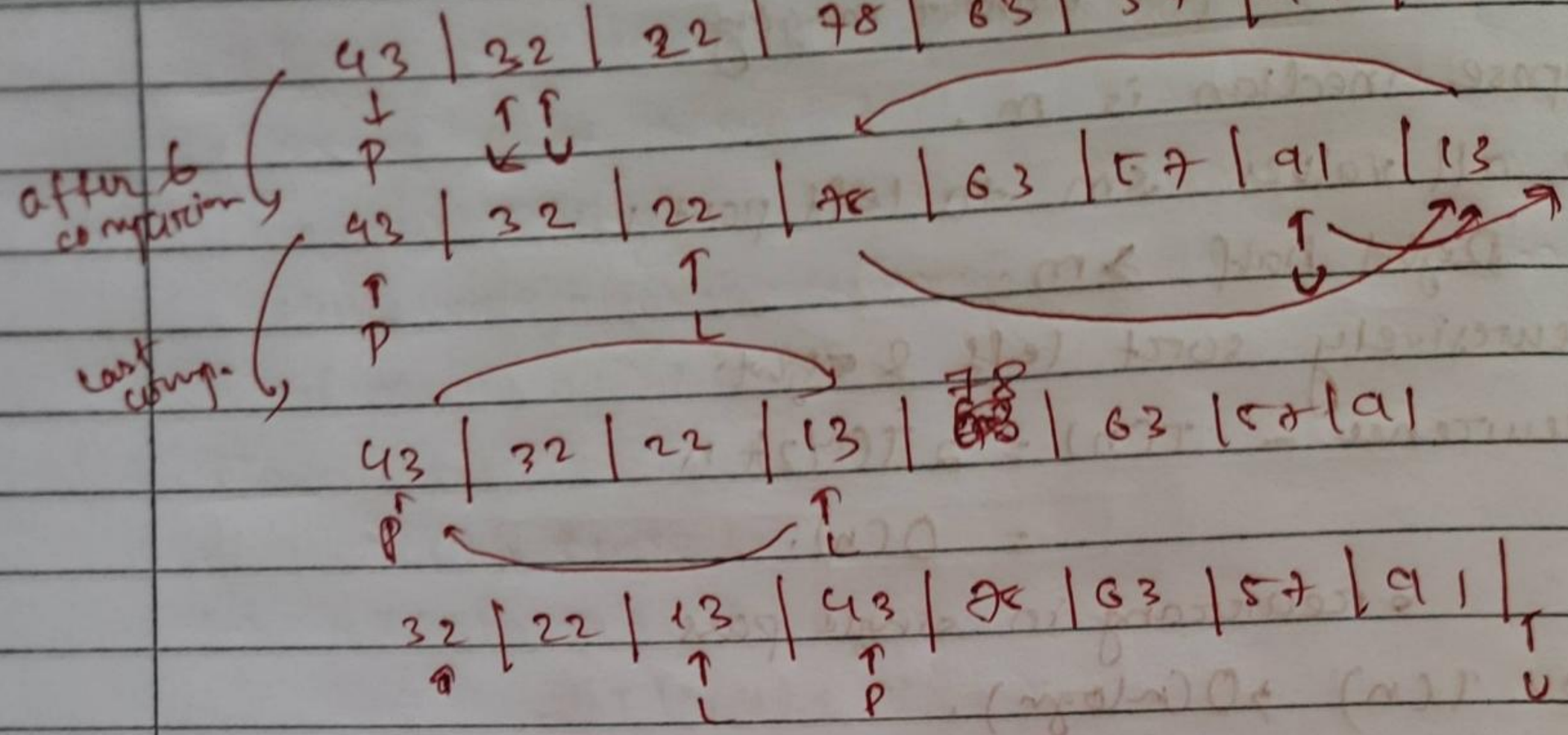
Pivot	Lower	upper	unclassified

check its first element

if > pivot ⇒ extend upper
≤ pivot ⇒ exchange with first element of upper.
and shifts upper by one position.

Scanned by TapScanner

Date :



Code:-

```
def quicksort (L, l, r) :    # sort L[l:r]
    if (r-l ≤ 1):
        return L
    (pivot, lower, upper) = (L[l], l+1, l+1)
    for i in range (l+1, r):
        if (L[i] > pivot):    # extend upper segment
            upper += 1
        else:    # exchange L[i] with start of upper
            (L[i], L[lower]) = (L[lower], L[i])
            # shift both segment
            (lower, upper) = (lower+1, upper+1)
    # move pivot between lower and upper
    (L[l], L[lower-1]) = (L[lower-], L[l])
    lower = lower - 1
    # recursive calls without changing pos. of pivot
    quicksort (L, l, lower)
    quicksort (L, lower+1, upper)
    return(L).
```

Scanned by TapScanner

Page :
Date :

- allows an in place sort.

Analysis:-

partition takes time $O(n)$.

→ if pivot is median, $T(n) = 2T(n/2) + n$
↳ $O(n \log n)$

→ in worst case, pivot is either maxm. or minimum.

partition are of size : $0, n-1$.

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

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

→ Already sorted array : worst case.

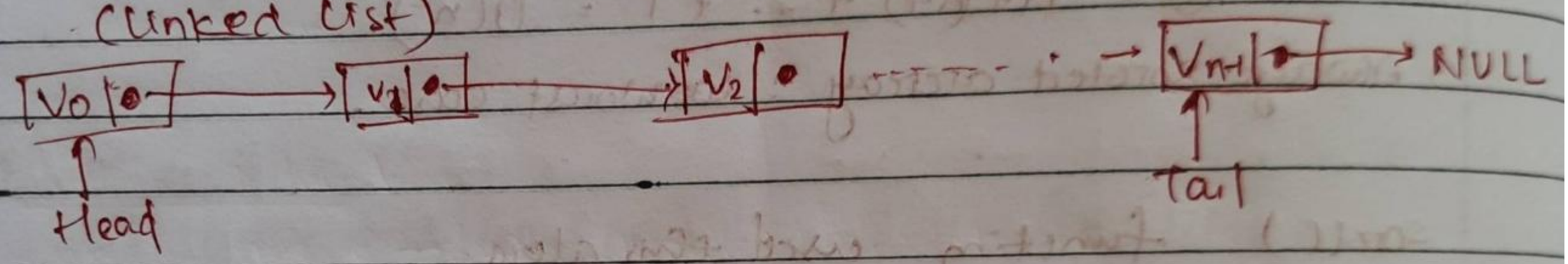
sort() function used this algo.

Scanned by TapScanner

List	Arrays
Flexible length	Fixed size.
Easy to modify structure.	Allocates a contiguous memory
values are scattered in memory	supports random access

Links

- sequence of nodes
- each nodes contains a value and points to the next node (linked list)



- Inserting and deletion is easy via local Plumbing
- to get the value at ith pos., we need to traverse to it which takes time O(i).

Arrays

Index	A[0]	A[1]	A[2]	...	A[n-1]
Value	V_0	V_1	V_2		V_{n-1}

- Random Access
- Inserting and Deleting is expensive with worst case time O(n)

Ur's only

Exchange value (swap)	→ O(n) for lists	constant time for Arrays
Delete A[i] / Insert A[i] after A[i]	→ constant time for list if we're at A[i]	O(n) for Arrays

Scanned by TapScanner

Implementing List

Date :

Python class Node

List is sequence of node

self.value → stores value

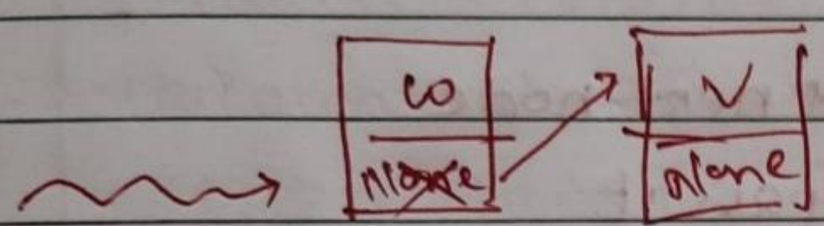
self.next → points to next

L1 = Node → empty

L2 = Node(5) → singleton.

L1.isempty() = True

L2.isempty() = False

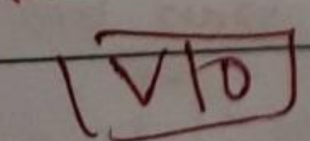



```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return
    def isempty(self):
        if self.value == None:
            return True
        else:
            return False
    def append(self, v)
        if self.isempty():
            self.value = v
        elif self.next == None:
            self.next = Node(v)
        else:
            self.next.append(v)
            # recursion
        return
```

- Iterative Appending

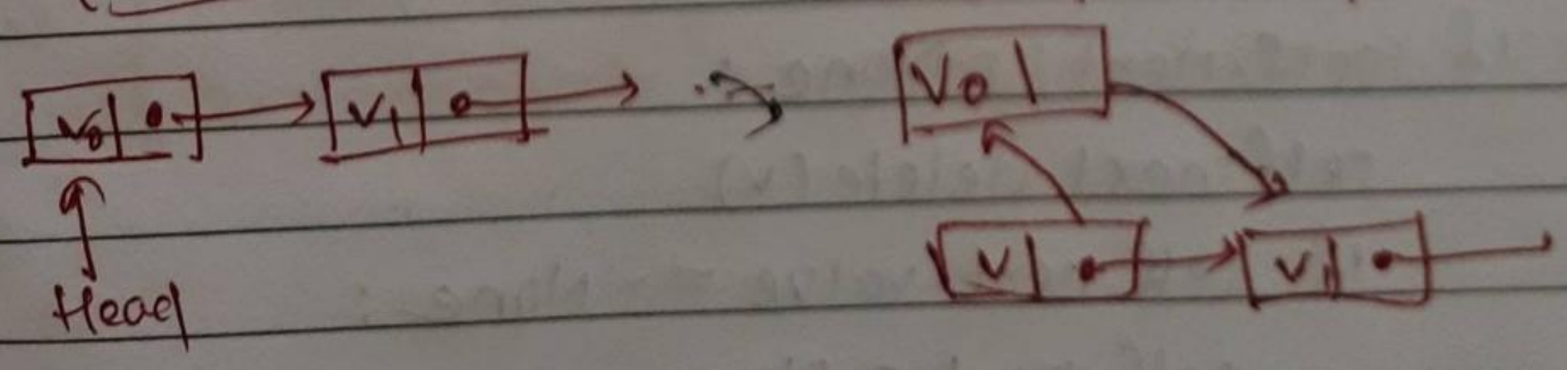
```
def appendI(self, v):
    # iterative way.
    if self.isempty():
        self.value = v
        return
    temp = self
    while temp.next != None:
        temp = temp.next
    temp.next = Node(v)
```

Insert at Beginning



Swap values of v0 and v

now do the plumbing



Scanned by TapScanner

```
def insert(self, v):
    if self.isempty():
        self.value = v
        return
    newnode = Node(v)

    (self.value, newnode.value) = (newnode.value, self.value)
    (self.next, newnode.next) = (newnode, self.next).
    return
```

Delete a value V

- scan list for v — look ahead at next node
- if next node value is v, bypass it
- can not by pass first node
  So ~~copy~~ swap it with second node value
  now bypass the second.

```
def delete(self, v):
    # recursive delete.
    if self.isempty():
        return
    if self.value == v:
        self.value = None
        if self.next != None:
            self.value = self.next.value
            self.next = self.next.next
        return
    else:
        if self.next != None:
            self.next.delete(v)
            if self.next.value == None:
                self.next = None
    return
```

Scanned by TapScanner

Page :
Date :

Python List

Lists in python are NOT implemented as linked list, rather than it is done as array.

- assign a fixed block when you create it.
- double size if it overflows.

Arrays

Arrays are useful in matrices.

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \rightarrow [[0,1],[1,0]]$$

*need to be careful in N-D array

code
```
zerolist = [0,0,0]
zeromatrix = [zerolist, zerolist, zerolist]
zeromatrix[1][1] = 1
print(zeromatrix)
```
↳ [[0,1,0],[0,1,0],[0,1,0]] → why this happen

Because zeromatrix is pointing to same zerolist everytime if one copy changes other copies also changes.

[0,0,0] = zerolist
zeromatrix = [↑, ↑, ↑]

- Mutability Arises different values.

How To Do Then ?

→ use list comprehension like

code
```
zeromatrix = [[0 for i in range(3)] for j in range(3)]
```

→ or use Numpy.

code
```
import numpy as np.
zeromatrix = np.zeros(shape = (3,3))
```

Scanned by TapScanner

Page :
Date :

Numpy Arrays

→ if provide basic array

```
import numpy as np
zeromatrix = np.zeros(shape=(3,3))
```

→ can create array from any sequence type

```
newarray = np.array([1,0],[0,1])
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

→ arange is equivalent of range for lists

```
row2 = np.arange(5)
```

→ can operate on matrix as a whole

```
C = 3*A + B
C = np.matmul(A, B)
```

Scanned by TapScanner

Page :
Date :

Dictionaries in Python

allows access through keys.

- collection of key-value pairs
- random access

underlying storage is an array.

keys have to mapped to $\{0, 1, \dots n-1\}$.

use HASH Function

- $h: S \to X$, maps a set of values S to small range of integeres $X = \{0, 1, 2, \dots n-1\}$.
- typically $|X| \ll |S|$; so there will be collisions, $h(s) = h(s')$, $s \neq s'$
-
- SHA-256 is an industry standard hashing function whose range is 256 bits.
  used to hash large files - avoid uploding duplicate files to cloud storage

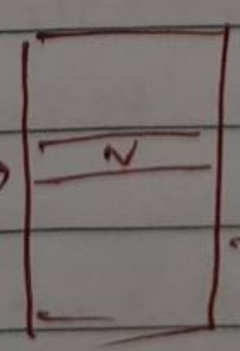
Hash Table

- An array A of size n combined with function h.
- h maps keys to $\{0, \dots n-1\}$
- so when we create an entry for key k, $A[h[k]]$ will be unused.

what if place is preoccupied?

$d[k] = v$

$k - h(k) \to$ some no. is 0 to n-1



array.

Dealing with collision →
next page.

Scanned by TapScanner

Page :
Date :

Dealing with collision

- Open addressing (closed hashing)
  - probe a sequence of alternate slots in same array
- Open hashing.
  - each pos. in array points to a list of values insert into list for given slot.

So Dictionary keys in Python must be immutable.

Scanned by TapScanner

Page :
Date :

Graph

$G = (V, E)$

V is vertices or nodes

E is set of edges.

$E \subseteq V \times V$ binary relation.

directed graph

$(v, v') \in E$ does not imply $(v', v) \in E$. (Teacher → course)

undirected graph

$(v, v') \in E \Rightarrow (v', v) \in E$ (friend ↔ friend)

they have the same edge

Path – it is a sequence of vertices connected by edges

for $1 \leq i < k$, $(v_i, v_{i+1}) \in E$

it does not visit a vertex twice.

Walk – A sequence which re-visit a vertex is usually called walk.

Reachability → if there is a path, it is reachable.

connected graph → all nodes is reachable from every other nodes

Graph Colouring

$G = (V, E)$; set of colour C.

colouring is a fun. $c: V \to C$ s.t. $(u, v) \in E \Rightarrow c(u) \neq c(v)$

Smallest set of colours c ?

↳ four color theorem for planar graphs

Vertex Cover

marking v covers all edges from v.

mark smalest subset of V to cover all edges.

Independent Set

subset of vertices such that no two are connected by edges

Matching

$G = (V, E)$, undirected graph

A matching is a subset $M \subseteq E$ of mutually disjoint ~~sets~~. edges

Scanned by TapScanner

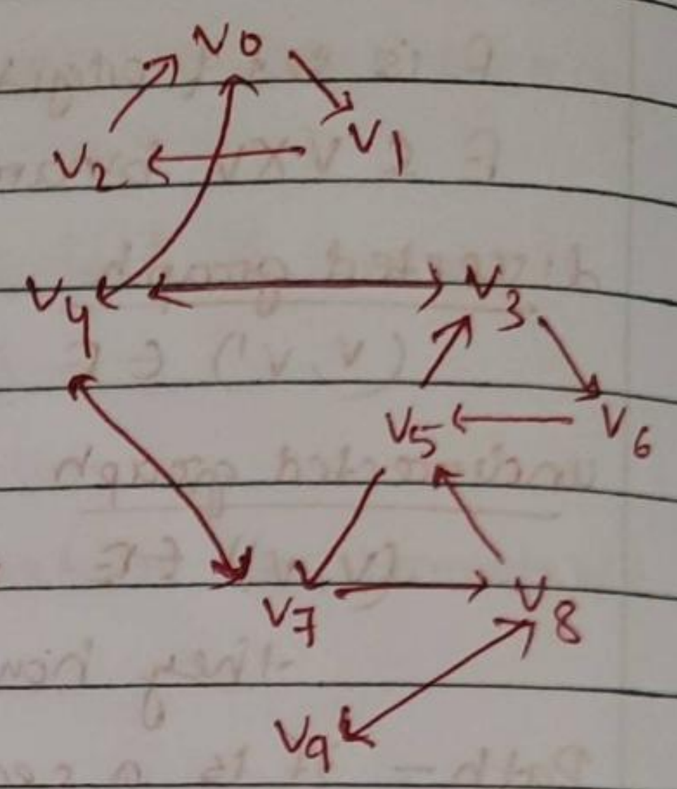
size in n^2
undirected $|E| \le n(n-1)/2$
directed $|E| \le n(n-1)$
overcally $|E| << n^2$

Page :
Date :

Adjency Matrix

rown & columns numbered 0 to n-1

$A[i][j] = 1$ if $(i,j) \in E$



edges = [(0,1), (0,4), (1,2), (2,0), (3,4),
(3,6), (4,0), (4,3), (4,7), (5,3),
(5,7), (6,5), (7,4), (7,8), (8,5),
(8,9), (9,8)]

```
import numpy as np.
A = np.zeros(shape=(10,10))
for (i,j) is edges:
    A[i,j] = 1.
```

- Finding neighbours of a node:-

```
def neighbours(A,i):
    nbrs = []
    (rows, colu) = Amat.shape
    for j in range(cols):
        if A[i,j] == 1:
            nbrs.append(j)
    return nbrs.
```

Adjency List using Dictionaries

```
AList = {}
for i in range(10):
    AList[i] = []
for (i,j) in edges:
    AList[i].append(j)
```

Directed Graph rows → columns is the direction

degree:-

in the graph degree(6) = 2.
but indegree(6) = 1 and outdegree(6) = 1

degree(5) = 4. indegree(5) = 2 outdegree(5) = 2

Scanned by TapScanner

Page :
Date :

Checking reachability.

(explore a path till it dies out, then backtrack)

use BFS or DFS
(Propage in layers)

BFS

assume $V = \{0, 1, \dots n-1\}$

visited : $V \to \{True, False\}$ tells us whether node is visited or not

initially visited(v) = False. $\forall v \in V$

maintain a sequence of visited vertices yet to be explorced.

- use a queue structure (FIFO)

Construction and working of a queue.

```
class queue:
    def __init__(self):
        self.queue = []
    def addq(self, v):
        self.queue.append(v)
    def delq(self):
        v = None
        if not self.isempty():
            v = self.queue[0]
            self.queue = self.queue[1:]
        return(v)
    def isempty(self):
        return(self.queue == [])
    def __str__(self)
        return(str(self.queue))
q = queue().
for i in range(3):
    q.addq(i)                    # q = [0, 1, 2]
print(q.isempty()).              False
for i in range(3).
    q.delq()                     # q = []
print(q.isempty())               True.
```

Scanned by TapScanner

how to do ?

initially visited(v) = false $\forall v \in V$

queue is empty

start from vertex j

visited (j) = True

Add j to queue.

remove and explore i at head of queue

- for each edge (i, j), if visited(j) is False
  - visited (j) = True
  - append j to queue

stop when queue is empty.

breadth first search

matrix way

```
def BFS (A, v):
    (rows, cols) = A.shape
    visited = {}
    for i in range (rows):
        visited [i] = False
    q = queue()
    visited [v] = true
    q.addq(v)
    while (not q.isempty()):
        j = q.delq()
        for k in neighbours(A, j)
            if (not visited [k]):
                visited [k] = True
                q.addq(k)
    return (visited).
```

Scanned by TapScanner

Date :

complexity

$G = (V, E)$

$|V| = n$ $|E| = m$. m varies from $n-1$ to $n(n-1)/2$

(sum of degrees $= 2m$).

with adjency matrix - $O(n^2)$

with adjency list - $O(n+m)$

we can use level instead of visited.

[level = -1 not visited and increase it acc. to parent node

help in finding shortest path.

BFS list way

```
def BFSListPath (Alist,v):
    (visited, parent) = ({}, {})
    for i in Alist.keys():
        visited [i] = False
        parent [i] = -1
    q = queue().
    visited [v] = True
    q.addq (v)
    while (not q.isempty()):
        j = q.delq()
        for k in Alist[j]:
            if (not visited[k]):
                visited[k] = True
                parent[k] = j
                q.addq (k)
    return (parent, visited)
```

after running this code (start from 7)

node	visited	parent	node	visited	parent
0	True	4	6	True	5
1	True	0	7	True	-1
2	True	0	8	True	7
3	True	4	9	True	8
4	True	7			
5	True	7			

path for 2 to 7 → 2 - 0 - 4 - 7.

Scanned by TapScanner

Date :

DFS

Stared with i, visit unvisited j.
suspend i, explore j.
contine till deadend, backtrack to nearest suspended vertex
suspended are stored using stack DS (LIFO)

Using Adjency Matrix:-

```
def DFSInit(Amat):
    (rows, cols) = Amat.shape
    (visited, parent) = ({}, {})
    for i in range(rows):
        visited[i] = False
        parent[i] = -1
    return(parent, visited)
def DFS(Amat, visited, parent, v):
    visited[v] = True
    for k in neighbours(Amat, v):
        if (not visited[k]):
            parent[k] = v
            (visited, parent) = DFS(Amat, visted, parent, k)
    return (parent, visited)
```

Using an Adjency List:-

```
def DFSInitList(AList):
    (visited, parent) = ({}, {})
    for i in AList.keys():
        visited[i] = False
        parent[i] = -1
    return (visited, parent)
```

```
def DFSlist(AList, visited, parent, v):
    visited[v] = True
    for k in AList[v]:
        if (not visited[k]):
            parent[k] = v
            (visited, parent) =
                DFSlist(AList, visited, parent
    return (parent, visted)
```

Scanned by TapScanner

Page :
Date :

Complexity -

using matrix $O(n^2)$
list $O(m+n)$

Summary :

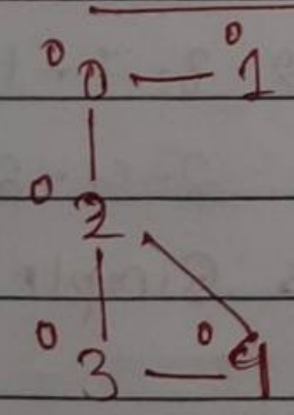
* Paths discovered by DFS are not shortest unless BFS.
* It is more informative than BFS in some cases.

Application

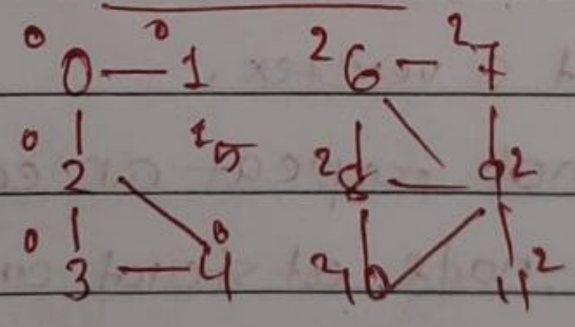
BFS discovery shortest path in terms of no. of edges
connected graph → every vertex is reachable for other every vertex

connected comp. → maximal subset of vertices that are connected
trivial comp → isolated vertices

Connected



Disconnected



Algo →

① assign vertex component no.

② start for 0.
↳ all visited no. assigned no. 0.

③ pick smallest unvisited node assign it 1.
↳ all visited through this should be assigned 1.

④ Repeat it upto every node is visited.

⑤ One component no. → connected
more than 1 component no. → disconnected.

Scanned by TapScanner

Page :
Date :

BFS way code -

```
def Components (Alist):
    component = {}
    for i in Alist.keys():
        component[i] = -1.
    (compid, seen) = (0, 0)
    while seen < max(Alist.keys()):
        startv = min([i for i in Alist.keys()
                        if component[i] == -1])
        visited = BFSList(Alist, startv)
        for i in visited.keys():
            if visited[i]:
                seen = seen + 1
                component[i] = compid
        compid += 1.
    reeturn(component)
```

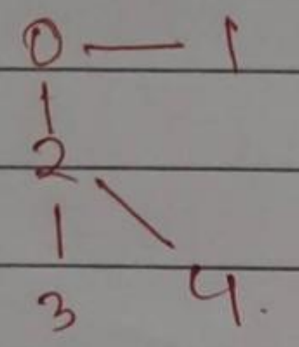
→ cycle → path that start at end at same vertex 4-8-9-4 ✓
  it may repeat a vertex. 2-3-7-10-6-7-
  but it must not repeat an edge. 2-4-2 ✗
→ cycle with one repeat node at start and end is Simple cycle.
→ graph is acyclic → it has no cycle.

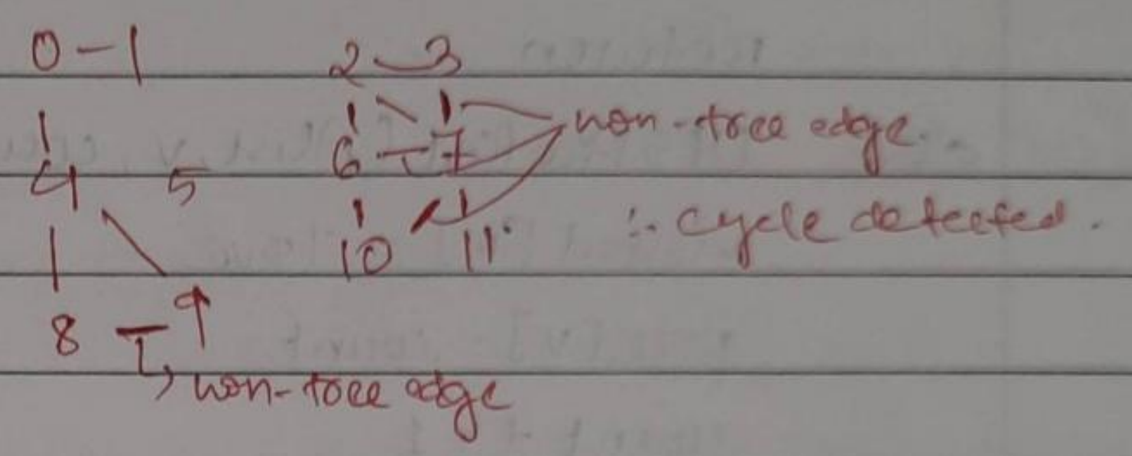
Scanned by TapScanner

Page :
Date :

Edges explored by BFS form TREE.

if there is a edge in a graph which is not part of tree, it forms a cycle. Because the edge must be discarded previously as its target node is encountered before





no non-tree edge

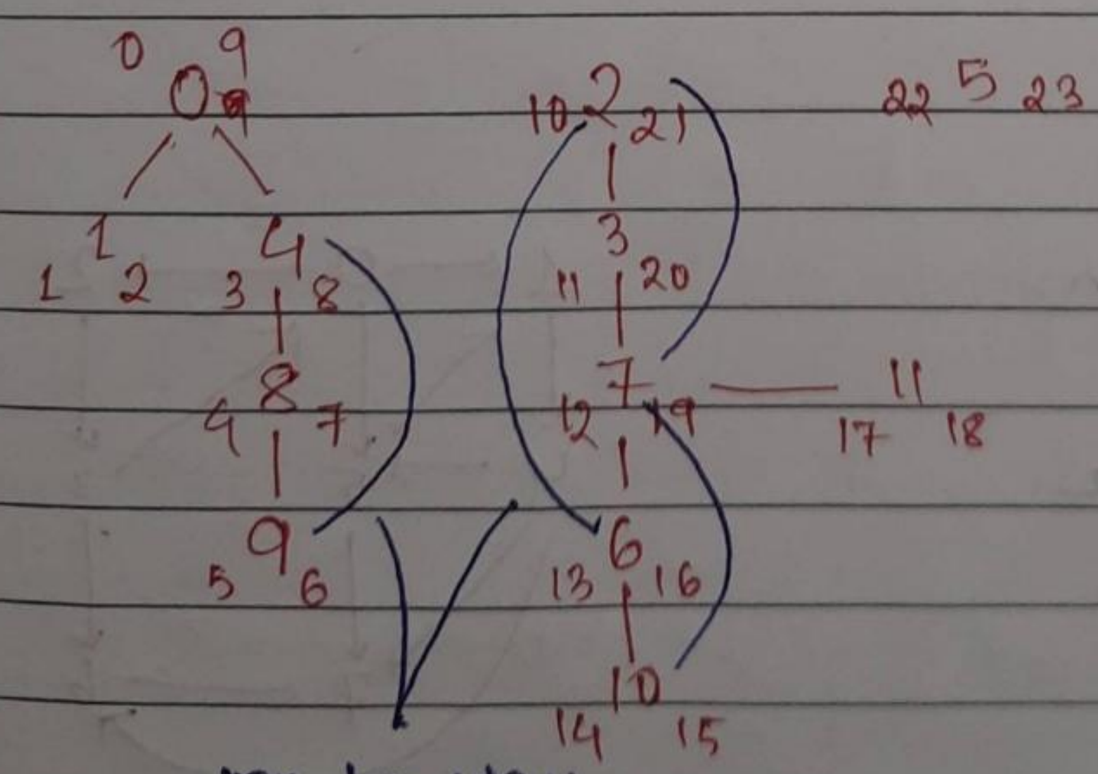
DFS Tree

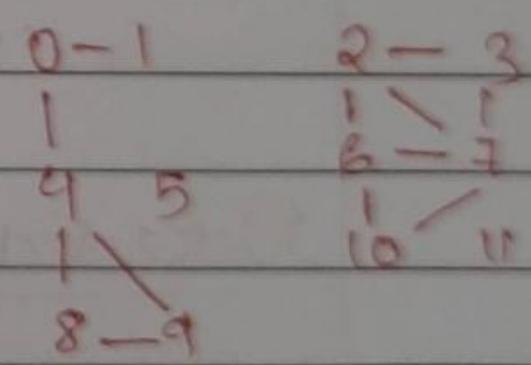
DFS counter. maintained.

Increment each time when start and finish exploring a node.

each vertex have 2 no.s (pre) and (post).

vertex
(pre) (post)





Scanned by TapScanner

Date :

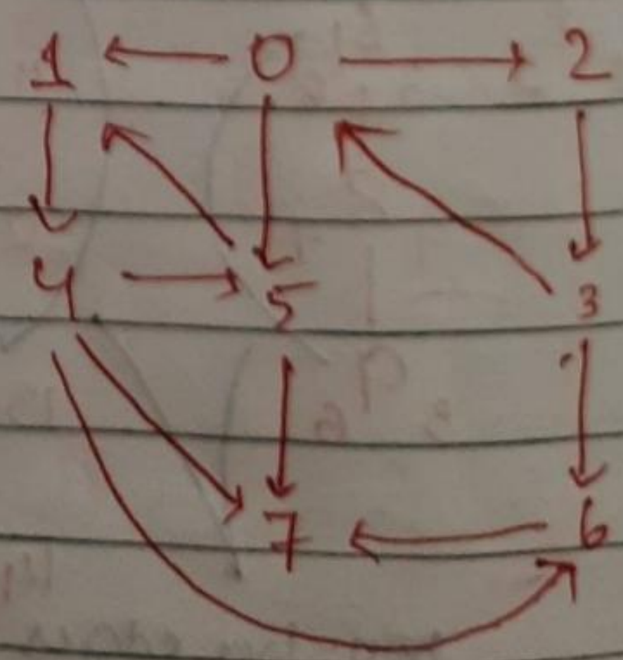
```
(visited, Pre, Post) = ({}, {}, {})
def DFSInitPrePost (Alist):
    #initialization.
    for i in Alist.keys():
        visited [i] = false
        pre [i], post [i] = (-1, -1)
    return
def DFSPrePost (Alist, v, count):
    visited [v] = True
    pre [v] = count
    count += 1
    for k in Alist [v]:
        if not visited [k]:
            #parent [k] = v
            count = DFSPrePost(Alist, k, count)
    post [v] = count
    count += 1.
    return (count)
```

Directed cycles :-

cycle must follow direction.

0→2→3→0 cycle

0→5→1←0 not cycle



Types of non-tree edges

- forward
- back
- cross

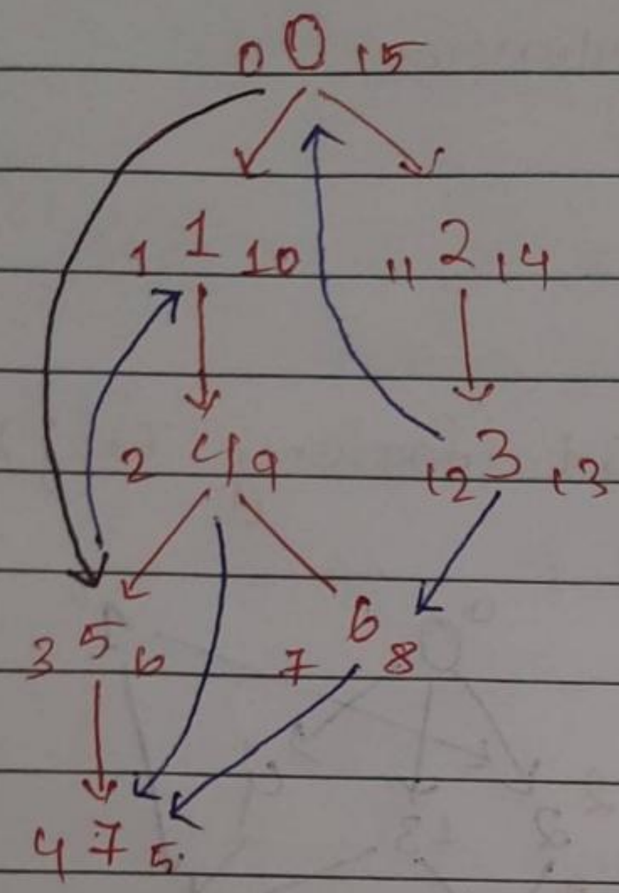
Only back edges corresponds to a cycle

Scanned by TapScanner

Page :
Date :

~~DAG (Directed Acyclic Graphs)~~

DFS Tree



- Forward Edge (u, v)
  [pre(u), post(u)] contains [pre(v), post(v)]
- Back Edge (u, v)
  [pre(v), post(v)] contains [pre(u), post(u)]
- Cross edge (u, v)
  [pre(u), post(u)] and [pre(v), post(v)] disjoint.

- Strongly Connected Component (SCC) vertices are those vertices which is accessible to other node from both end.
  Like if i and j are (SCCs), then i is reachable from j and vice-versa.

- DFS numbering used to identify articulation point (cut vertices) and bridges (cut edges).

Scanned by TapScanner

DAGs

Page :
Date :

$G = (V, E)$ → directed graph without directed cycle.

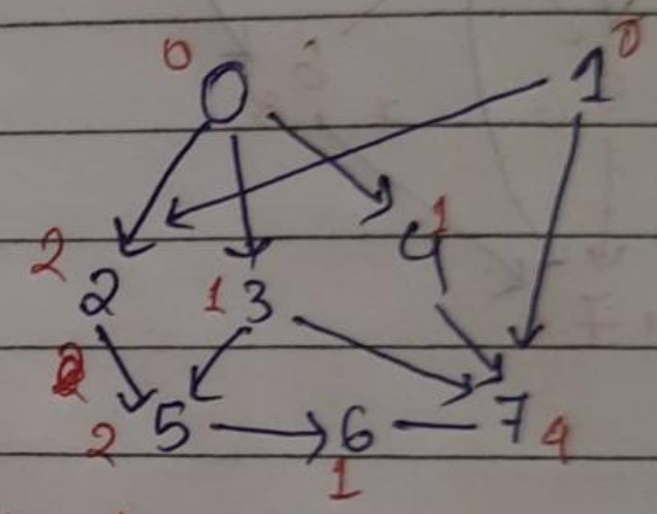
- ↳ Topological Sorting
- ↳ Longest Paths

[no cycle is possible].

natural way to represent dependencies

Topological Sorting.

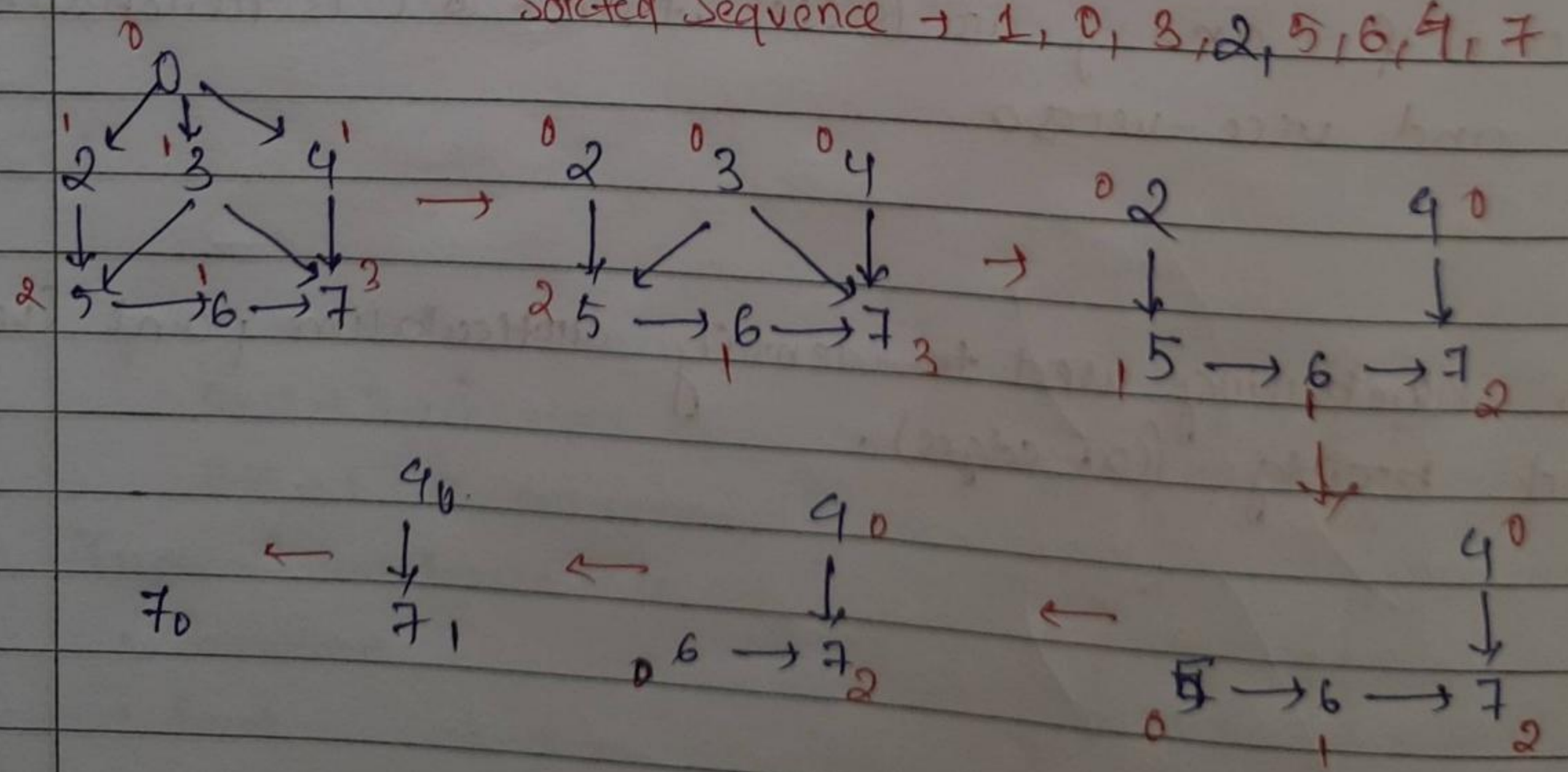
Enumerate $V = \{0, 1, \dots, n-1\}$ s.t. for any $(i, j) \in E$, i appears before j.



vertex with no dependencies has indegree$(v) = 0$.

- → pick vertex with indegree 0 and eliminate it
- → eliminate associate edges.
- → update indegrees of left DAG.
- → continue until all vertices is eliminated.

Sorted Sequence → 1, 0, 3, 2, 5, 6, 4, 7



Page :
Date :

```
def toposort(AMat):
    (rows, cols) = Amat.shape
    indegree = {}
    toposortlist = [].

O(n²)  for c in range(cols):
           indegree[c] = 0
           for r in range(rows):
               if AMat[r, c] == 1:
                   indegree[c] += 1.

O(n²)  for i in range(rows):
O(n)       j = min([k for k in range(cols) if indegree[k] == 0])
           toposortlist.append(j)
           indegree[j] -= 1.
O(n)       for k in range(cols):
               if AMat[j, k] == 1
                   indegree[k] -= 1.

    return (toposortlist)
```

Overall $O(n^2)$.

In using Adjency list: (maintaining zeroindegree q)

- initializing indegree $O(m+n)$

  - updating indegree: amortised $O(m)$

Overall $O(m+n)$

Scanned by TapScanner

Page :
Date :

Longest Path

indegree (i) = 0 ⇒ longest-path-to(i) = 0

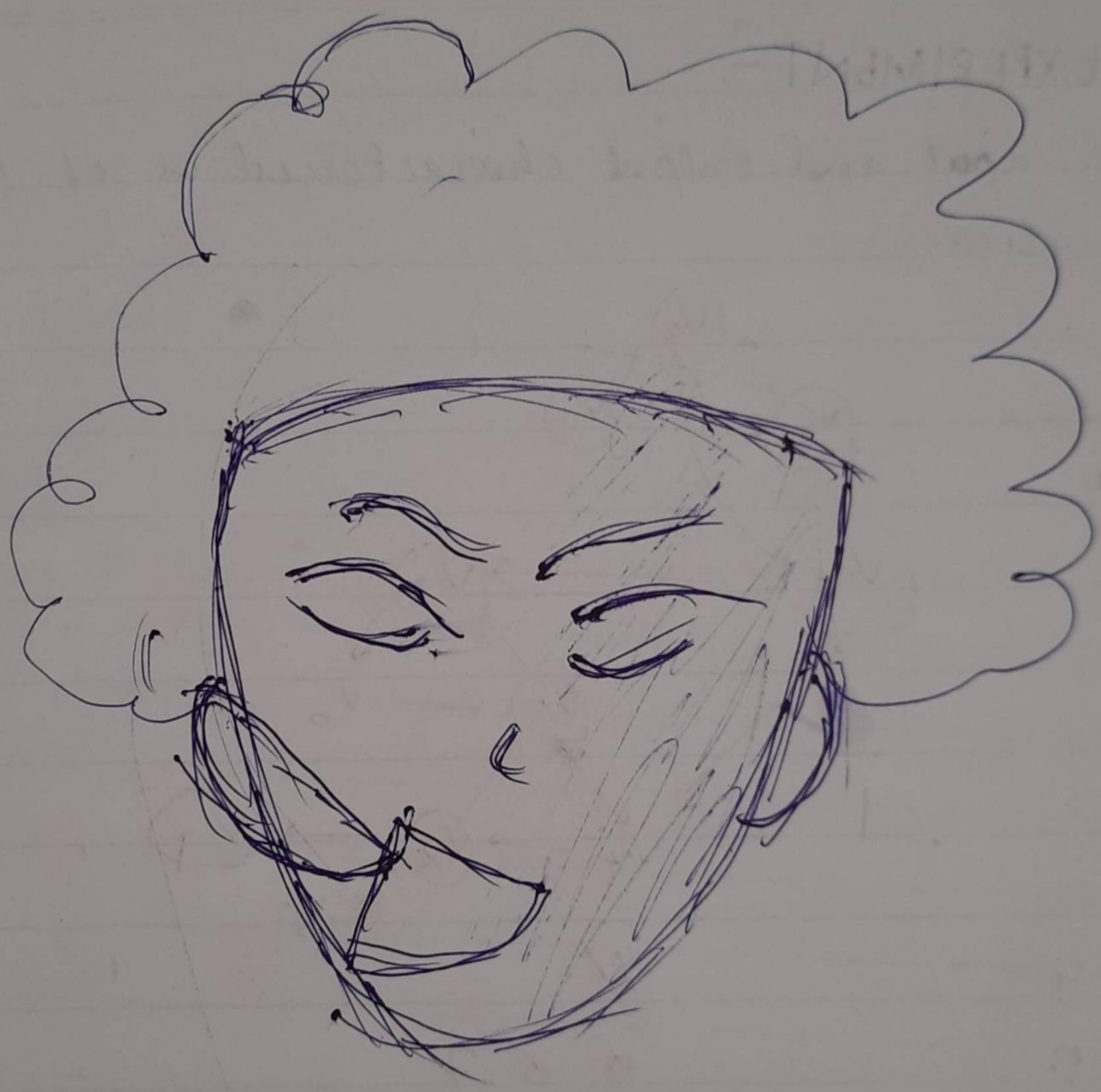
indegree (i) > 1 ⇒ longest-path-to(i) =

1 + max {longest path to (j) | (j, i) ∈ E}

$O(m+n)$.

it can be calculated parallel to topological sort

Scanned by TapScanner



Scanned by TapScanner