W-1

Exception Handling

Page :
Date :

Exception Handling

- Syntax Error → can't handle the error

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

Handle an Exeptim Code

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

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

example:-

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

a) updating

if batter[s] exists.

```
scores[c].append(s)
```

if new batter

```
scores[c] = [s]
```

traditional way

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

Using except

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

Page :
Date :

GCD - reecursion Way -

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

* this take much longer steps fore 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
```

upper bound

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

Page :
Date :

- $100n+5$ in $O(n^2)$

  for $n \geq n_0 = (5)$ $c = 101$

  i.e. $100n + 5 \leq 101n^2 \quad \forall n \geq 5$.

  where,

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

  $105n \leq 105n^2$

  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, $cn^2 \geq n^3$

  $\Rightarrow cn^2 > n^3$

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 $O(g_m(n))$; then as a whole algorithm will take $O(\max(g_A(n), g_m(n)))$ time.

i.e. least efficient phase is upper bound of the whole algorithm.

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 f 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 $\Theta(n^2)$

upper bound

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

lower bound

$n(n-1)/2 \geq n^2/2 - (n/2 \times n/2) \geq n^2/4$ for $n \geq 2$

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

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

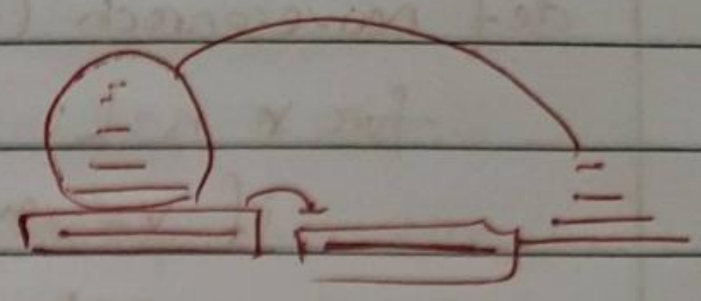
tight bound is the optimal algo. for a problem.

Page :
Date :

Calculating Complexity
(recursive / iterative)

- **Towers of Hanoi**

move n-1 from A to C
move largest to B
move n-1 from C to B as A (transitive)



Recurrence Method

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

$M(1) = 1$

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

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

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

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

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

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

$\boxed{M(1) = 2^n - 1}$ → 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.

Date :

Searching in a list:-

```
def naivesearch (v, L):
    for x is L:                    worst time complexity - O(n)
        if v == x:
            return(True)
    return(False).
```

what if L is sorted:

```
def binarysearch (v, L):
    if L==[]:
        return (False)          O(log2 n) steps.
    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:]))
```

n-0

$T(0) = 1$

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

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

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

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

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

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

Page :
Date :

Selection_Sort_Technique:-

Applications → 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) + -- (1).
= n(n-1)/2 . = O(n²).

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 + -- + (n-1)
= n(n-1)/2
= O(n²).

recursive way:

```
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:]
```

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

⇒ T(n) = n

I may arise (recursion) depth error

```
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) + T(n-1)

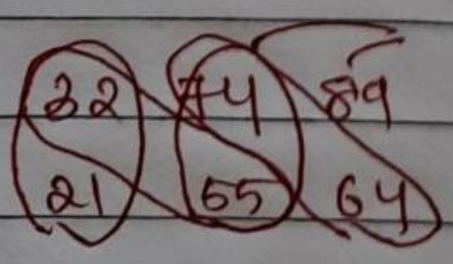
= 1 + 2 + ... + n-1

= n(n-1)/2 = O(n²)

L[-1] → last element

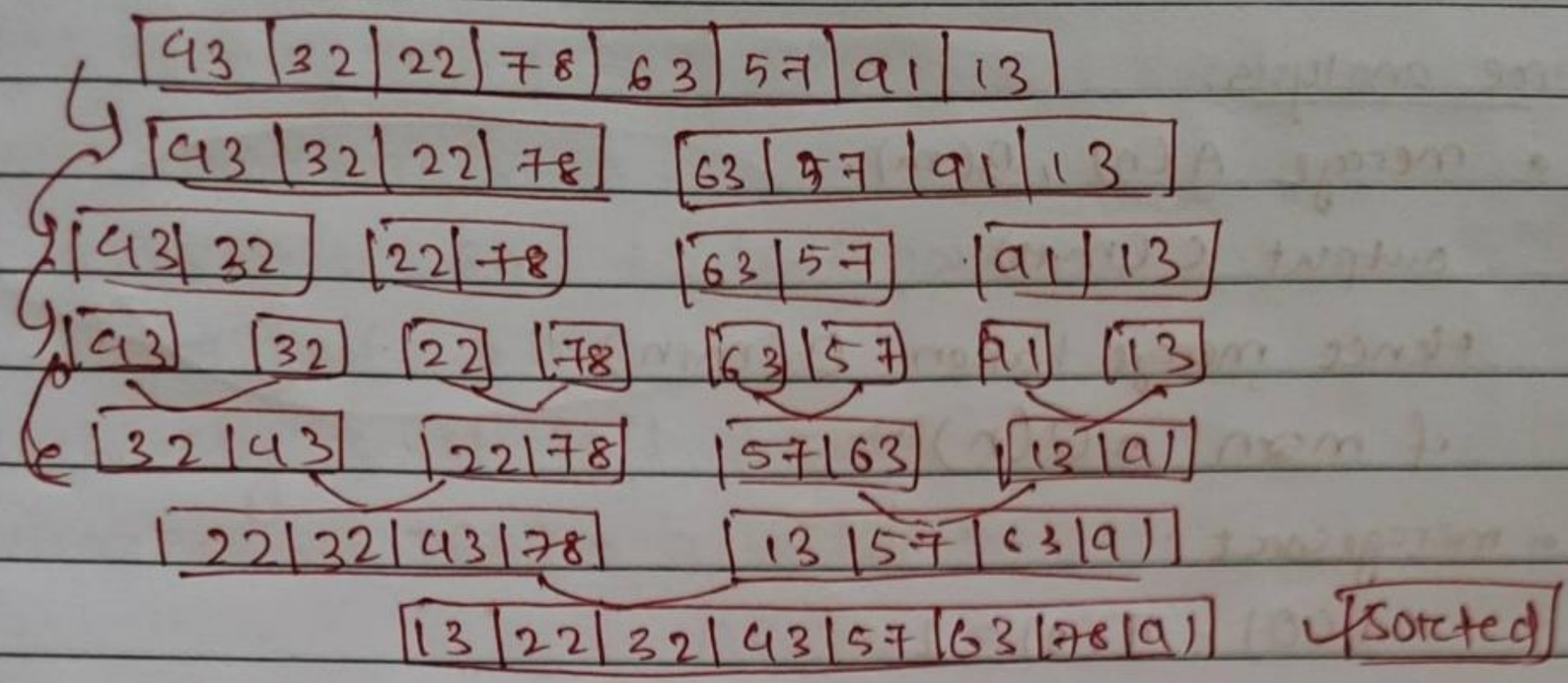
Merge Sort Technique (Divide & Conquer)

- Divide the list into two parts A & B
- Sort both of them
- 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



→ 21 32 55 64 74 89

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)
```

time analysis:

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

output C(m+n)

Hence merge taken O(m+n)

if m≈n → 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 8(n/8)=8

⋮ $2^k T(n/2^k) + kn$

when k = log n

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

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

Variation

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

short coming

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

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

Insertion Sort → $O(n^2)$

Merege Sort → $O(n \log n)$ → need a new list

Page :
Date :

Quick-Sort-Technique

→ divide & conquer without merging.

① suppose median is m.
② move all values ≤m to left part L
Right half ≥m.
③ recursively sort left & right.
④ Recurrence - T(n) = 2 T(n/2) + n
= O(n).
• rearranging in single pass

so T(n) → O(nlogn).

→ To find median we have to sort first. (thats impossible).
Lets 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 greath greath greath greath less

43	32	22	98	63	57	91	13

pivot

32	22	13	43	98	63	57	91

sort acc. to recursion.

How to partition

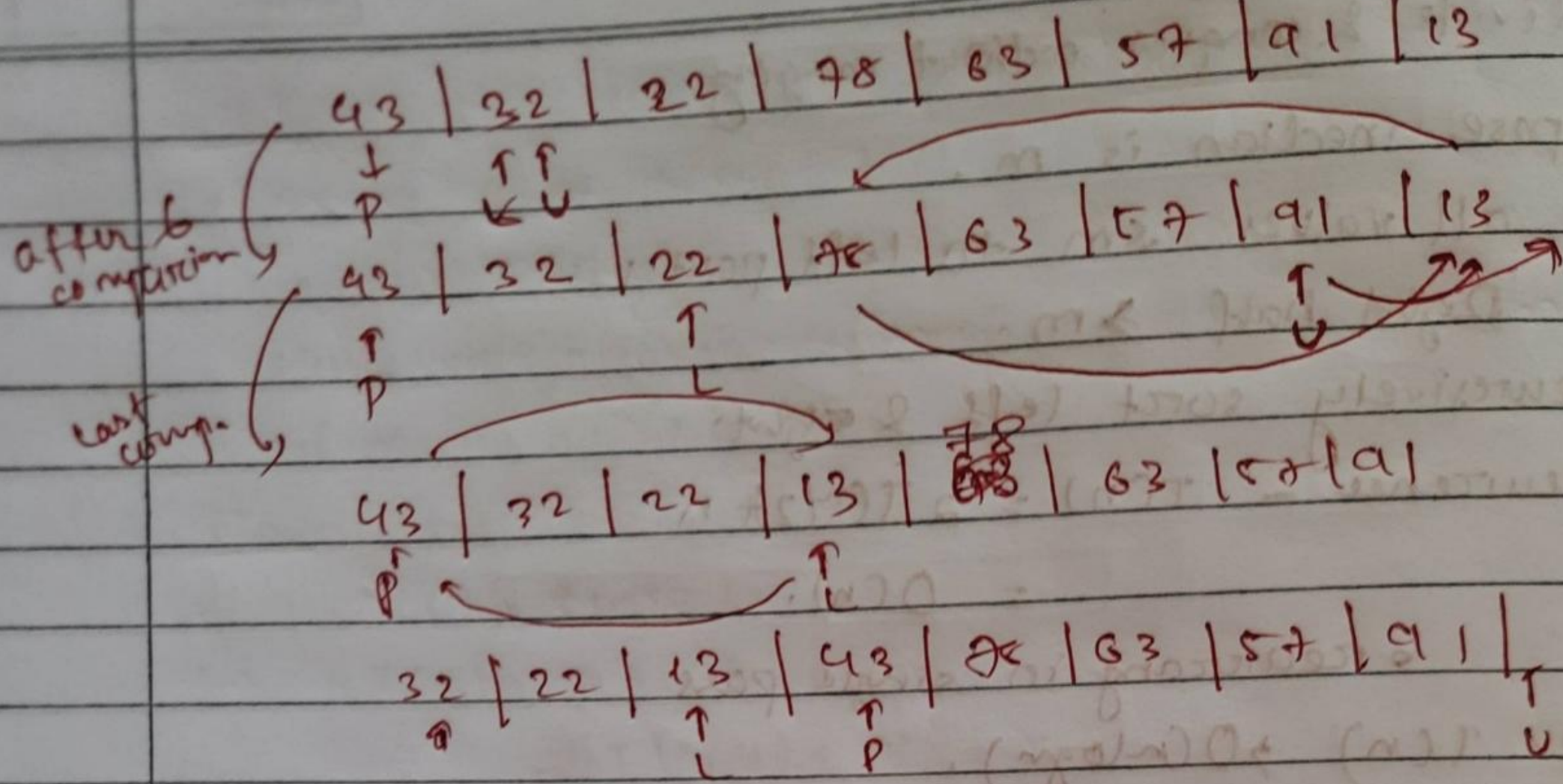
Four parts :- Pivot Lower Upper Unclassified

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.

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-1], L[l])
    lower = lower - 1
    # recursive calls without changing pos. of pivot
    quicksort (L, l, lower)
    quicksort (L, lower+1, upper)
    return (L).
```

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 : $0, n-1$.

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

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

→ Always sorted array : worst case.

sort() function used this algo.

List

flexible length

easy to modify structure.

values are scattered in memory

Arrays

fixed size.

Allocates a contiguous memory

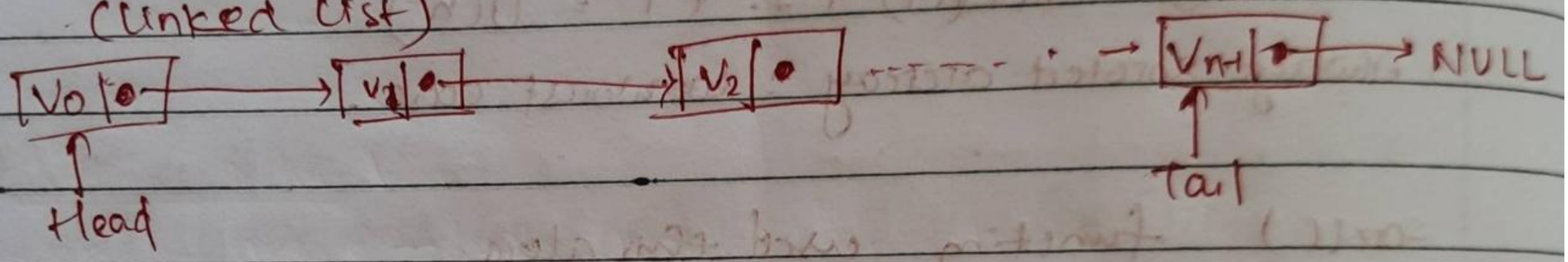
supports random access

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)

	Lists	Arrays
Exchange Value (Swap)	O(n)	Constant time for Arrays
Delete A[i] / Insert AV after A[i]	Constant time for List if we're at A[i]	O(n) for Arrays

Implementing List

Date :

Python class Node

List is sequence of node

self.value → stores value

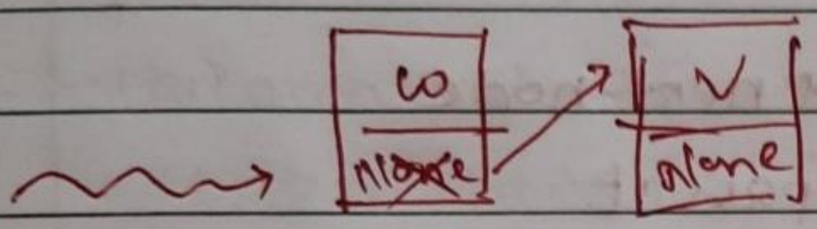
self.next → points to next

L = Node → empty

L2 = Node(5) → singleton.

L1.isempty() = True

L2.isempty() = False

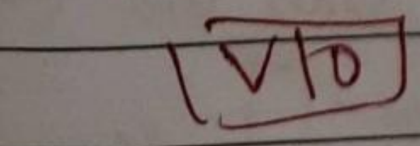



```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return
    def isempty(self):
        if self.value == None:
            return True
        else:
            return False
    def append(self,v)
        if self.isempty():
            self.value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

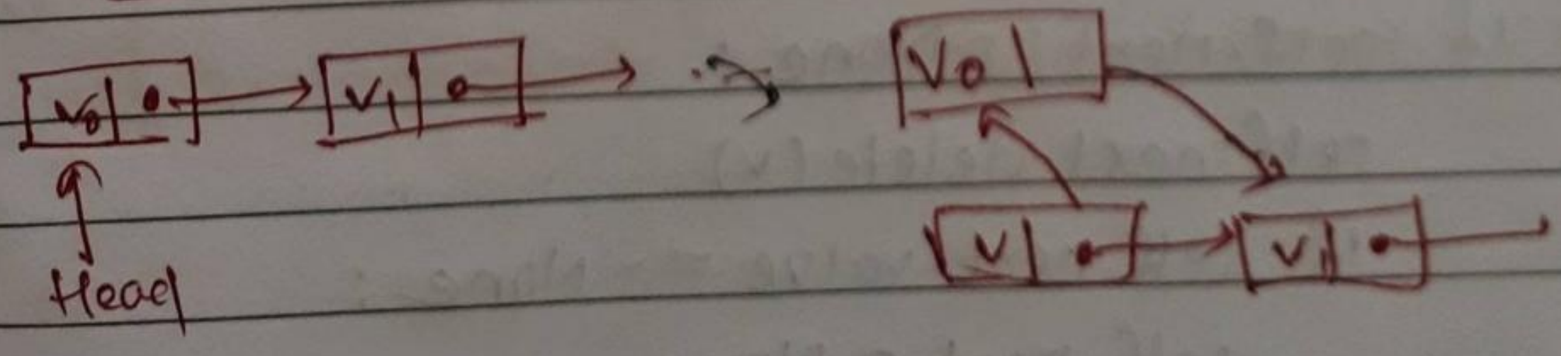
- Iterative Appending

```
def append1(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



```
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
```

Python List

Page :
Date :

Lists in python are NOT implemented as linked list, rather than if 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.

- Mutability Alises different values.

How To Do Then ?

→ use list comprehensim 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))
```

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)
```

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 $\{0, 1, \dots n-1\}$.

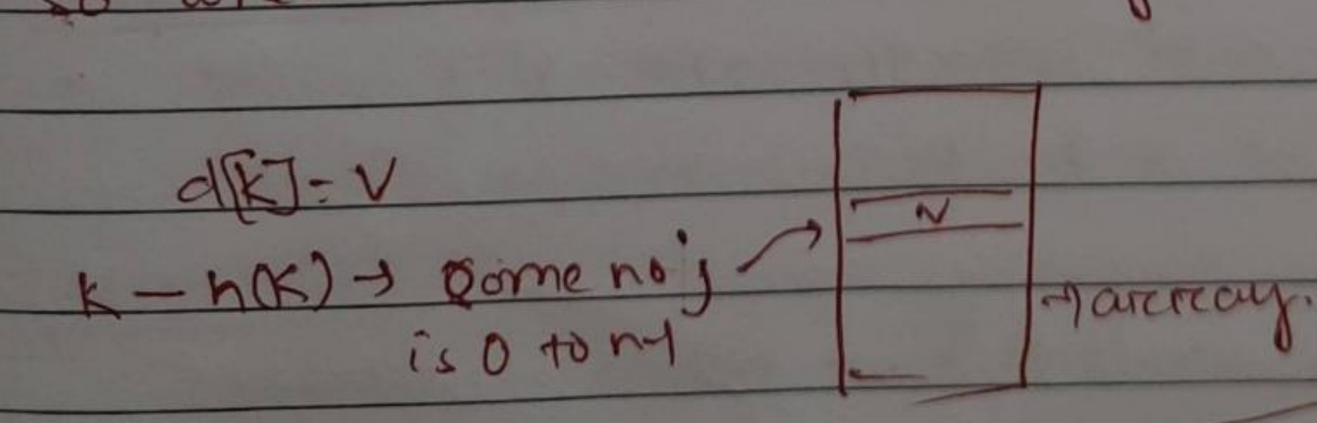
use HASH Function

- $h: S \to X$, 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 standared 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 fuction h.
- h maps keys to $\{0, \dots n-1\}$
- so when we create an entry for key k, A[h[k]] will be unused.



Dealing with collision →
next page.

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.

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 $(v', v) \in E$. (Teachers → courese)

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

Page :
Date :

size in n^2

undirected $|E| \leq n(n-1)/2$

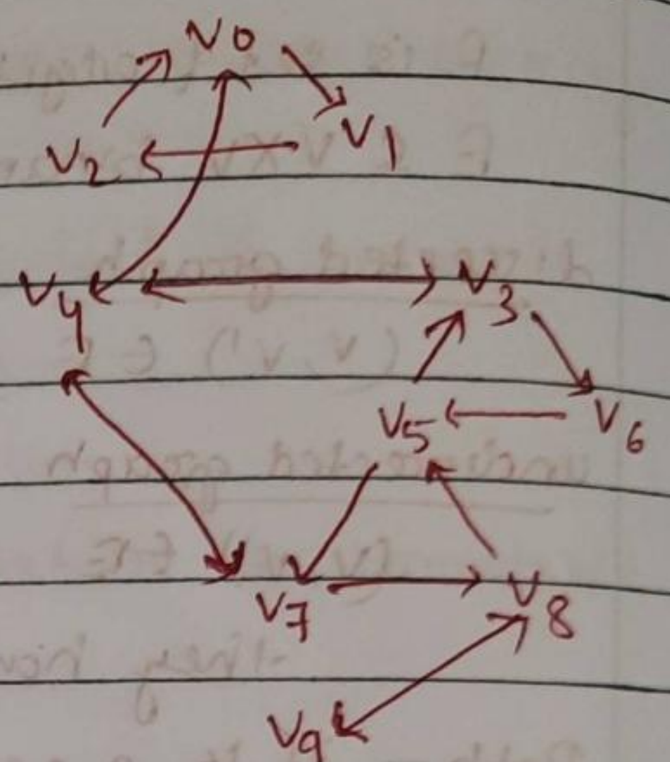
directed $|E| \leq n(n-1)$

overcally $|E| << n^2$

Adjency Matrcix

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.

Adjancy List using Dictionaries

AList = {}

for i in range (10):

AList[i] = []

for (i,j) in edges:

AList[i].append (j)

- Finding neighbours of a node: —

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

Directed Graph rows → columns is the direction

degree:-

in the graph degree (G) = 2.

but indegree (G) = 1 and outdegree (G) = 1

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

Page :
Date :

checking reachability.

→(explore a path till it dies and then backtrack)

use BFS or DFS

↳(propage in layers)

BFS

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

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

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

maintain a sequence of visted vertices yet to be explorced.

- use a queue structuree (FIFO)

```
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.
```

Construction and working of a queue.

how to do ?

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

queue is empty

started 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 ia 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).
```

Date :

complexity

$G = (V, E)$

$|V| = n$ $|E| = m$. m varay fm $n-1$ to $n(n-1)/2$

(sum of degrees $= 2m$).

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

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

we can use level instead of visited.

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

help in finding shortest path.

BFS List way

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

after running this code (stard from 7)

node	visited	parcent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7

node	visited	parcent
6	True	5
7	True	-1
8	True	7
9	True	8

path for 2 to 7 → 2-0-4-7.

Date :

DFS

stared with i, visit unvisited j.
suspend i, explorce 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] = Tove
    for k in neighbours(Amat, v):
        if (not visited[k]):
            parent[k] = v
            (visited, parent) = DFS(Amat, visited, 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] = Tove
    for k in AList[v]:
        if (not visited[k]):
            parent[k] = v
            (visited, parent) =
                DFSList(AList, visited, parent
    return (parent, visted)
```

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

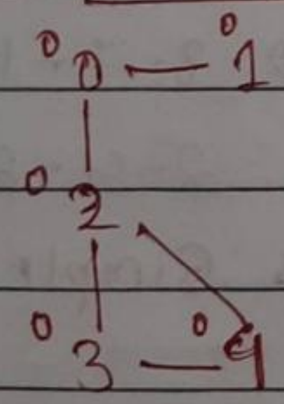
BFS discovery shortest path in terms of no. of edges

connected graph → every vertex is reachable for other every node

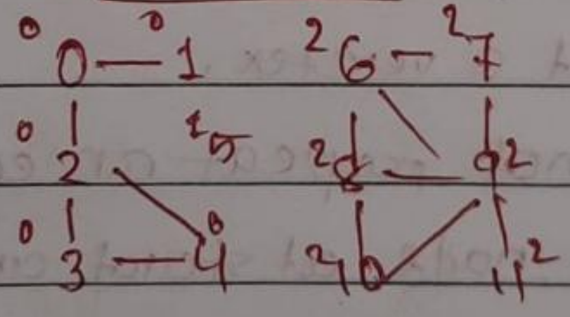
connected comp. → maximal subset of vertices that are connected

trivial comp → isolated vertices

connected



Disconnected



Algo →

① assign vertex component no.

② stared 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.

Page :
Date :

BFS way code -

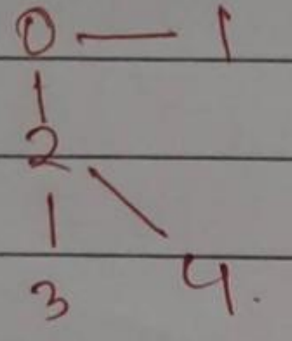
```
def components (Alist):
    component = {}
    for i in Alist.keys():
        component[i] = -1.
(compid, seen) = (0, 0)
while seen < max (Alist.keys()):
    startv = min ([i for i in Alist.keys()
                   if component[i] == -1])
    visited = BFSList (Alist, startv)
    for i in visited.keys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

→ cycle → path that start at end at same vertex 4-8-9-4 ✓
  it may repeat a veretex. 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.

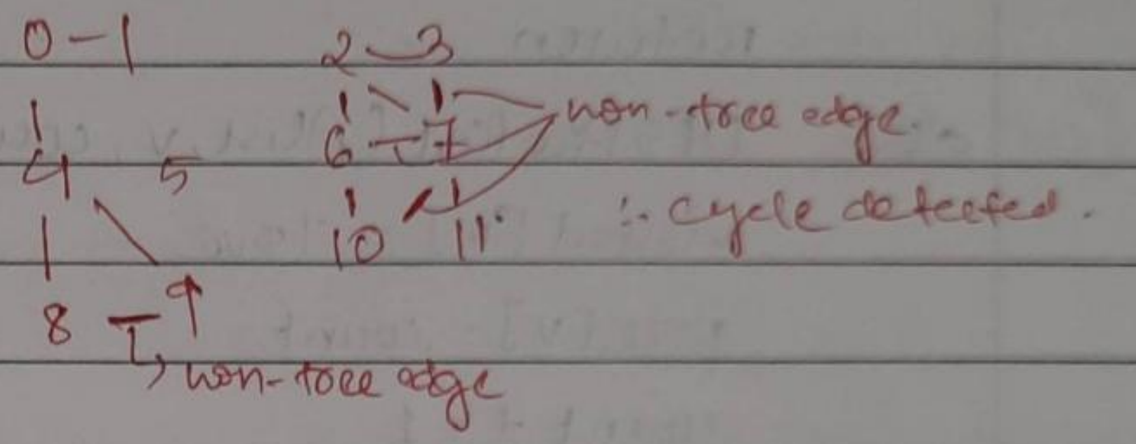
Page :
Date :

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



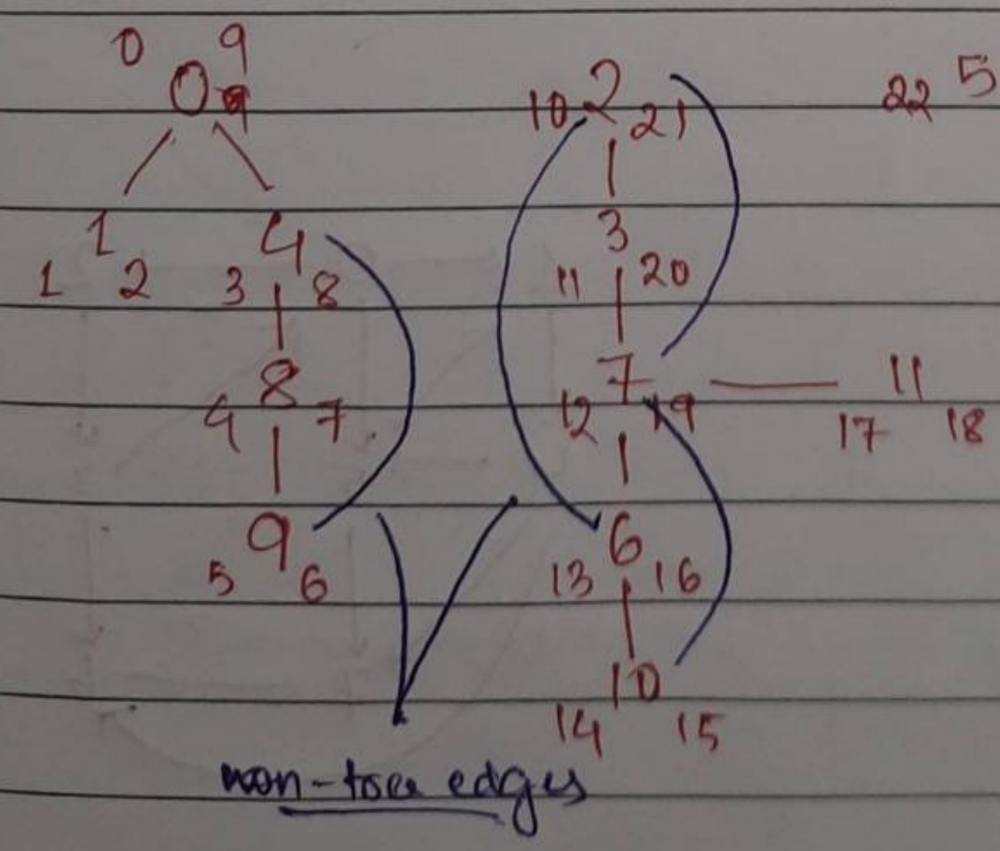
DFS Tree

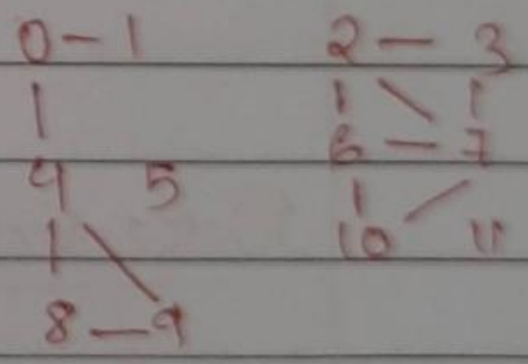
DFS counter. maintained.

Increment each time when start and finish exploring a node.

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

vertex
(pre) (post)





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.

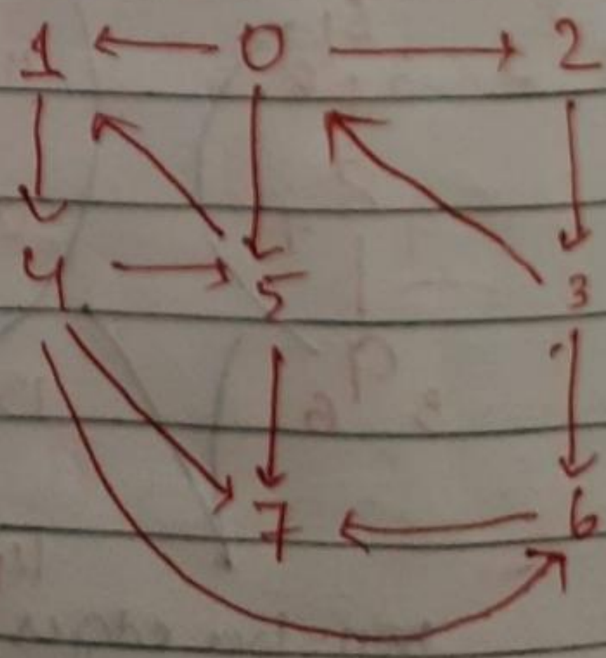
0→2→3→0 cycle

0→5→1←0 not cycle

Types of non-tree edges

- forward
- back
- cross

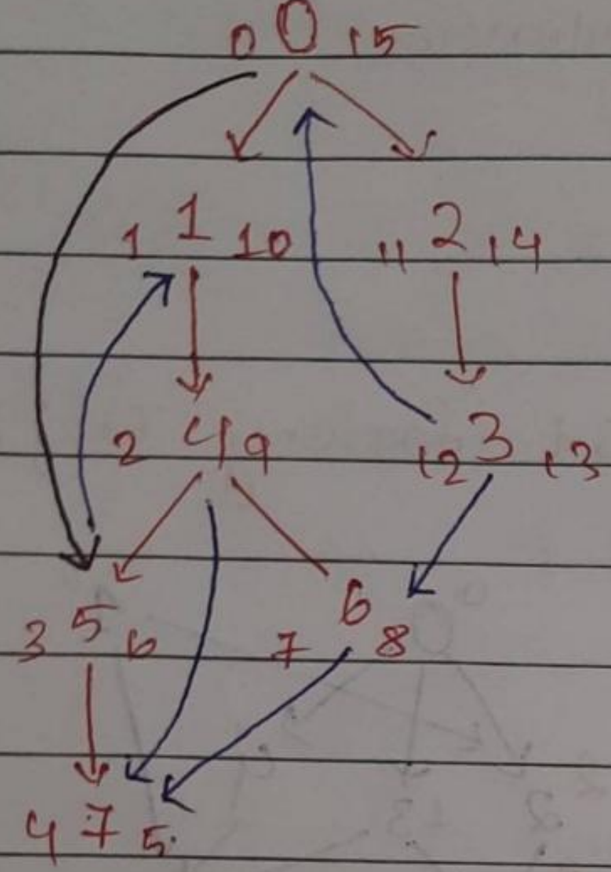
only back edges corresponds to a cycle



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 Components (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).

DAGs

Page :
Date :

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

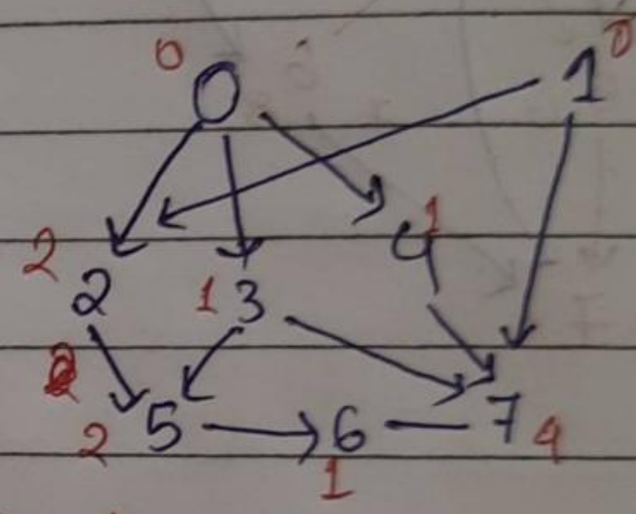
- Topological Sorting
- Longest Paths

[no cycle is possible].

natural way to represent dependencies

Topological Sorting.

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



vertex with no dependecies
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 = [].

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

    for i in range(rows):                    O(n²)
        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 (toposort list)
```

Overall $O(n^2)$.

In using Adjency list: (maintaining zero indegree Q)

- initializing indegree $O(m+n)$

→ updating indegree: amortised $O(m)$

Overall $O(m+n)$

Page:
Date:

Longest Path

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

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

$1 + \max\{\text{longest path to } (j) \mid (j,i) \in E\}$

$O(m+n)$.

it can be calculated parallel to topological sort

Balanced search tree (AVL Tree) - : - Greedy Algorithm

Binary search tree

- find(), insert() and delete() all walk down a single path
- Worst-case: height of the tree An unbalanced tree with n nodes may have height O(n)

AVL Tree

- Balanced trees have height ***O(logn)***
- Using rotations, we can maintain height balance
- Height balanced trees have height ***O(logn)***
- find(), insert() and delete() all walk down a single path, take time ***O(logn)***
- Minimum number of node $S(h)=S(h-2)+S(h-1)+1$
- Maximum number of nodes 2^h-1

Greedy Algorithm

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion
- Never go back and revise an earlier decision
- Drastically reduces space to search for solutions
- Greedy strategy needs a proof of optimality
- Example :
    - Dijkstra's
    - Prim's
    - Kruskal's
    - Interval scheduling
    - Minimize lateness
    - Huffman coding

Algorithm

1. Sort all jobs which based on end time in increasing order.
2. Take the interval which has earliest finish time.
3. Repeat next two steps till you process all jobs.
4. Eliminate all intervals which have start time less than selected interval's end time.
5. If interval has start time greater than current interval's end time, at it to set. Set current interval to new interval.

Analysis

- Initially, sort n bookings by finish time - ***O(nlogn)***
- Single scan***, O(n)***
- overall ***O(nlogn)***

```
#Interval Scheduling:
def tuplesort(L, index):
    L_ = []
    for t in L:
        L_.append(t[index:index+1] +
                  t[:index]+t[index+1:])
    L_.sort()

    L__ = []
    for t in L_:
        L__.append(t[1:index+1] +
                   t[0:1]+t[index+1:])
    return L__

def intervalschedule(L):
    sortedL = tuplesort(L, 2)
    accepted = [sortedL[0][0]]
    for i, s, f in sortedL[1:]:
        if s > L[accepted[-1]][2]:
            accepted.append(i)
    return accepted
```

Minimize Lateness

Algorithm

1. Sort all job in ascending order of deadlines
2. Start with time t = 0
3. For each job in the list
    1. Schedule the job at time t
    2. Finish time = t + processing time of job
    3. t = finish time
4. Return (start time, finish time) for each job

```
from operator import itemgetter

def minimize_lateness(jobs):
    schedule =[]
    max_lateness = 0
    t = 0

    sorted_jobs = sorted(jobs,key=itemgetter(2))

    for job in sorted_jobs:
        job_start_time = t
        job_finish_time = t + job[1]

        t = job_finish_time
        if(job_finish_time > job[2]):
            max_lateness =  max (max_lateness,
                                 (job_finish_time-job[2]))
        schedule.append((job[0],job_start_time,
                         job_finish_time))

    return max_lateness, schedule
```

Analysis

- Sort the requests by D(i) — ***O(nlogn)***
- Read all schedule in sorted order — ***O(n)***
- overall ***O(nlogn)***

Huffman Coding

Algorithm

1. Calculate the frequency of each character in the string.
2. Sort the characters in increasing order of the frequency.
3. Make each unique character as a leaf node.
4. Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.
5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies.
6. Insert node z into the tree.
7. Repeat steps 3 to 5 for all the characters.
8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.

Analysis

- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency
- Store frequencies in an array
- Linear scan to find minimum values
- $|A|=k$, number of recursive calls is k–1
- Complexity is $O(k^2)$
- Instead, maintain frequencies in an heap
- Extracting two minimum frequency letters and adding back compound letter are both $O(\log k)$
- Complexity drops to ***O(klogK)***

```
1
2  class Node:
3      def __init__(self,frequency,symbol = None,left = None,right = None):
4          self.frequency = frequency
5          self.symbol = symbol
6          self.left = left
7          self.right = right
8
9  # Solution
10
11 def Huffman(s):
12     huffcode = {}
13     char = list(s)
14     freqlist = []
15     unique_char = set(char)
16     for c in unique_char:
17         freqlist.append((char.count(c),c))
18     nodes = []
19     for nd in sorted(freqlist):
20         nodes.append((nd,Node(nd[0],nd[1])))
21     while len(nodes) > 1:
22         nodes.sort()
23         L = nodes[0][1]
24         R = nodes[1][1]
25         newnode = Node(L.frequency + R.frequency, L.symbol + R.symbol,L,R)
26         nodes.pop(0)
27         nodes.pop(0)
28         nodes.append(((L.frequency + R.frequency, L.symbol + R.symbol),newnode))
29
30     for ch in unique_char:
31         temp = newnode
32         code = ''
33         while ch != temp.symbol:
34             if ch in temp.left.symbol:
35                 code += '0'
36                 temp = temp.left
37             else:
38                 code += '1'
39                 temp = temp.right
40         huffcode[ch] = code
41     return huffcode
```

Union-Find data structure

- A set S partitioned into components {C1, C2, . . . , Ck }
  - Each s ∈ S belongs to exactly one Cj
- Support the following operations
  - MakeUnionFind(S) — set up initial singleton components {s}, for each s ∈ S
  - Find(s) — return the component containing s
  - Union(s,s') — merges components containing s, s 0

```
#Naïve Implementation of Union-Find
class MakeUnionFind:
    def __init__(self):
        self.components = {}
        self.size = 0
    def make_union_find(self,vertices):
        self.size = vertices
        for vertex in range(vertices):
            self.components[vertex]=vertex
    def find(self,vertex):
        return self.components[vertex]
    def union(self,u,v):
        c_old = self.components[u]
        c_new = self.components[v]
        for k in range(self.size):
            if Component[k] == c_old:
                Component[k] = c_new
```

```
#Improved Implementation of Union-Find
class MakeUnionFind:
    def __init__(self):
        self.components = {}
        self.members = {}
        self.size = {}
    def make_union_find(self,vertices):
        for vertex in range(vertices):
            self.components[vertex] = vertex
            self.members[vertex] = [vertex]
            self.size[vertex] = 1
    def find(self,vertex):
        return self.components[vertex]
    def union(self,u,v):
        c_old = self.components[u]
        c_new = self.components[v]
        '''Always add member in components which
           have greater size'''
        if self.size[c_new] >= self.size[c_old]:
            for x in self.members[c_old]:
                self.components[x] = c_new
                self.members[c_new].append(x)
                self.size[c_new] += 1
        else:
            for x in self.members[c_new]:
                self.components[x] = c_old
                self.members[c_old].append(x)
                self.size[c_old] += 1
```

Complexity

- MakeUnionFind(S) — $O(n)$
- Find(i) — $O(1)$
- Union(i,j) — $O(n)$
- Sequence of m Union() operations takes time $O(mn)$

Complexity

- MakeUnionFind(S) — $O(n)$
- Find(i) — $O(1)$
- Union(i,j) — $O(logn)$

Improved Kruskal's using algorithm using Union-find:

Complexity

- Tree has $n-1$ edges, so $O(n)$ Union() operations
- $O(nlogn)$ amortized cost, overall
- Sorting E takes $O(mlogm)$
  - Equivalently $O(mlogn)$, since $m \leq n^2$
- Overall time, $O((m+n)logn)$

```python
#Improved Kruskal's algorithm using Union-find
def kruskal(WList):
    (edges,TE) = ([],[])
    for u in WList.keys():
        edges.extend([(d,u,v) for (v,d) in WList[u]])
    edges.sort()
    mf = MakeUnionFind() #Given on Page1
    mf.make_union_find(len(WList))
    for (d,u,v) in edges:
        if mf.components[u] != mf.components[v]:
            mf.union(u,v)
            TE.append((u,v,d))
        '''We can stop the process if the size becomes
           equal to the total number of vertices'''
        # Which represent that a spanning tree is completed
        if mf.size[mf.components[u]]>= mf.size[mf.components[u]]:
            if mf.size[mf.components[u]] == len(WList):
                break
        else:
            if mf.size[mf.components[v]] == len(WList):
                break
    return(TE)
```

Priority Queue

Need to maintain a collection of items with priorities to optimize the following operations

- **delete max( )**
  - Identify and remove item with highest priority
  - Need not be unique
- **insert( )**
  - Add a new item to the list

Implementing Priority Queues

One dimensional :

- Unsorted list
  - insert( ) is ***O(1)***
  - delete_max( ) is ***O(n)***
- Sorted list
  - delete_max( ) is ***O(1)***
  - insert( ) is ***O(n)***
- Processing n items requires $O(n^2)$

Two dimensional :

- $\sqrt{N} \times \sqrt{N}$ array with sorted rows
  - insert( ) is $O(\sqrt{N})$
  - delete_max is $O(\sqrt{N})$
  - Processing N items is $O(N\sqrt{N})$

Binary tree

A binary tree is a tree data structure in which each node can contain at most 2 children, which are referred to as the left child and the right child.

Heap

Heap is a binary tree, filled level by level, left to right. There are two types of the heap:

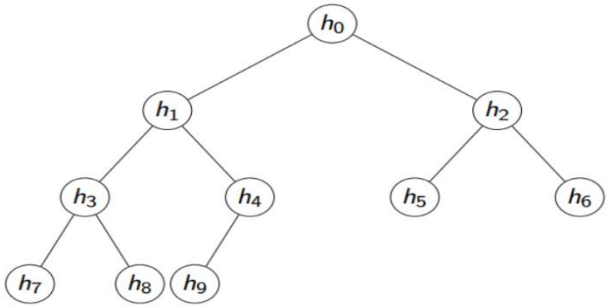
- Max heap - For each node V in heap except for leaf nodes, the value of V should be greater or equal to its child's node value.
- Min heap - For each node V in heap except for leaf nodes, the value of V should be less or equal to its child's node value.
- **We can represent heap using array(list in python)**

`H = [h0, h1, h2, h3, h4, h5, h6, h7, h8, h9]`

left child of `H[i] = H[2 * i + 1]`

Right child of `H[i] = H[2 * i + 2]`

Parent of `H[i] = H[(i-1) // 2], for i > 0`




```python
#Max Heap
class maxheap:
    def __init__(self):
        self.A = []
    def max_heapify(self,k):
        l = 2 * k + 1
        r = 2 * k + 2
        largest = k
        if l < len(self.A) and self.A[l]>self.A[largest]:
            largest = l
        if r < len(self.A) and self.A[r]>self.A[largest]:
            largest = r
        if largest != k:
            self.A[k],self.A[largest]=self.A[largest],self.A[k]
            self.max_heapify(largest)

    def build_max_heap(self,L):
        self.A = []
        for i in L:
            self.A.append(i)
        n = int((len(self.A)//2)-1)
        for k in range(n, -1, -1):
            self.max_heapify(k)

    def delete_max(self):
        item = None
        if self.A != []:
            self.A[0],self.A[-1] = self.A[-1],self.A[0]
            item = self.A.pop()
            self.max_heapify(0)
        return item

    def insert_in_maxheap(self,d):
        self.A.append(d)
        index = len(self.A)-1
        while index > 0:
            parent = (index-1)//2
            if self.A[index] > self.A[parent]:
                self.A[index],self.A[parent]=self.A[parent],
                self.A[index]
                index = parent
            else:
                break
```

```python
#Min Heap
class minheap:
    def __init__(self):
        self.A = []
    def min_heapify(self,k):
        l = 2 * k + 1
        r = 2 * k + 2
        smallest = k
        if l < len(self.A) and self.A[l]<self.A[smallest]:
            smallest = l
        if r < len(self.A) and self.A[r]<self.A[smallest]:
            smallest = r
        if smallest != k:
            self.A[k], self.A[smallest]=self.A[smallest],self.A[k]
            self.min_heapify(smallest)

    def build_min_heap(self,L):
        self.A = []
        for i in L:
            self.A.append(i)
        n = int((len(self.A)//2)-1)
        for k in range(n, -1, -1):
            self.min_heapify(k)

    def delete_min(self):
        item = None
        if self.A != []:
            self.A[0],self.A[-1] = self.A[-1],self.A[0]
            item = self.A.pop()
            self.min_heapify(0)
        return item

    def insert_in_minheap(self,d):
        self.A.append(d)
        index = len(self.A)-1
        while index > 0:
            parent = (index-1)//2
            if self.A[index] < self.A[parent]:
                self.A[index],self.A[parent] = self.A[parent],
                self.A[index]
                index = parent
            else:
                break
```

Complexity

Heaps are a tree implementation of priority queues

- insert( ) is ***O(logN)***
- delete max( ) is ***O(logN)***
- heapify( ) builds a heap in ***O(N)***

Complexity : Heap Sort

- Start with an unordered list
- Build a heap — ***O(n)***
- Call delete max() n times to extract elements in descending order — ***O(nlogn)***
- After each delete max(), heap shrinks by 1
- Store maximum value at the end of current heap
- **In place *O(nlogn)* sort**

Binary Search Tree (BST)

A **binary search tree** is a binary tree that is either empty or satisfies the following conditions:

For each node V in the Tree

- The value of the left child or left subtree is always less than the value of V.
- The value of the right child or right subtree is always greater than the value of V

```
# considering dictionary as a heap for given code
def min_heapify(i,size):
    lchild = 2*i + 1
    rchild = 2*i + 2
    small = i
    if lchild < size-1 and HtoV[lchild][1] < HtoV[small][1]:
        small = lchild
    if rchild < size-1 and HtoV[rchild][1] < HtoV[small][1]:
        small = rchild
    if small != i:
        VtoH[HtoV[small][0]] = i
        VtoH[HtoV[i][0]] = small
        (HtoV[small],HtoV[i]) = (HtoV[i], HtoV[small])
        min_heapify(small,size)

def create_minheap(size):
    for x in range((size//2)-1,-1,-1):
        min_heapify(x,size)

def minheap_update(i,size):
    if i!= 0:
        while i > 0:
            parent = (i-1)//2
            if HtoV[parent][1] >  HtoV[i][1]:
                VtoH[HtoV[parent][0]] = i
                VtoH[HtoV[i][0]] = parent
                (HtoV[parent],HtoV[i]) = (HtoV[i], HtoV[parent])
            else:
                break
            i = parent

def delete_min(hsize):
    VtoH[HtoV[0][0]] = hsize-1
    VtoH[HtoV[hsize-1][0]] = 0
    HtoV[hsize-1],HtoV[0] = HtoV[0],HtoV[hsize-1]
    node,dist = HtoV[hsize-1]
    hsize = hsize - 1
    min_heapify(0,hsize)
    return node,dist,hsize
```

```
#Heap sort Implementation:
def max_heapify(A,size,k):
    l = 2 * k + 1
    r = 2 * k + 2
    largest = k
    if l < size and A[l] > A[largest]:
        largest = l
    if r < size and A[r] > A[largest]:
        largest = r
    if largest != k:
        (A[k], A[largest]) = (A[largest], A[k])
        max_heapify(A,size,largest)

def build_max_heap(A):
    n = (len(A)//2)-1
    for i in range(n, -1, -1):
        max_heapify(A,len(A),i)

def heapsort(A):
    build_max_heap(A)
    n = len(A)
    for i in range(n-1,-1,-1):
        A[0],A[i] = A[i],A[0]
        max_heapify(A,i,0)
```

```
#Updated Implementation for adjacency matrix using min heap:
#global HtoV map heap index to (vertex,distance from source)
#global VtoH map vertex to heap index
HtoV, VtoH = {},{}
def dijkstra(WMat,s):
    (rows,cols,x) = WMat.shape
    infinity = float('inf')
    visited = {}
    heapsize = rows
    for v in range(rows):
        VtoH[v]=v
        HtoV[v]=[v,infinity]
        visited[v] = False
    HtoV[s]= [s,0]
    create_minheap(heapsize)

    for u in range(rows):
        nextd,ds,heapsize = delete_min(heapsize)
        visited[nextd] = True
        for v in range(cols):
            if WMat[nextd,v,0] == 1 and (not visited[v]):
                '''update distance of adjacent of v if it is
                   less than to previous one'''
                HtoV[VtoH[v]][1] = min(HtoV[VtoH[v]][1],ds+WMat[nextd,v,1])
                minheap_update(VtoH[v],heapsize)
```

```
#Updated Implementation for adjacency list using min heap:
HtoV, VtoH = {},{}
#global HtoV map heap index to (vertex,distance from source)
#global VtoH map vertex to heap index
def dijkstralist(WList,s):
    infinity = float('inf')
    visited = {}
    heapsize = len(WList)
    for v in WList.keys():
        VtoH[v]=v
        HtoV[v]=[v,infinity]
        visited[v] = False
    HtoV[s]= [s,0]
    create_minheap(heapsize)

    for u in WList.keys():
        nextd,ds,heapsize = delete_min(heapsize)
        visited[nextd] = True
        for v,d in WList[nextd]:
            if not visited[v]:
                HtoV[VtoH[v]][1] = min(HtoV[VtoH[v]][1],ds+d)
                minheap_update(VtoH[v],heapsize)
```

Complexity : BST

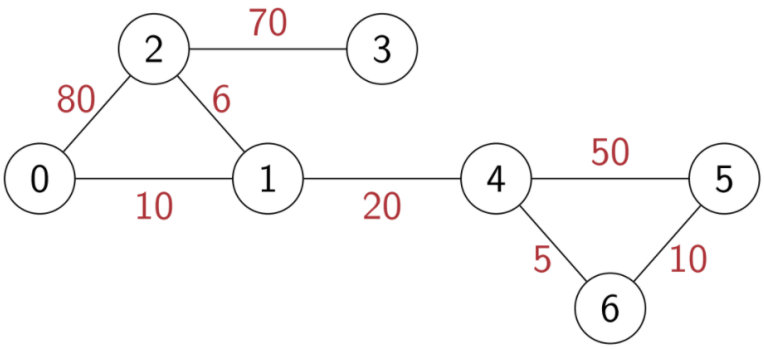
- find( ), insert( ) and delete( ) all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with n nodes may have height ***O(n)***
- Balanced trees have height ***O(logn)***
- Will see how to keep a tree balanced to ensure all operations remain ***O(logn)***

BY – ADITYA DHAR DWIVEDI

Shortest Paths in Weighted Graphs

Weighted Graphs

- Recall that BFS explores a graph level-by-level
- BFS computes the shortest path, in terms on number of edges, to every reachable vertex
- May assign values to edges
  - Cost, time, distance, ...
  - Weighted graph
- G = (V, E), W: E→R, where R represents the set of real numbers



- **Adjacency matrix**
  - Record weights along with edge information -- weight is always 0 if there is no edge

	0	1	2	3	4	5	6
0	(0,0)	(1,10)	(1,80)	(0,0)	(0,0)	(0,0)	(0,0)
1	(1,10)	(0,0)	(1,6)	(0,0)	(1,20)	(0,0)	(0,0)
2	(1,80)	(1,6)	(0,0)	(1,70)	(0,0)	(0,0)	(0,0)
3	(0,0)	(0,0)	(1,70)	(0,0)	(0,0)	(0,0)	(0,0)
4	(0,0)	(1,20)	(0,0)	(0,0)	(0,0)	(1,50)	(1,5)
5	(0,0)	(0,0)	(0,0)	(0,0)	(1,50)	(0,0)	(1,10)
6	(0,0)	(0,0)	(0,0)	(0,0)	(1,5)	(1,10)	(0,0)

- **Adjacency list**
  - Record weights along with edge information

0	[(1,10),(2,80)]
1	[(0,10),(2,6),(4,20)]
2	[(0,80),(1,6),(3,70)]
3	[(2,70)]
4	[(1,20),(5,50),(6,5)]
5	[(4,50),(6,10)]
6	[(4,5),(5,10)]

```python
#Weighted directed graph
#Adjacency matrix representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
size = 7
import numpy as np
W = np.zeros(shape=(size,size,2))
for (i,j,w) in dedges:
    W[i,j,0] = 1
    W[i,j,1] = w
print(W)

#Adjacency list representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
size = 7
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in dedges:
    WL[i].append((j,d))
print(WL)
```

```python
#Weighted undirected graph
#Adjacency matrix representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
edges = dedges + [(j,i,w) for (i,j,w) in dedges]
size = 7
import numpy as np
W = np.zeros(shape=(size,size,2))
for (i,j,w) in edges:
    W[i,j,0] = 1
    W[i,j,1] = w
print(W)

#Adjacency list representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
edges = dedges + [(j,i,w) for (i,j,w) in dedges]
size = 7
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in edges:
    WL[i].append((j,d))
print(WL)
```

Shortest paths in weighted graphs

- BFS computes shortest path, in terms of number of edges, to every reachable vertex
- In a weighted graph, add up the weights along the path
- Weighted shortest path need not have minimum number of edges
    - Shortest path from 0 to 2 is via 1

Single source shortest paths

- Find shortest paths from a fixed vertex to every other vertex
- Transport finished product from factory (single source) to all retail outlets
- Courier company delivers items from distribution centre (single source) to addresses

All pairs shortest paths

- Find shortest paths between every pair of vertices i and j
- Optimal airline, railway, road routes between cities

Negative edge weights

- Can negative edge weights be meaningful?
- Taxi driver trying to head home at the end of the day
    - Roads with few customers, drive empty (positive weight)
    - Roads with many customers, make profit (negative weight)
    - Find a route toward home that minimizes the cost

Negative cycles

- A negative cycle is one whose weight is negative
    - Sum of the weights of edges that make up the cycle

- By repeatedly traversing a negative cycle, total cost keeps on decreasing
- If a graph has a negative cycle, shortest paths are not defined
- Without negative cycles, we can compute shortest paths even if some weights are negative

Summary

- In a weighted graph, each edge has a cost
  - Entries in adjacency matrix capture edge weights
- Length of a path is the sum of the weights
  - Shortest path in a weighted graph need not be the minimum in terms of number of edges
- Different shortest path problems
  - Single source: four one designated vertex to all others
  - All-pairs: between every pair of vertices
- Negative edge weights
  - Should not have negative cycles
  - Without negative cycles, shortest paths still well defined

Dijkstra's Algorithm : Single Source Shortest Path

- Dijkstra’s algorithm works for both directed and undirected graphs.
- Dijkstra’s algorithm doesn’t work for graphs with negative weights or negative weight cycles.
- This algorithm returns the shortest distance from the source to all other nodes, but after some modification like maintaining parent information of each node we can find out the shortest path.

Implementation

- Maintain two dictionaries with vertices as they keys
  - visited, initially False for all v (burnt vertices)
  - distance, initially infinity for all v (expected burn time)
- Set distance[s] to 0
- Repeat, until all reachable vertices are visited
  - Find unvisited vertex nextv with minimum distance
  - Set visited[nextv] to True
  - Recompute distance[v] for every neighbour v of nextv

Summary

- Dijkstra's algorithm computes single source shortest paths
- Use a greedy strategy to identify vertices to visit
  - Next vertex to visit is based on shortest distance computed so far
  - Need to prove that such a strategy is correct
  - Correctness requires edge weights to be non-negative

Complexity is $O(n^2)$

- Even with adjacency lists
- Bottleneck is identifying unvisited vertex with minimum distance
- Need a better data structure to identify and remove minimum (or max) from a collection.

```python
# Adjacency matrix implementation
def dijkstra(WMat, s):
  (rows, cols, x) = WMat.shape
  infinity = np.max(WMat) * rows + 1
  (visited, distance) = ({}, {})

  for v in range(rows):
    (visited[v], distance[v]) = (False, infinity)

  distance[s] = 0

  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                     if not visited[v]])
    nextvlist = [v for v in range(rows)
                   if (not visited[v]) and
                       distance[v] == nextd]

    if nextvlist == []:
      break

    nextv = min(nextvlist)
    visited[nextv] = True

    for v in range(cols):
      if WMat[nextv, v, 0] == 1 and (not visited[v]):
        distance[v] = min(distance[v], distance[nextv]
                          + WMat[nextv, v, 1])

  return distance
```

```python
# Adjacency list implementation
def dijkstra(WList, s):
  infinity = 1 + len(WList.keys()) * max([d for u in WList.keys()
                                          for (v, d) in WList[u]])
  (visited, distance) = ({}, {})

  for v in WList.keys():
    (visited[v], distance[v]) = (False, infinity)

  distance[s] = 0

  for u in WList.keys():
    nextd = min([distance[v] for v in WList.keys()
                     if not visited[v]])
    nextvlist = [v for v in WList.keys()
                   if (not visited[v]) and
                       distance[v] == nextd]
    if nextvlist == []:
      break

    nextv = min(nextvlist)
    visited[nextv] = True

    for (v, d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v], distance[nextv] + d)

  return distance
```

Bellman Ford algorithm : Single Source Shortest with Negative Weights

- Bellman-Ford works for both directed and undirected graphs with non-negative edges weights.
- Bellman-Ford does not work with an undirected graph with negative edges weight, as it will be declared as a negative weight cycle.
- Bellman-Ford works for a directed graph with negative edge weight, but not with negative weight cycle.

Complexity

- **O(n^3)** for adjacency matrix.
- **O(mn)** for adjacency list : where m is number of edges and n is number of vertices.

Summary

- Dijkstra's algorithm assumes non-negative edge weights
  - Final distance is frozen each time a vertex "burns"
  - Should not encounter a shorter route discovered later
- Without negative cycles, every shortest route is a path
- Every prefix of a shortest path is also a shortest path
- Iteratively find the shortest paths of length 1,2,...,n−1
- Update distance of each vertex with every iteration -- **Bellman-Ford algorithm**
- $$ time with adjacency matrix, **O(mn)** time with adjacency list
- if Bellman-Ford algorithm does not converge after n−1 iterations, there is a negative cycle

```
def bellman_ford(WMat, s):
  (rows, cols, x) = WMat.shape
  infinity = np.max(Wmat) * rows + 1
  distance = {}

  for v in range(rows):
    distance[v] = infinity

  for i in range(rows):
    for u in range(rows):
      for v in range(cols):
        if WMat[u, v, 0] == 1:
          distance[v] = min(distance[v],
                distance[u] + WMat[u, v, 1])

  return distance
```

```
def bellman_ford_list(WList, s):
  infinity = 1 + len(WList.keys()) * max
  ([d for u in WList.keys()for (v, d) in WList[u]])
  distance = {}

  for v in WList.keys():
    distance[v] = infinity

  distance[s] = 0

  for i in WList.keys():
    for u in WList.keys():
      for (v, d) in WList[u]:
        distance[v] = min(distance[v],
                          distance[u] + d)

  return distance
```

All pair of shortest path

- Find the shortest paths between every pair of vertices i and j.
- It is equivalent to if run Dijkstra or Bellman-Ford from each vertex.

Floyd-Warshall algorithm

- Floyd-Warshall's works for both directed and undirected graphs with non-negative edges weights.
- Floyd-Warshall's does not work with an undirected graph with negative edges weight, as it will be declared as a negative weight cycle.
- Floyd-Warshall's algorithm is an alternative way to compute transitive closure **B k [i, j] = 1** if we can reach j from i using vertices in {0, 1, . . . , k−1}
- Floyd-Warshall works for a directed graph with negative edge weight, but not with a negative weight cycle.
- Formula for Floyd-Warshall algorithm is given below:-
- $$SP^k[i,j] = min[SP^{k-1}[i,j],\ SP^{k-1}[i,k] + SP^{k-1}[k,j]]$$

Summary

- Warshall's algorithm is an alternative way to compute transitive closure
  - **B^k[i,j]=1** if we can reach j from i using the vertices in {0,1,...,k−1}

- Adapt Warshall's algorithm to compute all pairs of shortest paths
  - **SP^k[i,j]** if the length of the shortest path from i to j using vertices in {0,1,...,k−1}
  - **SP^n[i,j]** is the length of the overall shortest path

- **Works with negative edge weights assuming no negative cycles**
- **Simple nested loop implementation, time $O(n^3)$**
- Space can be limited to **$O(n^2)$** by re-using 2 "slices" SP and SP'

```python
#For adjacency matrix
def floyd_warshall(WMat):
  (rows, cols, x) = WMat.shape
  infinity = np.max(WMat) * rows * rows + 1
  Sp = np.zeros(shape=(rows, cols, cols + 1))

  for i in range(rows):
    for j in range(cols):
      SP[i, j, 0] = infinity

  for i in range(rows):
    for j in range(cols):
      if WMat[i, j, 0] == 1:
        SP[i, j, 0] = WMat[i, j, 1]

  for k in range(1, cols + 1):
    for i in range(rows):
      for j in range(cols):
        SP[i, j, k] = min(SP[i, j, k - 1]),
        SP[i, k - 1, k - 1] + SP[k - 1, j, k - 1])

  return SP[:, :, cols]
```

Minimum Cost Spanning Tree(MCST)

Spanning Tree(ST)

- Retain a minimal set of edges so that graph remains connected
- Recall that a minimally connected graph is a tree
- Adding an edge to a tree creates a loop
- Removing an edge disconnects the graph
- Want a tree that connects all the vertices — **spanning tree**
- More than one spanning tree, in general

Spanning trees with cost

- Restoring a road or laying a fiber optic cable has a cost
- Minimum cost spanning tree
  - Add the cost of all the edges in the tree
  - Among the different spanning trees, choose one with the minimum cost
- Some facts about trees
  - **A tree on n vertices has exactly n − 1 edges**
  - **Adding an edge to a tree must create a cycle.**
  - **In a tree, every pair of vertices is connected by a unique path**

Building minimum cost spanning trees

- We will use these facts about trees to build minimum cost spanning trees
- Two natural strategies
- Start with the smallest edge and "grow" a tree
  - **Prim's Algorithm**
- Scan the edges in ascending order of weight to connect components without forming cycles
  - **Kruskal's Algorithm**

Summary

- **Prim's algorithm grows an MCST starting with any vertex**
- Implementation similar to Dijkstra's algorithms : Update rule for distance is different

- At each step, connect one more vertex to the tree using minimum cost edge from inside the tree to outside the tree

- Complexity is **O(n^2)**
  - Even with adjacency lists
  - Bottleneck is identifying unvisited vertex with minimum distance
  - Need a better data structure to identify and remove minimum (or max) from a collection

- **Kruskal's algorithm builds an MCST bottom up**
  - Start with n components, each an isolated vertex
  - Scan the edges in ascending order of cost
  - Whenever an edge merges disjoint components, add it to the MCST
- Correctness follows from Minimum Separator Lemma
- Complexity is **O(n^2)** due to naive handling of components
  - We will see how to improve to **O(m.logn)**
- If edge weights repeat, MCST is not unique
- "Choose minimum cost edge" will allow choices
  - Consinder a triangle on 3 vertices with all edges equal
- Different choices lead to different spanning trees
- In general, there may be a very large number of minimum cost spanning trees

```
#Prim's Algorithm using List
def prim_list2(WList):
  infinity = 1 + max([d for u in WList.keys()
                        for (v, d) in WList[u]])
  (visited, distance, nbr) = ({}, {}, {})

  for v in WList.keys():
    (visited[v], distance[v], nbr[v]) =
                    (False, infinity, -1)

  visited[0] = True

  for (v, d) in WList[0]:
    (distance[v], nbr[v]) = (d, 0)

  for i in range(1, len(WList.keys())):
    nextd = min([distance[v] for v in WList.keys()
                    if not visited[v]])
    nextvlist = [v for v in WList.keys()
                    if (not visited[v]) and
                          distance[v] == nextd]

    if nextvlist == []:
      break

    nextv = min(nextvlist)
    visited[nextv] = True

    for (v, d) in WList[nextv]:
      if not visited[v]:
        (distance[v], nbr[v]) = (min(distance[v], d),
                                 nextv)
  return nbr
```

```
#Kruskal's Algorithm using List
def kruskal(WList):
  (edges, components, TE) = ([], {}, [])

  for u in WList.keys():
    edges.extend([(d,u,v) for (v,d) in WList[u]])
    component[u] = u
  edges.sort()

  for (d, u, v) in edges:
    if component[u] != component[v]:
      TE.append((u, v))
      c = component[u]

      for w in WList.keys():
        if component[w] == c:
          component[w] = component[v]

  return TE
```