Algorithms & Data Structures for Big Data

Date 07/08/2023

Algorithm :- Set of clear, unambiguous instructions to solve a problem in finite amount of time.

Conditions :- a) input b) output c) finiteness
d) definiteness e) effectiveness

Data structures :- A systematic scheme of organizing the related data.

Program :- Converting an algorithm to an executable language.

09/08/2023

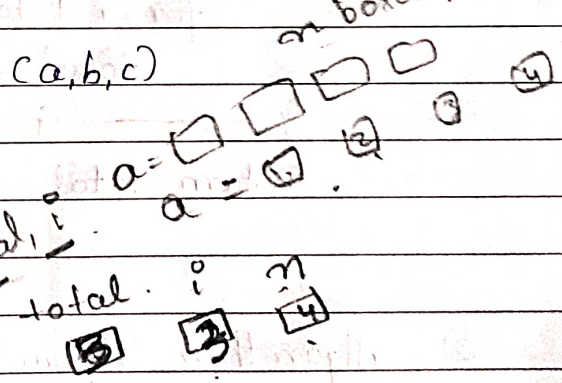
Performance of algorithm :-

Space - Memory required for the computation.
Time - CPU time considered, not the compile time.

Space — Variable → instance size, reference variables, recursion function
Fixed → code, fixed variables, constants.

Total space = Fixed space + variable space.

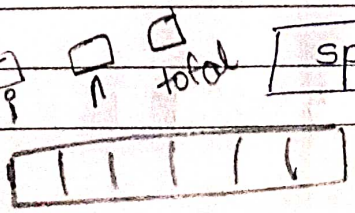
Ex:- 1) Algorithm addElement (a,b,c)
begin
return (a+b+c)
end
→ Only fixed space



2) Algorithm sum (a, n) {
total = 0 — 1
for i = 1 to n { n = 1 i = 1
total = total + a[i] : n
}
return total
}

ans:-
n = 1
total = 1
i = 1
a = variable = n

space ≥ (n+3)

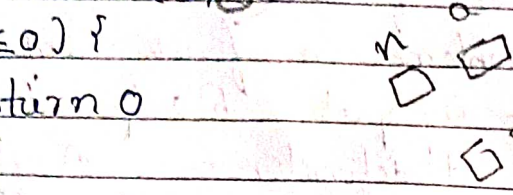


3 (n+1)

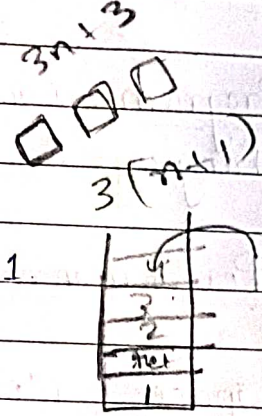
3) Algorithm Rsum (a, n) {
if (n≤0) {
return 0
} else {
return (Rsum (a, n-1) + a[n])
}}

1 return address
3 spaces

Space :-
a = 1 unit (base address)
n = 1
return address = 1.
Total no. of function calls = n+1
Total space ≥ 3n+3



1, 2, 3, 4



11/08/2023

Time complexity :-
Only execute and time is considered. Compile time is excluded.

Ex: 1) Algorithm sum (a, n) { — 0
total = 0. — 1.
for (i = 1 to n) { — n
total = total + a[i] — n
} — 1
return total — 1

Ans
c = 0
c = 1
c = 2
c = 3
c++ / i > n
c++
2n+3

2) Algorithm Rsum (a, n) { a = {1, 2, 3, 4, 5} n = 5
if (n≤0) { — 1
return 0 — 1
} else { — 2
return (Rsum (a, n-1) + a[n])
}}
a = {1, 2, 3, 4} n = 4
Rsum (1, 2, 3, 4) + 4)
Rs

Rsum (1,2,3,4) = Rsum (1,2,3) + 5
Rsum (1,2,3) = Rsum (1,2) + 4
Rsum (1,2) = Rsum (1) + 3
Rsum (1) = Rsum (0) + 2
Rsum (0) = 1

Date 11/8/2023

$T(n) = 2$ when $n \le 0$.

$T(n) = 2 + T(n-1)$ when $n > 0$

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

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

$T(k) = 2*k + T(n-k)$ → Substitution method.

$= 2n + T(n-n)$ — $T(n) = 2$

$= 2n + T(0)$ — $T(0) = 2$.

$T(n) = 2n + 2$ — $n - k = 0$

$n = k$

=

3) Algorithm addMatrix (a, b, m, n, c)

for (i = 1 to m) { — c++ — m

for (j = 1 to n) { — c++ — (n+1)m

c[i,j] = a[i,j] + b[i,j] — c++ — m*n

} }

return c. — c++ — 1.

$mn + m + n + 2$

$2 \quad 2$

a b

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \begin{bmatrix} 2 & 3 \\ 1 & 4 \end{bmatrix}$

3) Algorithm addMatrix (a, b, m, n, c) {

for (i = 1 to m) { → c++ = m

for (j = 1 to n) { → c++ = mn

c[i,j] = a[i,j] + b[i,j] → c++ = m*n

} c++ / when j > n = m

} c++ / when i > m = 1

return c. → c++ = 1

}

$2mn + 2m + 2$

if (m == n)

$\Rightarrow 2n^2 + 2n + 2$

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$

$2 + 2 + 2$.

$3 * 2$

$m * n$

classmate
Date 11/8/2023
Page

Asymptotic notation :-

Big oh (O) :- Upper bound time complexity : Worst case value

Omega (Ω) :- Best case value. Time complexity in lower bound value.

Theta (Θ) : Average time complexity.

14/08/2023

General solution for large classes of ~~occur~~ recurrence problems :-

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

n = no. of elements
a = subproblems
n/b = elements in subproblem
$d(n)$ = Time to create subproblem + merge the solution

Homogenous Solution :-

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

Particular solution :-

case 1 : when $a > d(b)$

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

case 2 : when $a < d(b)$

$T(n) = O(n^{\log_b d(b)})$

case 3 : when $a = d(b)$

$T(n) = O(\log_b n * n^{\log_b d(b)})$

Ex :- 1) $T(n) = 4T(n/2) + n$.

$a = 4 \quad b = 2. \quad d(n) = n \Rightarrow d(b) = b = 2.$

$\therefore a > d(b)$

$O(n^{\log_2 4})$

$T(n) \Rightarrow O(n^2)$ = both homogenous & particular solution

classmate
Date 14/8/2023
Page

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

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

$\Rightarrow d(b) = b^2 = 4$

$a = b$

$T(n) = O(\text{Log } n * n^{\log_b d(b)})$

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

$\left(n^{\log_2 4} \quad n^2\right)$

3. ~~$T(n) = 4T(n/2) + 2$~~

$a = 4 \quad b = 2 \quad d(n) = 2 \Rightarrow d(b) = 2$

$a > d(b)$

$T(n) = O(n^{\log_2 4})$

$= O(n^2)$

$\log_4 2$

$2 = 4^{1/2}$

$2 = 4^{(k)}$

$\log_2 8$

$8 = 2^{(k)} = 3$

$\log_4 2$

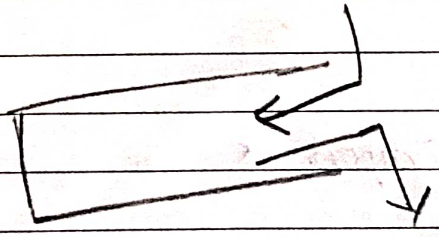
$2 = 4^k \Rightarrow k = 1/2$

Stacks :-

→ LIFO data structure
→ Last in First out
→ Function calls
→ Evaluation expression
→ Conversion of postfix/prefix expression
→ Balancing of symbols
→ Webpages, editors

$2+3*4$

16/8/2023 "self" tells the __init__ is an instance method.
"this" represents the object.



1	2	3	4

3!

1
2
3

$2*3 = 6$

```
class simpleStack :
    def __init__(self):
        self.data = []
        self.count = 0
    def getElementCount(self):
        return self.count
    def isStackEmpty(self):
        return self.count == 0
    def stackPush(self, element):
        self.data.append(ele)
        self.count += 1.
    def stackPop(self):
        if not self.isempty():
            self.count -= 1
            return self.data.pop()
        else:
            return None
    def stackPeek(self):
        if not self.isEmpty():
            return self.data[-1]
        else:
            return None
    def printElements(self):
```

testSimpleStack.py :-

```
from simplestack import *

def testEmptyStack():
    S1 = simpleStack()
    assert(S1.getElementCount() == 0)
    assert(S1.isStackempty() == True)
```

Date 16/8/2023

```
def testPush():
    s2 = simplestack()
    s2.stackPush(10)
    assert(s2.getElementCount()==1)

def testPop():
    s3 = simplestack()
    s3.stackPush(20)
    s3.stackPush(30)
    assert(s3.stackPop()==30)
```

16/8/2023

```
from simplestack import *

def testSymbols(data, stacks):
    lefty = '{[('
    righty = '}])'
    for ch in data:
        if ch in lefty
            stack.stackPush(ch)
        elif ch in righty:
            if stacks.isEmpty():
                return False
            elif righty.index(ch) != lefty.index(stack.stack
                                                 Pop())
                return False
    return (stack.isEmpty())
```

test

```
def testApp():
    stacks = simpleStack()
    f_read = open('','r'). open("","r")
    data = f_read.read()
    f_read.close()
    assert (testSymbols(data, stacks)==True)

testApp()
```

classmate
Date 18/08/2023
Page

Queue:

→ First in First out principle (FIFO)

```
Class Queue:
    def __init__(self):
        self.data = []
        self.count = 0          self.front = 0.
    def getElementCount(self):
        return self.count
    def isQueueEmpty(self):
        return self.count == 0
    def enqueue(self, element):
        self.data.append(element)
        self.count += 1
    def deque(self):
        if not self.isEmpty():
            count -= 1          self.count -= 1
            return self.data.remove(front)
            front += 1
            return self.data.pop(0)
        else:
            return None.
```

23/08/23 Time complexity → enqueue - O(1)
dequeue - O(n)

classmate
Date 23/08/2023
Page

Dynamic Queue:-

```
class flexiQueue:
    defaultSize = 2
    def __init__(self):
        self.data = [None] * flexiQueue.defaultSize
        self.front = 0
        self.count = 0

    def length(self):
        return self.count
    def isEmpty(self):
        return self.count == 0
    def getFirst(self):
        if not self.isEmpty():
            return self.data[self.front]
        else
            return None
    def enqueue(self, ele):
        if self.count == len(self.data):
            self.reSize(2 * len(self.data))
        idx = (self.front + self.count) % len(self.data)
        self.data[idx] = ele
        self.count += 1
    def dequeue(self):
        if not self.isEmpty():
            elem = self.data[self.front]
            self.count -= 1
            self.data[front] = None
            self.front = (self.front++) % len(self.data)
            if 0 < self.size < len(self.data)//4:
                self.resize(len(self.data)//3
            return size
        else:
            return None
```

```
def resize(self, cap):
    oldData = self.data
    walk = self.front
    self.data = [None] * cap
    for k in range(len(oldData)):
        self.data[k] = oldData[walk]
        walk = (walk+1) % len(oldData)
    self.front = 0
```

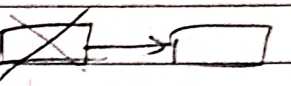
25/08/2023

Linked List :-

```
class slist:
    class _Node:
        def __init__(self, ele):
            self.data = elem
            self.next = None
    def __init__(self):
        self.head = None
        self.tail = None
        self.count = 0
    def isempty(self):
        return self.count == 0
    def listCount(self):
        return self.count
    def addAtHead(self, ele):
        # if not self.isempty():
        new_node = self._Node(ele)
        if not self.isEmpty():
            new_node.next = self.head
            self.head = new_node
        else:
            self.head = self.tail = new_node
        self.count += 1
```

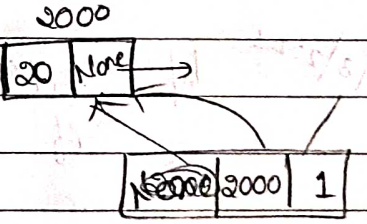
classmate
Date 28/08/2023
Page

```
def addAtTail(self,ele):
    new_node = self.Node(ele)
    if not self.isEmpty():
        self.tail.next = new_node
        self.tail = new_node
    else:
        self.head = self.tail = new_node
    self.count += 1
```

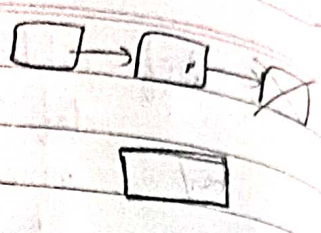


```
def deleteAtHead(self):
    if not self.isEmpty():
        temp = self.head.next
        self.head = temp
    if self.count > 1:
        temp = self.head.next
        del(self.head)
        self.head = temp
        self.count -= 1
    elif self.count == 1:

        self.head = self.tail = None
        self.count = 0
    else:
        return None
```

(partial code)




```
def deleteAtHead(self):
    if not self.isEmpty():
        data = self.head.data
        self.head = self.head.next
        if self.head == None:
            self.tail = None

        self.count -= 1
        return data
    else: return None
```



```
def deleteAtTail(self):
    if not self.isEmpty():
        data = self.tail.data
        self.tail = self.head
        temp = self.head
        while (temp.next != None):
            temp = temp.next
        self.tail = temp
        temp.next = None
```

(partial code)

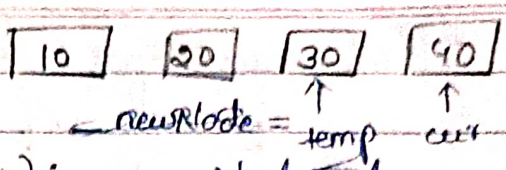
```
def deleteAtTail(self):
    if not self.isEmpty():
        if self.count != 1:
            last = self.tail
            cur = self.head
            while cur.next != last:
                cur = cur.next
            self.tail = cur
            cur.next = None
        else:
            self.head = self.tail = None
        self.count -= 1
    else:
        return None
```

30/8/2023

```
def isMember(self, key):
    if not self.isEmpty():
        cur = self.head
        while cur != None:
            if cur.data == key:
                break
            else
                cur = cur.next
        return cur != None
    else
        return False
```

Classmate
Date 30/08/2023
Page

def addGiven




```
def addAfterNode(self, ele, pos):
    if not self.isEmpty():
        temp = self.head
        while(temp.data != pos):
            temp = temp.next
        temp = temp.next
        cur = temp.next
        temp.next = newNode
        newNode.next = cur
        self.count += 1
```

```
def addAfterNode(self, ele, value):
    newNode = self._Node(ele)
    if not self.isEmpty():
        temp = self.head
        while(temp.data != value):
            temp = temp.next
        cur = temp.next
        temp.nex
        if not temp.next == None:
            cur = temp.next
            temp.next = newNode
            newNode.next = cur
        else:
            temp.next = newNode
            self.tail = newNode
        self.count += 1
    else:
        return None
```

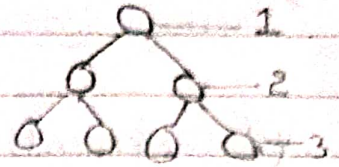
```
def addNtpos (self, ele, pos):
    if not self.isEmpty():
        temp = self.head
        while self.iteration = 0
        while (self.iteration < pos):
            temp = temp.next
            self.iteration += 1
        if temp != None:
            cur = temp.next
            newNode = self.__newNode(ele)
            if cur != None:
                temp.next = newNode
                newNode.next = cur
            else:
                temp.next = newNode
                self.tail = newNode
            self.count += 1
        else:
            return None
    else:
        return None
```

01/09/2023

```
def StackUsingQueues(self):
    q1 = Queue()
    q2 = Queue()
    def push(self, ele):
        q1.enqueue(ele)
    def pop():
        while (q1.count > 1):
            q2.enqueue(q1.dequeue), ele = q1.dequeue
        while (q2.count > 0):
            q1.enqueue(q2.dequeue())
        return ele
```

discrete
Date 4/09/2023
Page

Tree data structure

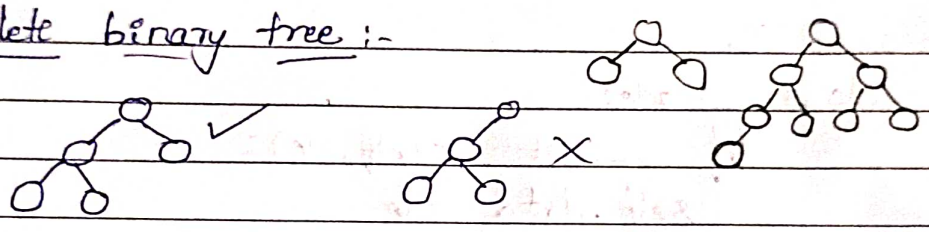


→ Non-linear data structure
→ For file system
→ Root node — Starting node
→ Degree — No. of children nodes for a given node

Binary tree

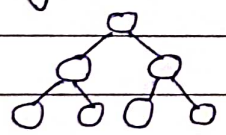
i) No. of nodes at any level $i = 2^{i-1} = 2^{1-1} = 2^0 = 1$
ii) Total No. of nodes at any given depth $k = 2^k - 1$ $2^3 - 1$
7

Complete binary tree :-



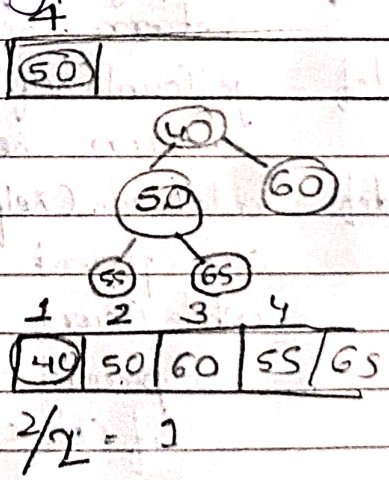
Nodes are filled from left to right then top to bottom

Full binary tree :-



All leaf nodes are filled. There should not be any unfilled slots at any level.

*. node at i
*. parent node at i/2
*. left child at 2i
*. right child at 2i+1

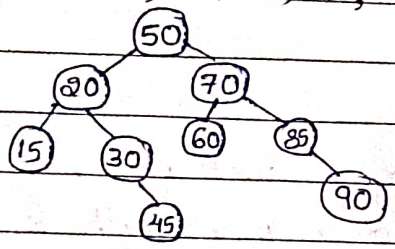


Classmate
Date 06/09/2021
Page

Binary Search Tree :-

1) All elements should be unique
2) Left child should be less than parent
3) Right child should be greater than parent.

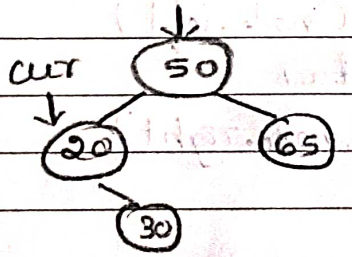
Ex:- 50, 20, 70, 30, 60, 85, 45, 15, 90




```
class BST:
    class _Node:
        def __init__(self,ele):
            self.data =ele
            self.left = None
            self.right = None
    def __init__(self):
        self.root= None
        self.count=0
    def isEmpty(self):
        return self.root ==None
    def getCount(self):
        return self.count
    def addNode(self,ele):
        cur = parent = self.root
        while (cur != None and cur.data != ele):
            parent = cur
            if (ele< cur.data):
                cur = cur.left
            else.
                cur= cur.right
        if (cur==None):
            new_node = self._Node(ele)
```

Jassmate
Date 06/09/2023
Page

```
if parent = None:
        self.root = new_node
elif    ele < parent.data:
        parent.left = new_node
elif    ele > parent.data :
        parent.right = new_node
self.count += 1
def searchNode (self, key):
    if not self.isEmpty():
        cur = self.root
        while (cur != None):
            if cur.data == key:
                break
            elif:
            else:
                cur = cur.n
            elif (key < cur.data):
                cur = cur.left
            else:
                cur = cur.right
        return cur != None      False      cur == None
    else:
        return None  False
```



key = 20

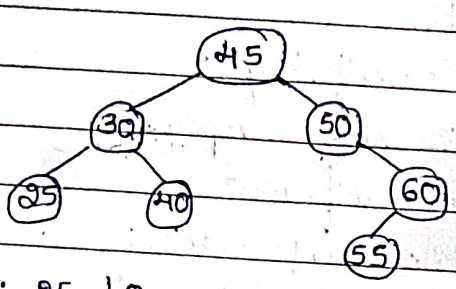
cur != None → True

cur == 20

Date 11/09/2023
Page

BST Traversals :-

1) Inorder – Left → Parent → Right
2) Preorder – Parent → Left → Right
3) Post order – Left → Right → Parent
4) Level order



Post order : 25, 40, 30, 55, 60, 50, 45
Pre order : 45, 30, 25, 40, 50, 60, 55
Inorder : 25, 30, 40, 45, 50, 55, 60
Levelorder : 45, 30, 50, 25, 40, 60, 55

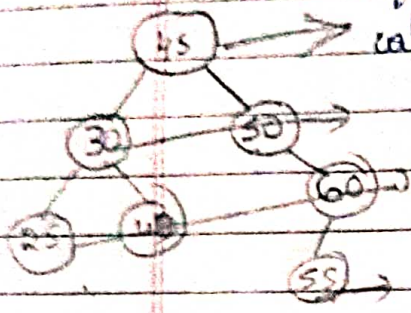
```
def Inorder(self):
    if not self.isEmpty():
        self._inorder(self.root)

def _inorder(self, node):
    if node != None:
        self._inorder(node.left)
        print(node.data)
        self._inorder(node.right)


def PreOrder(self):
    if not self.isEmpty():
        self._preorder(self.root)
def _preOrder(self, node):
    if node != None:
        print(node.data)
        self._preOrder(node.left)
        self._preorder(node.right)
```

Date 11/09/2023
Page

```
def PostOrder(self):
    if not self.isEmpty():
        self.postOrder(self.root)
def _postOrder(self, node):
    if node != None:
        self._postOrder(node.left)
        self._postOrder(node.right)
        print(node.data)


def LevelOrder(self):
    if not self.isEmpty():
        self.levelOrder(self.root)
def _levelOrder(self, node):
    while(node != None):
        if (node.left != None):
            queue.enqueue(node.left)
        if (node.right != None):
            queue.enqueue(node.right)

        queue.enqueue(node)
        print
        ~~enqueue~~(queue.dequeue())

def _leveOrder(self, node):
    queue.enqueue(node)
    while(node != None):
        ele = queue.dequeue()
        print(ele)
        if(node.left != None):
            queue.enqueue(node.left)
        if(node.right != None):
            queue.enqueue(node.right)
```

13/09/2023

```
def level_order(self):
    if not self.isEmpty():
        q = FlexQueue()
        q.enqueue(self.root)
        while not q.isEmpty():
            node = q.dequeue()
            print(node.data)
            if node.left:
                q.enqueue(node.left)
            if node.right:
                q.enqueue(node.right)
```



45	30	50	25	40	60

45, 30, 50

Q: ~~45~~ ~~30~~ ~~50~~ 25 40 60

node = 45

node = 30

45, 30, 50, 25, 40, 60

→ Implementation of tree traversal using an iterative approach

Delete Node :-

```
def deleteNode(self, key):
    if not self.isEmpty():
        self.root = self._nodeDelete_(self.root, key)
def _nodeDelete_(self, node, key):
    if node == None:
        return None
    elif key < node.data:
        node.left = self._nodeDelete_(node.left, key)
    elif key > node.data:
        node.right = self._nodeDelete_(node.right, key)
    elif node.left and node.right:
        temp = self._findMin_(node.right)
        node.data = temp.data
        node.right = self._nodeDelete_(node.right, temp.data)
    else:
        if node.right == None:
            node = node.left
```

```
        else:
            if node.left == None:
                node = node.right
            self.count -1 = 1
            return node
    def _findMin_(self, node):
        if node.left == None:
            return Node.
        else:
            return (self.findMin_(node.left))
```

18/09/2023

```
    def getLeafCount(self):
        count = 0
        if not self.isEmpty():
            q1 = LinkedQueue()
            q1.enque(self.root)
            while not q1.isEmpty():
                node = q1.dequeue()
                if (node.right = node.left == None):
                    count + = 1
                if node.left:
                    q1.enqueue(node.left)
                if node.right:
                    q1.enqueue(node.right)
            return count.

    def getLeafCount(self):
        if not self.isEmpty():
            return (self._leafCount_(self.root))
        else
            return 0

def _leaf_count(self, node):
    if node:
        if self._isLeafNode(node):
            return 1.
        else:
            return (self._leafCount_(self.left) + self._leafCount(node.right)
```

```
    return 0

def is_leafNode_(node):
    if node.left == None and node.right == None
        return True
    else:
        return False
```

25/09/2023

Bubble sort :-

```
def bubbleSort(input):
    for i in range(len(input)-2,0,-1):
        for j in range(len(input)-1, i, -1):
            if input[j] < input[j-1]:
                input[j], input[j-1] = input[j-1], input[j]
```

Time complexity = $O(n^2)$

Improve bubble sort to reduce the number of sorts.

```
def testSorting(input)
    for idx in range(0, len(input)-1):
        assert(input[idx] <= input[idx+1])

test son testSorting(input)
```

Selection sort:-

Less no. of swapping

```
def selectionSort(input):
    for itr in range(0, len(input)-2):
        pos = itr
```

classmate
Date 25/09/2023
Page

```
for idx in range(itr+1, len(input)-1):
    if input[idx] < input[pos]:
        pos = idx
if input[itr], input[pos] = input[pos], input[itr]
```

27/09/2023

Insertion Sort:-

```
def insertionSort(input):
    for itr in range(1, len(input)):
        idx = itr - 1
        key = input[itr]
        while idx >= 0 and key < input[idx]:
            input[idx+1] = input[idx]
            idx -= 1
        input[idx+1] = key
```

Time complexity = $O(n^2)$

no. of comparison is very less.

insertion sort is better than selection & bubble sort

29/09/2023

Quick sort :-

→ Divide & conquer principle

0	1	2	3	4	5	6	7	8	9	10
35	20	45	15	98	9	69	95	16	30	60

pivot = 35

up up up down down

input[up] ≤ pivot

up++;

input[down] > pivot

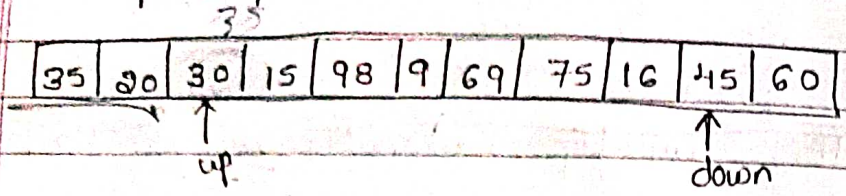
down--;

input[up] = 45 ≰ 35

input[down] ≯ 35

If all elements are not compared & can't increment decrement up & down

Swap up & down



pivot = 35
up = 30
down = 45

Swap pivot with down if all elements are traversed

Depends on the partition, the time complexity will be
→ either $O(n^2)$ or $O(n \log n)$

```
def quickSort(input):
    start = 0
    end = len(input) - 1
    _quicksort_(input, start, end)
    return input

def _quicksort_(input, start, end):
    if start < end: # if start == end, only one element
        mid = _partition_(input, start, end)
        _quicksort_(input, start, mid-1)
        _quicksort_(input, mid+1, end)

def _partition_(input, start, end):
    up = start
    down = end
    pivot = input[start]
    while up <= down:
        while up <= down and input[up] <= pivot:
            up += 1
        while up <= down and input[down] > pivot:
            down -= 1
        input[up], input[down] = input[down], input[up]
    input[start], input[down] = input[down], input[start]
    return down
```

Date 29/09/2023
Page

Balanced partition :- (or best case) :-

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

partition takes $O(n)$ time, maximum time for number of comparison needed is $n+1$.

$a = 2$ $b = 2$ $d(n) = n$

$d(b) = b = 2$

$a = d(b)$.

Homogenous solution $= O(n^{\log_b a})$

$= O(n)$

Particular solution $= O(n^{\log_b d(b)} . \log n)$

$= O(\log n * n)$

$= O(n \log_2 n)$

Unbalanced partition :- (Worst case)

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

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

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

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

$=$ ⋮

$= T(n-n) + (n-n) + \cdots + 1 + 0$

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

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

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

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

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

$=$ ⋮

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

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

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

$= n + \frac{n^2 - n}{2}$

$= O(n^2)$

Merge sort :-

→ Divide & conquer method
→ $T(n) = O(n \log n)$
→ This is not an in-place algorithm. That means we will be using an additional temporary array.

```
def mergeSort(input):
    if len(input) == 0 or len(input) == 1:
        return input
    else:
        mid = len(input)//2
        a = mergeSort(input[:mid])
        b = mergeSort(input[mid:])
        return merge(a,b)

def merge(a,b):
    templist = []
    while len(a) != 0 and len(b) != 0:
        if a[0] < b[0]:
            templist.append(a[0])
            a.pop(0)
        else:
            templist.append(b[0])
            b.pop(0)
    if len(a) == 0:
        templist = templist + b
    else:
        templist = templist + a
    return templist
```

This merge function leads to $O(n^2)$ time complexity

classmate
Date 04/10/2023
Page

→ Implement merge function with time complexity O(n log n).

Time complexity.

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

$a = 2 \quad b = 2 \quad d(b) = 1$

$a = b$

$T(n) = O(n^{\log_2 2} * \log_2 n)$

$T(n) = O(n \log n)$ → Particular solution

Homogenous solution = $O(n)$

Heap:-

→ Complete binary tree

→ Maximum heap: The parent value should always greater or equal to its children.

→ Minimum heap: The parent value should always smaller or equal to its children.

→ Heap sort :- Efficient way of in-place sorting algorithm with time complexity $O(n \log n)$.

→ Works very well even if the dataset is very huge.

06/10/2023

```
class maxHeap:
    def __init__(self, lst=[]):
        self.data = lst
        self._buildHeap_()
    def isEmpty(self):
        return len(self.data) == 0
    def _parent_(self, idx):
        return (idx-1)//2
    def _lchild(self, idx):
        return (2*idx+1)
    def _rchild(self, idx):
        return (2*idx+2)
```

```
def _swap_(self, i, j):
    self.data[i], self.data[j] = self.data[j], self.data[i]

def _buildHeap(self):
    length = len(self.data)
    start = (length-2)//2          # from
    for idx in range(start, -1, -1):   # to
        self._downHeap_(idx, length)
def _downHeap_(self, idx, length):
    if self._lchild(idx) < length:
        left = self._lchild_(idx)
        bigChild = left
        if self._rchild_(idx) < length:
            right = self._rchild_(idx)
            if self.data[right] > self.data[left]:
                bigChild = right
        if self.data[bigChild] >= self.data[idx]:
            self._swap_(bigChild, idx)
            self._downHeap_(bigChild, length)
```

09/10/2023

```
def addElement(self, key):
    self.data.append(key)
    self._upHeap(len(self.data)-1)

def _upHeap(self, j):
    parent = self._parent_(j)
    if j > 0 and self.data[j] > self.data[parent]:
        self._swap_(j, parent)
        self._upHeap_(parent)
```

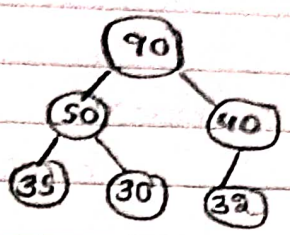
Time required to move biggest item from lowest level to the highest level (up-heap) = O(log n)

Time required to move smallest item from highest level to the lowest level (down-heap) = O(log n)

classmate
Date 09/10/2023
Page

1) Priority queue implementation using normal queue :-
Adding n elements = O(n)
Deleting n elements = $O(n^2)$

2) Sorted queue
$O(n^2)$ — Addition
O(n) — Deletion



90	50	40	35	30	32

3) Heap datastructure
O(n log n) — Addition
O(n log n) — Deletion

32	50	40	35	30	90

50	32	40	35	30	90

Building a heap → O(n log n)

30	32	40	35	50	90

40	32	30	35	50	90

35	32	30	40	50	90

```
def testMaxHeap(self):
    if not self.isEmpty():
        for idx in range(len(self.data)-1, 0, -1):
            assert (self.data[idx] <= self.data[self._parent(idx)])
```

30	32	35	40	50	90

```
def heapSort(self):
    if not self.isEmpty():
        for idx in range(len(self.data)-1, 0, -1):
            self._swap_(0, idx)
            self._downHeap_(0, idx)
```

Time complexity :-

heap sort — for loop = O(n-1) = O(n)
down heap = O(log n)
Total = O(n log n)

Hashing :-

Mechanism which allows insertion/deletion/search operations with constant average time.

16/10/2023

Array based :-

→ Collision is a common process in this
→ To overcome collision, we will do rehashing
Linear probing → $H_R(x) = H(x) + i$, $i = 0, 1, 2, \ldots n-1$
Quadratic probing → $H_R(x) = H(x) + i^2$. This is used to avoid only one particular area being thickly populated
→ Takes constant average time.
→ Searching → Iterate through hashmap and stop if a blank cell is seen in which ~~was never occup~~ no elem-nt was stored before.

Linked list based :-

→ Hashing & compression function is there
→ While adding a node, add at the head. external chaining
→ Efficiency wise it is better than the array based
→ No limit on the no. of elements stored.

18/10/2023

Hash function = hash code generation + compression

Hash code generation :-

1) Bit code method - sum of ascii (tempo1 = t+e+m+p+0+1
2) Polynomial method - $ta^{n-1} + ea^{n-2} + ma^{n-3} + pa^{n-4} + \ldots$ (33, 37, 39,
3) Cyclic Shift method - bit shift

Compression method :-

1) Division method - hashcode % N, table size is prime no.
2) MAD method - Multiplication, Addition, Division method

classmate
Date 18/10/2023
Page

Cyclic shift :-

```
def cyclic_shift(input):
    hashcode = 0
    for ch in input:
        hashcode = hashcode << 5
        hashcode = hashcode + ch
```

a = 11011

27 << 5

27 / 2^5

a << 1

11011 / 2

10110

16 + 8 + 1 + 4 + 2 = 30

16 + 4 + 1

MAD method :-

p, a, b

p > N (a prime no.)

a (1, (p-1))

b (0, (p+1))

hashcode = i.

MAD = [(a*i + b) % p] % N.

→ Duplicates are not stored.

→ Dictionaries are implemented using hash map in the backend.

```
class extHash:
    class _node_:
        def __init__(self, id, name):
            self.id = id
            self.name = name
            self.next = None
    tableSize = 11
    def __init__(self):
        self.hashtable = [None] * extHash.tableSize
        self.p = 13
        self.a = np.random.randint(1, self.p-1)
        self.b = np.random.randint(0, self.p+1)
```

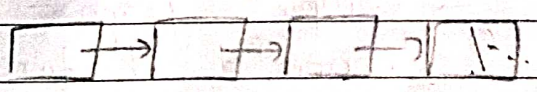
Classmate
Date 19/10/2023
Page

```
def __hashCode__(self, element):
    hashCode = 0
    for ch in element:
        hashcode = hashcode <<5
        hashcode = hashcode + ord(ch)
    return hashcode

def _compression_(self, hcode):
    return(((C.C.hcode * self.a + self.b) % self.p) % exthash.table
                                                        size)

def _hash_(self, element):
    hashCode = _hashcode_(element)
    tableIndex = _compression_(hashcode)
    return tableIndex
```

20/10/2023

```
def isMember(self, id):
    bIndex = _hash_(id)
    node = self.hashtable[bIndex]
    while node:
        if node.id = id:
            break
        else:
            node = node.next
    return node != None

def addElement(self, id, name):
    if not self.isMember(id):
        bIndex = _hash_(id)
        oldAdd = self.hashtable[bIndex]
        node = self._node_(id, name)
        node.next = oldAdd
        self.hashtable[bIndex] = node

def deleteElement(self, id):
```

```
if self.isMember(id):
    bIndex = __hash_(id)
    node = self.hashtable[bIndex]
    while node: id == id:
        if node.id = id:
prev = None
while node:
    if node.id = id:
        prev.n ele = node.name
        prev. if (prev == None):
            self.hashtable[bIndex] = node.next
        else:
            prev.next = node.next
    else:
        prev = node
        node = node.next
return ele

def dele deleteElement (self, id):
    bIndex = self.__hash__(id)
    if self.hashtable[bIndex]:
        if self.hashtable[bIndex].id == id:
            self.hashtable[bIndex] = self.hashtable
                                     [bIndex].next
    else:
        node = self.hashtable[bIndex]
        while node.next != None:
            if node.next.id == id:
                node.next = (node.next).next
            else:
                node = node.next
```



```
def printHashTable(self):
    for idx in range(0, extHash.tableSize):
        node = self.hashTable[idx]
        while node:
            print(node.id)
            node = node.next
        print("\n")
```

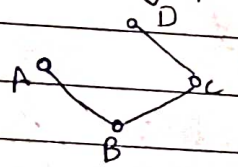
25/10/2023

Graphs:-

→ G = (V, E)
→ Set of vertices (nodes) : V = {$V_1, V_2, V_3, \ldots, V_n$}
→ Set of edges : E = {$e_1, e_2, e_3, \ldots, e_m$}

Types of graphs:-

- Undirected graph - pair of vertices in a edge is unordered (A, B) = (B, A)



- Directed graph - each edge is a directed pair of vertices

→ Self edge / loop edge - start & end vertex is same (u, u)
→ We say "a path exists from V_0 to V_n" if there is a list of vertices [$V_0, V_1, V_2, \ldots, V_n$] such that ($V_i, V_{i+1}$) ∈ E for all $0 \le i \le n$

→ A cycle is a path that begins and ends at the same node.

→ Connected graph - An undirected graph is connected if for all pairs of vertices $u \neq v$, there exists a path from u to v.
→ Complete graph - An undirected graph is complete or fully connected, if for all pairs of vertices $u \neq v$ there exists an edge from u to v.

Date 25/10/2023

Representation of graph:-

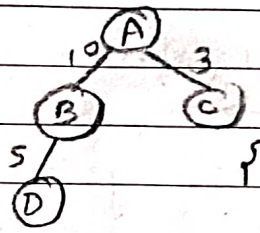
1) Adjacency matrix:-
→ Let G = (V, E) be a graph with n vertices
→ The adjacency matrix of G is a 2-D n by n array
→ If the edge (vi, vj) is in E(G), adj_mat[i][j] = 1
→ If there is no edge in E(G) for (vi, vj), adj_mat[i][j] = 0
→ Adjacency matrix for an undirected graph is symmetric
→ Adjacency matrix for a digraph need not be symmetric

2) Adjacency List:-
→ Each row in adjacency matrix is represented as list.

26/10/2023

Implementation of adjacency list representation of graph

```
import sys

class Graphs:
    class _vertex_:
        def __init__(self, id):
            self.id = id
            self.adjacentNeighbour = {}
        def _isNeighbour_(self, id):
            return (id in self.adjacentNeighbour)
        def _addNeighbour_(self, id, wt=0):
            if not self._isNeighbour_(id):
                self.adjacentNeighbour[id] = wt
        def _listNeighbours_(self):
            return self.adjacentNeighbour.keys()
        def _edgeCost_(self, id):
            if self._isNeighbour(id):
                return self.adjacentNeighbour[id]
            else:
                return sys.maxsize
```



{A:
{B: 10, C: 3},
B:
{A: 10, D: 5},

→ dictionary keyword

classmate
Date 26/10/2023
Page

```
def __init__(self):
    self.vertexCount = 0
    self.adjacencylist = {}
def getVertexCount(self):
    return self.count
def addVertex(self, id):
    if not id in self.adjacencylist:
        newVertex = self.__Vertex(id)
        self.adjacencylist[id] = newVertex
        self.vertexCount += 1
def addEdge(self, from, to, wt=0):
    if not from in self.adjacencylist:
        self.addVertex(from)
    if not to in self.adjacencylist:
        self.addVertex(to)
    self.adjacencylist[from].__addNeighbour__(to, wt)
    self.adjacencylist[to].__addNeighbour__(from, wt)
```

this is only for undirected graph

```
def getEdgeCost(self, from, to):
    if from in self.adjacencylist and to in self.adjacencylist:
        return self.adjacencylist[from].__edgeCost__(to)

def getVertextlast(self):
    return(list(self.adjacencylist.keys()))

def neighborlist(self, id):
    if id in self.adjacencylist:
        return(list(self.adjacencylist[id].__listNeighbours__()))
```

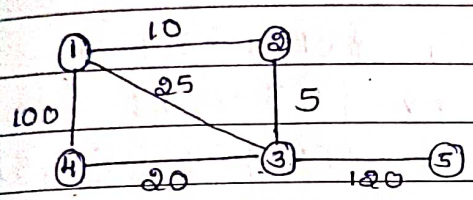
classmate
Date 30/10/2023
Page

Spanning tree :-

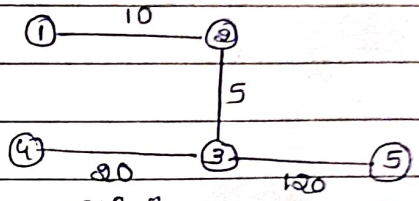
Spanning tree is a acyclic connected subgraph containing all the vertices of a given ~~sub~~graph.
spanning tree should have ~~maximum~~ (n-1) edges, where 'n' is the number of nodes.

Minimum spanning tree :-

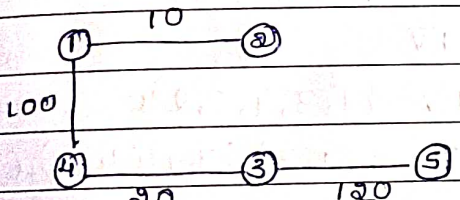
~~Spanning tree is any~~
A spanning tree whose total cost of selected edges is minimum.



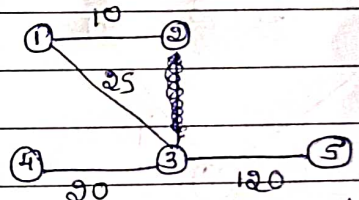
Graph



minimum spanning tree



Spanning tree



Spanning tree

Greedy algorithm :- In later steps, we can't change decision made in previous stages.

Prim's algorithm :-

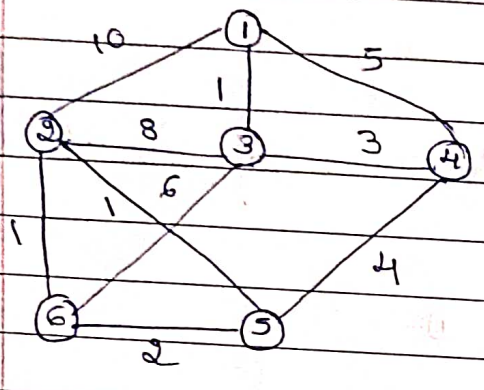
Let TV be set of vertices and T be set of edges
To start with let TV contain an one vertex and T is empty

TV = { any one vertex }
T = { }

while (T contains less than n-1 edges) {
Let (u, v) be least cost edge such that vertex 'u' is

in 'TV' and 'v' is not in 'TV'
add vertex 'v' to TV and edge (u,v) to T
if no such edge break
}

Print (not a minimum spanning tree if T contains less than (n-1) edges)



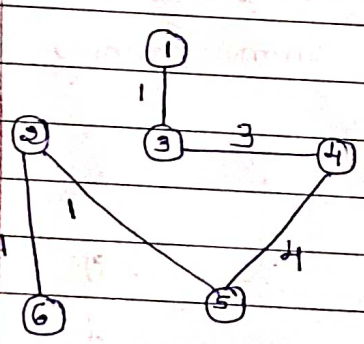
① TV = { 1 }
T = { }

② TV = { 1, 3 }
T = { (1,3) }

③ TV = { 1, 3, 4 }
T = { (1,3) (3,4) }

④ TV = { 1, 3, 4, 5 }
T = { (1,3), (3,4), (4,5) }

⑤ ~~TV = { 1, 3, 4, 5 }~~
~~T = { 1, 3, 4, 5, 2 }~~
TV



MST

⑤ TV = { 1, 3, 4, 5, 2 }
T = { (1,3), (3,4), (4,5), (5,2) }

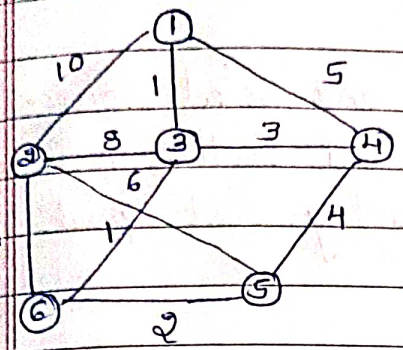
⑥ TV = { 1, 3, 4, 5, 2, 6 }
T = { (1,3), (3,4), (4,5), (5,2), (2,6) }

Kruskal's algorithm :-

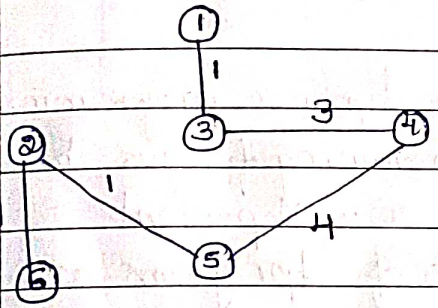
Let T be set edges, E is set representing set of all edges of given graph.
To start with T is empty

T = { }
while (T contains less than n-1 edges and E is not empty){
choose least cost edge (u,v) from E. Remove (u,v) from E
if (u,v) doesn't form cycle in T
Add edge (u,v) to T

else
Discard edge (u, v)
}
print (not a minimum spanning tree if T contains less than (n-1) edges)



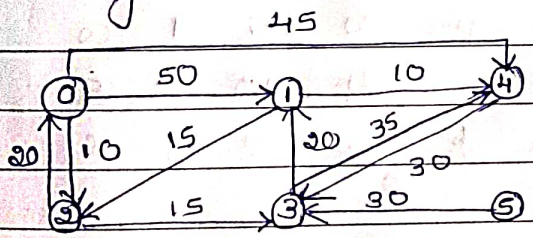
① T = { }
② T = {(1, 3)}
③ T = {(1,3), (2, 5)}
④ T = {(1,3), (2,5), (2,6)}
⑤ T = {(1,3), (2,5), (2,6), (3,4)}
⑥ T = {(1,3), (2,5), (2,6), (3,4), (4,5)}



→ MST.

Dijkstra's algorithm :-

→ Single source all destination
→ Single source shortest path



cost[ ]

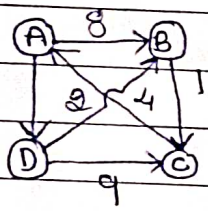
	0	1	2	3	4	5
0	0	50	10	1000	45	1000
1	1000	0	15	1000	10	1000
2	20	1000	0	15	1000	1000
3	0	20	1000	0	35	1000
4	1000	1000	1000	30	0	1000
5	1000	1000	1000	30	1000	0

	0	1	2	3	4	5
0	0	50	10	1000	45	1000
0,2	0	50	10	25	45	1000
0,2,3	0	45	10	25	45	1000
0,2,3,4	0	45	10	25	45	1000

1) Initialize distance ~~matrix~~ vector and select source
2) Select the vertex which is nearest to the source
3) If new path cost is less than existing cost update 'Distance vector' with new value

06/11/2023

All pair shortest path algorithm :-
(Floyd Warshall algorithm)



→ Example for a dynamic programming
→ Dynamic programming follows bottom up
→ Subprograms are dependent
→ ~~We~~ All results of subprograms are stored.

$cost(i,j) = cost(i,k) + cost(k,j)$

Let intermediate vertex = k = A

	A	B	C	D
A	0	8	∞	1
B	∞	0	1	∞
C	4	∞	0	∞
D	∞	2	9	0

dist =

	A	B	C	D
A	0	8	∞	1
B	∞	0	1	∞
C	4	12	0	5
D	∞	2	9	0

dist(A,A) = dist(A,A) + dist(A,A) = 0+0 = 0
dest (A,B) = dest(A,A) + dist(A,B) = 0+8 = 8
dist (A,C) = dist(A,A) + dist(A,C) = 0+∞ = ∞
dist (A,D) = dist(A,A) + dist(A,D) = 0+1 = 1

dist(B,A) = dist(B,A) + dist(A,A) = ∞+0 = ∞
dest(B,B) = dist(B,A) + dist(A,B) = ∞+8
dist(B,C) = dist(B,A) + dist(A,C) = ∞+∞ = ∞

classmate
Date 06/11/2023
Page

dist(B,D) = dist(B,A) + dist(A,D) = ∞ + 1 = ∞

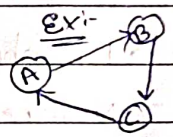
dist(C,A) = dist(C,A) + dist(A,A) = 4 + 0 = 4
dist(C,B) = dist(C,A) + dist(A,B) = 4 + 8 = 12
dist(C,C) = 0
dist(C,D) = dist(C,A) + dist(A,D) = 4 + 1 = 5

dist(D,A) = dist(D,A) + dist(A,A) = ∞
dist(D,B) = dist(D,A) + dist(A,B) = ∞
dist(D,C) = dist(D,A) + dist(A,C) = ∞
dist(D,D) = ∞

Final answer :-

	A	B	C	D
A	0	3	4	1
B	5	0	1	6
C	4	7	0	5
D	7	2	3	0

Ex:-




```
def allpairshortestPath(graph):
    nodes = len(graph)
    dist = graph
    for k in range(nodes):
        for i in range(nodes):
            for j in range(nodes):
                dis[i,j] = min(dist[i,j], dist[i,k] + dist[k,j])
    return dist
```

Time complexity → $O(n^3)$

classmate
Date 06/11/2023
Page

Binary Search :-

→ It works on divide & conquer method.
→ Input should be a sorted array.
→ Time complexity → O(log n)

08/11/2023

Brute Force :-

→ Pattern matching
→ It will return the beginning of the first occurance of the text/pattern
→ Time complexity = O(n*m)

Boyer-Moore algorithm :-
→ Number of iteration less.
→ Number of comparison less
→ Requires additional datastructure dictonary where key is the character and value is the index value of it.
→ Time complexity = O(n+m)

10/11/2023

Knuth-Morris-Pratt :-

→ Keeps track how much pattern is matched
→ It works well only if overlap is present