#04. Write a program to sort an array using merge sort.

Theory: Merge sort is a popular sorting algorithm that uses the divide and conquer strategy. It works by dividing an unsorted array into two halves, sorting each half recursively, and then merging the two sorted halves into a single sorted array. It is similar to the quick sort algorithm as it uses the divide and conquer strategy.

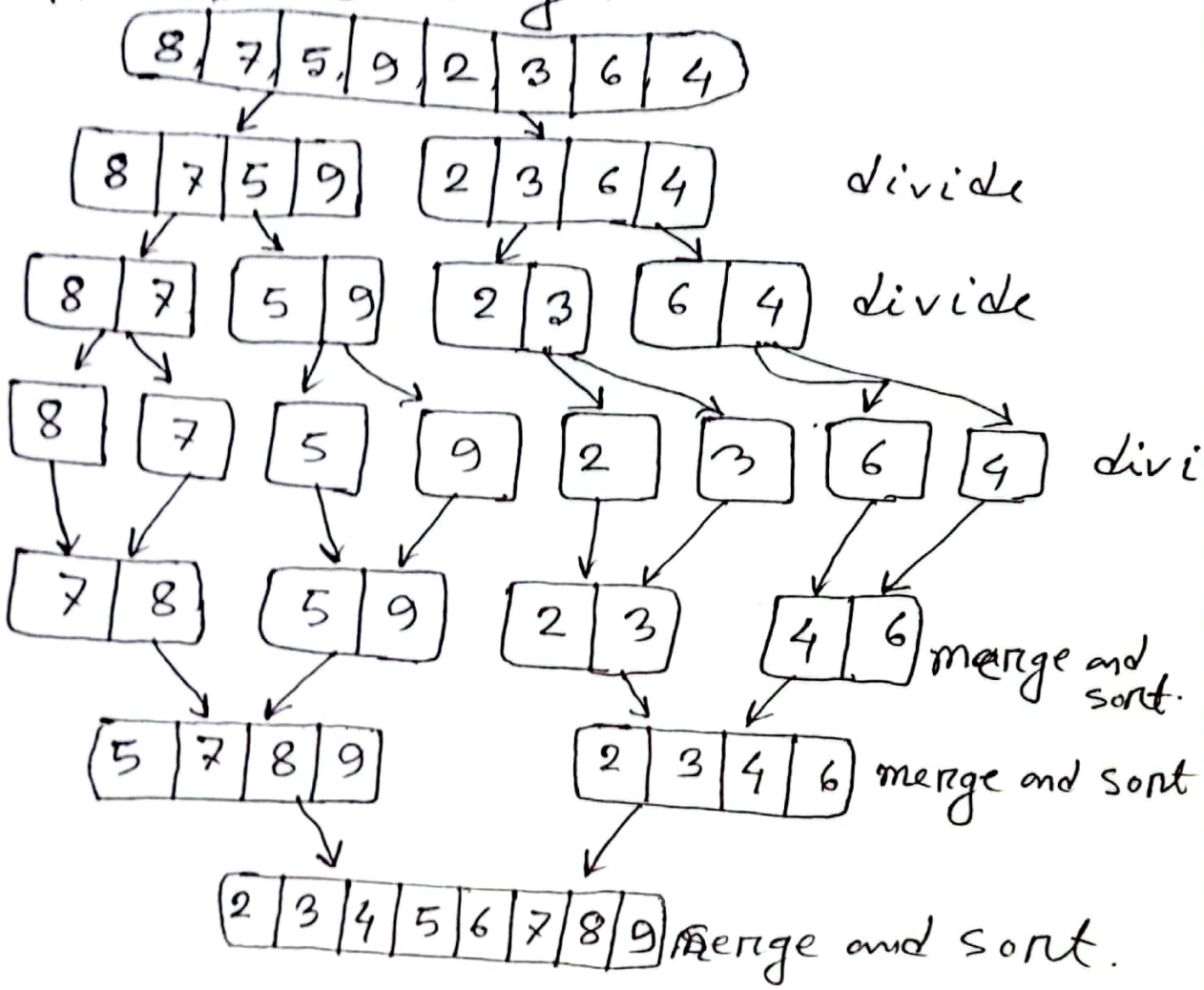
Here are the steps to perform Merge sort:

1. Divide the unsorted array into two halves.
2. Sort the left half recursively using Merge sort.
3. Sort the right half recursively using Merge sort.
4. Merge the two sorted halfes into a single sorted array.

The time complexity of merge sort is $O(n \log n)$, where n is number of element.

For example

Suppose that array A contains 8 element



In this example at first devide again and again to find the single value. the apply the mege and sorid again and again.

Algorithm:

MERGE_SORT(arr, beg, end, N)

where arr is an array with the elements N and beginning of the array is beg and the end or last position is end. mid = int(beg+end)/2 the middle point.

1. If beg < end :
2. Set mid = Int((beg+end)/2).
3. MERGE_SORT(arr, beg, mid)
4. MERGE_SORT (arr, mid+1, end)
5. MERGE (arr, beg, mid, end)
   end of if
6. Exit.

C++ Source code:

```
#include<iostream>
using namespace std;
void merge(int a[], int beg, int mid, int end)
{
  int i,j,k;
  int n1 = mid - beg + 1;
  int n2 = end - mid;
  int leftArray[n1], RightArray[n2];
  i=0;
  j= 0;
  k=beg;
  while(i<n1 && j<n2)
  {
    if(leftArray[i] <= RightArray[j])
    {
      a[k] = leftArray[i];
      i++;
    }
    else
    {
      a[k] = RightArray[j];
      j++;
    }
    k++;
}
```

```
int main()
{
  int a[] = {11,30,24,7,31,16,39,41};
  int n = sizeof(a)/sizeof(a[0]);
cout<<"Before sorting array elements are-\n";
printArray(a,n);
mergesort(a,0,n-1);
cout<<"\n After sorting the array elemet are-\n";
printArray(a,n);
return 0;
}
```

Output:

Before sorting array elements are -

11, 30, 24, 7, 31, 16, 39, 41

After sorting the array element are

7, 11, 16, 24, 30, 31, 39, 41.

*05 Write a program to insert a node into a linked list.

Theory: A linked list is a data structure used to store a collection of elements where each elements (or node) is linked to the next element in the list. There are two part of a linked list. Data or information part and other is next pointer part. Information part store any type of information pointer part connect the ~~list~~ node with next node. The list starts with the head node and end by 0 or null point.

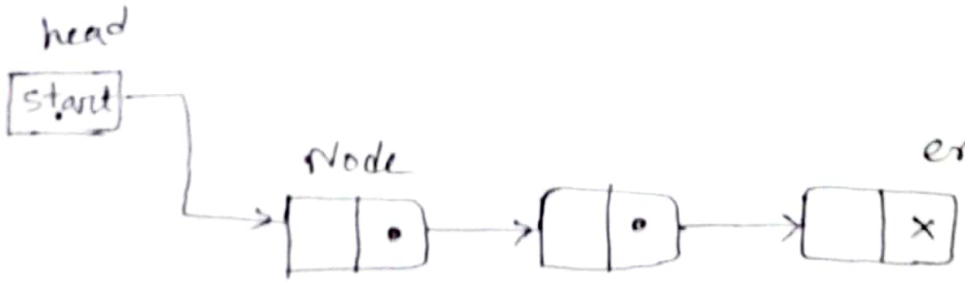


Figure a linked list.

To insert element or node in a linked list. There are two positim First position of the list and any position in the linked list.

To insert at the beginning of a list at first available list's information part insert a info data then remove this node from the list. The avail. list start the next node. Then the start part of the data list connect with the inserting node. the other is the head node.

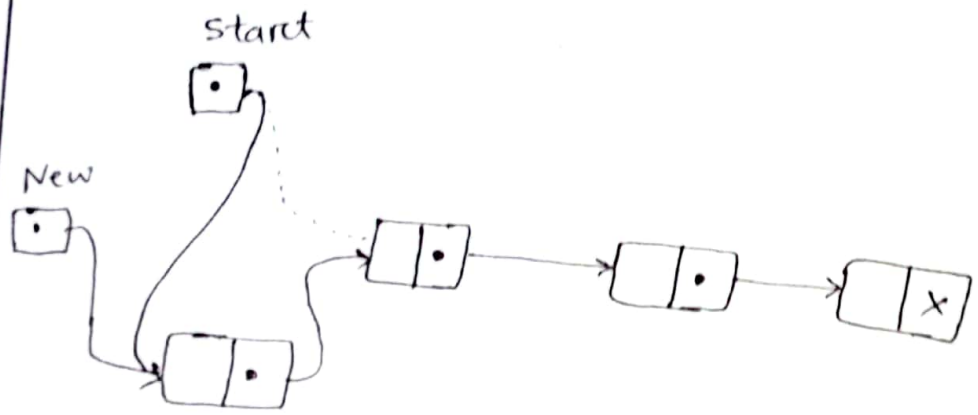


Figure: Insertion at the beginning of the list.

To insert after a given node

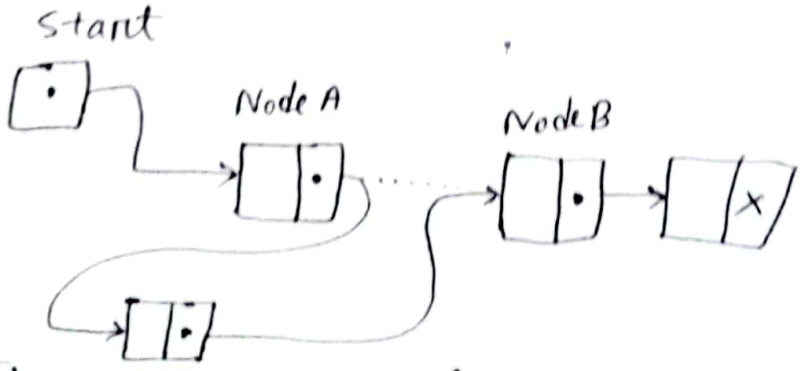


Figure: Insertion after a given node of the list.

Algorithm:

INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM)

This algorithm inserts ITEM so that ITEM follows the node with location LOC, or inserts ITEM as the first node when LOC = NULL

1. [OVERFLOW?] If AVAIL := NULL then write: overflow and exit.
2. [Remove first node from AVAIL list]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM
4. If LOC := NULL, then: Insert as a first node
   Set LINK[NEW] := START and START := NEW.
   Else: [Insert after a node]
   Set LINK[NEW] := LINK[LOC] and LINK[LOC] := NEW.
5. Exit.

Code plaython:

```
Class node:
        def __init__(self, data):
            self.data = data
            self.next = None

def linked_list_traversal(head):
    ptr = head
    while ptr is not none:
            print(f"Element:{ptr.data}")
            ptr = ptr.next

def insert_at_first(head, data):
    ptr = Node(data)
    ptr.next = head
    return ptr.

def insert_at_index(head, data, index):
    ptr = Node(data)
    p = head
    i = 0
    while i != index - 1:
            p = p.next
            p.next = ptr
    return head
```

```
def insert_after at the end(head, data):
    ptr = Node(data)
    p = head
    while p.next is not None:
        p = p.next
    p.next = ptr
    ptr.next = None
    return head.

if name_==_'main':
head = Node(7)
second = Node(11)
third = Node(41)
four = Node(66)

head.next = second.
second.next = third
third.next = four
four.next = None.
print("Linked list before insertion")
Linked_list_traversal(head)

head = insert_after_node(head, third, 45)
print("\n Linked list after insertion")
Linked_list_traversal(head.)
```

Output:

Linked list before insertion

Element : 7

Element : 11

Element : 41

Element : 66

Linked list after insertion.

Element : 7

Element : 11

Element : 41

Element : 45 (inserted).

Element : 66

#06 Write a program to delete a node from linked list.

Theory: A linked list in data structure used to store collection of elements where each elements (or node) is linked with another node using next pointer.

There are two linked list.
A Data linked list and available linked list.
Deletion from linked list is the operation. At first remove a node from the data linked list and then linked it and then add the node in available list.

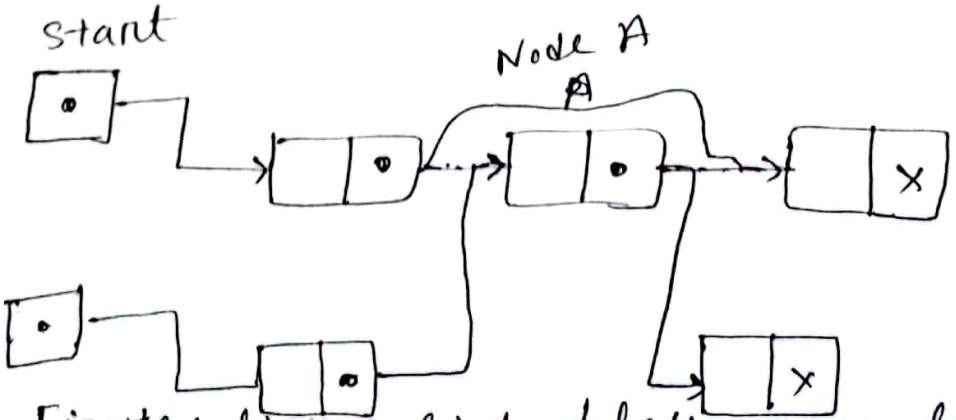


Figure: linked list delation operation.

Algorithm:

DEL(INFO, LINK, START, AVAIL, LOC, LOCP)

The algorithm deletes the node N with location LOC. LOCP is the location of the node which precedure N or when N is the first node, LOCP = NULL.

1. If LOCP = NULL, then:
   set START := LINK[START]
   Else:
   set LINK[LOCP] := LINK[LOC]

2. Return deleted node to the avail list.
   set LINK[LOC] := AVAIL and
   AVAIL := LOC.

3. Exit.

C++ code:

```
#include<iostream>
using namespace std;
struct Node{
    int data;
    Node * next;
};
void deleteNode(Node** head_ref, int key)
{
    Node* temp = *head_ref;
    Node * prev = NULL;
    if(temp != NULL && temp->data==key)
    {
        delete temp;
        return;
    }
    while(temp != NULL && temp-> data
                          != key){
        prev = temp;
        temp = temp-> next;
    }
    if(temp == NULL){
    cout << "key not found in linked
            list." << endl;
    prev->next = temp->next;
    delete temp;
}
```

```
Void push (Node** head_ref, int new_data){
    Node* new_node = New Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}
Void printList (Node* Node) {
  while (node != NULL){
    cout<< node->data<< " ";
    node = node->next;
  }
  cout<< endl;
int main() {
    None* head = NULL;
    push (&head, 7);
    push (&head, 1);
    push (&head, 3);
    push (&head, 2);
cout<<"Linked list before delation:")
print printList(head);
deleteNode (&head,1); printList(head);
cout " Linked list after delete node 1 : ");
```

output:

Linked list before delation: 2 3, 1, 7

linked list after delete node 1: 2,3,7

#8. Write a program to solve the following 0/1 knapsack problem using dynamic programming approach, profits $P=(11,21,31,33)$, weight $W=(2,11,22,15)$, knapsack capacity $c=40$ and number of items $n=4$.

Theory: The 0/1 knapsack problem is a ~~stack~~ classic optimization problem where a set of items with given weights and profits are to be packed in a knapsack with a limited capacity. It is called 0/1 knapsack problem because if the weight count then ~~it is called~~ we fill with 1 otherwise 0. So it is 0/1 knapsack. It can not get fractional weight.

Here is the algorithm for solving the knapsack problem using dynamic programming:-

1. Create a 2D table of size $(n+1) \times (c+1)$ where, n is the no. of items and c is the knapsack capacity or weight.
2. Initilize the first row and colum of the table to 0.
3. For each item i from 1 to n and for each capacity j from 1 to c:
   - If the weight of the item i is greater than the current capacity j set table[i][j] to the valve of the table [i-1][j].
   - $B[i] = \max(B[i-1, w], B[i-1, w - w[i] + v[i]])$
4. The maximum profit that can be obtained is the valve of table [n][c].
5. Exit.

Source code python:

```
def knapsack-01(c, w, p, n):
    table = [[0 for j in range(c+1)] for i in
                                    range(n+1)]

    for i in range(1, n+1):
        for j in range(1, c+1):
            if w[i-1] > j:
                table[i][j] = table[i-1][j]
            else:
                table[i][j] = max(table[i-1][j],
                                  p[i-1] + table[i-1]
                                           [j-w[i-1]])
    included_items = []
    j = c
    for i in range(n, 0, -1):
        if table[i][j] != table[i-1][j]:
            included_items.append(i-1)
            j -= w[i-1]

    return table[n][c], included_items.
```

```
c = 40
w = [2, 11, 22, 15]
p = [11, 21, 31, 35]
n = 4

max_profit, included_items = knapsack_01
                              (c, w, p, n)
print("Maximum profit : ", max_profit)
print("Included item : ", included_items)
```

output:

```
Maximum profit : 75
Included item : [3, 2, 0]
```

*09: Job sequencing with deadlines problem follow the following rules to obtain the feasible solution:

- Each job takes one unit of time.
- If job starts before or at its deadline, profit is obtained, otherwise no profit.
- Goal is schedule jobs to maximize the total profit.

Write a program using greedy method approach to solve this problem when $n = 7$, profit $(p1, p2 \ldots p7) = (30.5, 20, 18, 1, 6, 30)$ and dead line $(d1, d2 \ldots d7) = (1, 3, 4, 3, 2, 1, 2)$

Theory: The job sequencing with deadline problem is a classic problem in scheduling theory. The problem is to schedule a set of jobs, each with a dead line and a profit, to maximize the total profit,

Formally, we are given n jobs, each with a profit pi and a deadline di. The goal is to schedule the jobs within their deadlines such that the total profit is maximized.

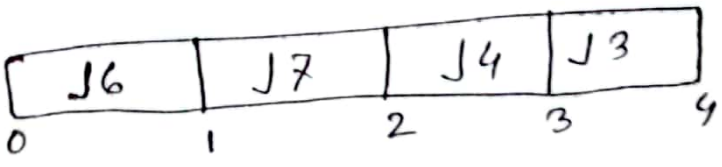
For example
Here is the problem:

Job id:	1	2	3	4	5	6	7
profit:	3	5	20	18	1	6	30
deadline:	1	3	4	3	2	1	2

Sol:

Job id:	7	3	4	6	2	1	5
profit:	30	20	18	6	5	3	1
deadline:	2	4	3	1	3	1	2

Job id	Profit	deadline	slot
J7	30	2	[1,2]
J3	20	4	[3,4]
J4	18	3	[2,3]
J6	6	1	[0,1]
J2	5	3	Rejected
J1	3	1	Rejected
J5	1	2	Rejected



Total profit = (30 + 20 + 18 + 6) = 74.

Algorithm:

1. Sort the jobs in decreasing order of their profits
2. Initialize empty schedule.
3. For each job in the sorted list:
   (a) Find the latest available time slot in the schedule that is before or at the job's deadline.
   (b) If such a time slot exists, schedule the job in that time slot and add its profit with the total profit.
   (c) If no time slot exists, skip the job.
4. Return the total profit and schedule.

The ~~comp~~ time complexity of the algorithm is $O(n^2)$. And we reduce it using job sequence is $O(n \log n)$.

Phy Python code:

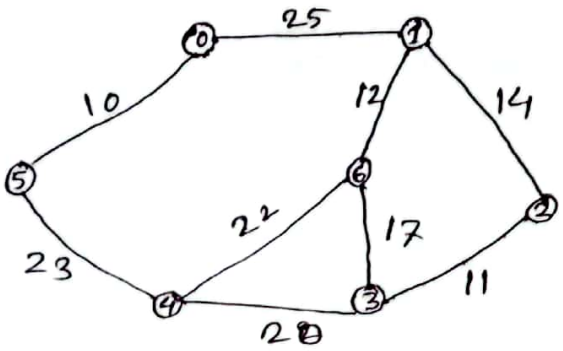
```
def job_sequencing( n, profit, deadlines):
    job = sorted(zip(profits, deadlines), key =
                 lambda x: x[0], reverse = True)
    schedule = [0] * max(deadlines)
    total profit = 0
    for job in jobs:
        profit, deadline = job
        for i in range(deadline-1, -1, -1):
            if schedule[i] == 0:
                schedule[i] = profit
                total profit += profit.
                break.
    return total profit, schedule

n = 7
profits = [3,5, 20, 18, 1,6 30]
deadlines = [1, 3, 4, 3, 2, 1, 2]
total. profit, schedule = job. sequencing
                ( n, profits, schedules)
```

```
print ("Total profit : ", total profit)
print ("Schedule : ", schedule)
```

Output:

```
Total profit : 74
schedule     : [6, 30, 18, 20]
```

#10. Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. Implement Kruskal's algorithm and find the minimum spanning tree for the follow graph.



Theory: Kruskal's minimum spanning tree algorithm is a greedy algorithm used to find the minimum spanning tree in a connected, undirected graph. The minimum spanning tree is a tree that connected all vertices in the graph with the minimum possible total edge weight. The algorithm starts by sorting all the edges of the graph in increasing order of their weight. It then pick the edges with the smallest weight

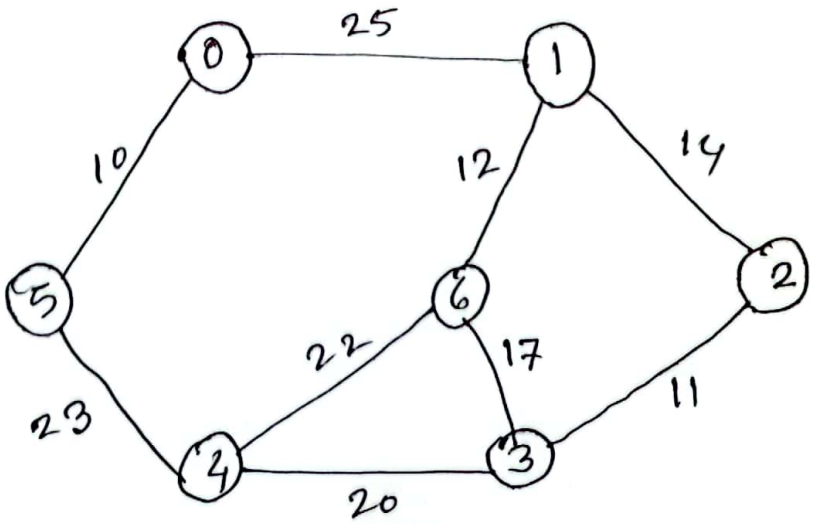
and checks if adding it to the existing tree creates a cycle.

How to solve? or algorithm:

1. Sort all the edges in non decreasing order of their weight.
2. Pick the smallest edge. check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed include it. otherwise discard it.
3. Repeat step (2) until there are (V-1) edges in the spanning tree.

Illustration:

Input graph

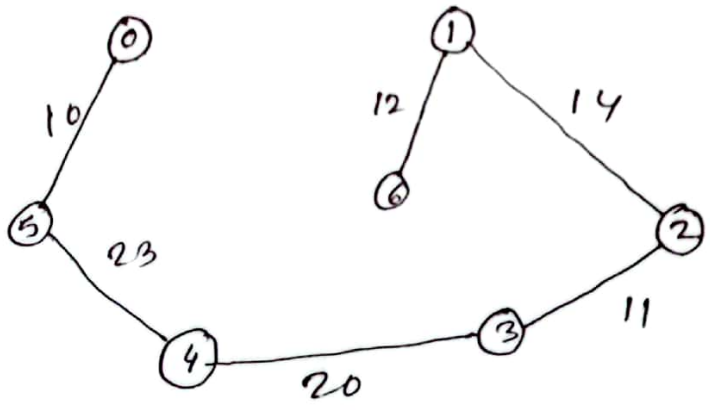


The graph contain 6 vertices and 9 edges. So minimum spanning tree formed will having (9-1) = 8 edges.

After sorting

Source(u)	destination(v)	weight (w)
0	5	10
2	3	11
1	6	12
1	2	14
3	6	17
3	4	20
4	6	22
4	5	23
0	1	25

The minimum spanning tree is



Total cost is = (10+11+12+14+20+23)

= 90

Python code:

```
Class Graph:
    def int(self, vertices):
        self.V = vertices
        self.graph = []

    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] != i:
            parent[i] = self.find(parent, parent[i])
        return parent[i].

    def union(self, parent, rank, x, y):
        if rank[x] < rank[y]:
            parent[x] = y
        elif rank[x] > rank[y]:
            parent[y] = x
        else:
            parent[y] = x.
            rank[x] += 1.
```

1)

```
def kruskal(self):
    result = []
    i = 0
    e = 0
    self.graph = sorted(self.graph,
                        key=lambda item: item[2])
    parent = []
    rank = []

    for node in range(self.V):
        parent.append(node)
        rank.append(0)

    while e < self.V - 1:
        u, v, w = self.graph[i]
        i = i + 1
        x = self.find(parent, u)
        y = self.find(parent, v)

        if x != y:
            e = e + 1
            result.append([u, v, w])
            self.union(parent, rank, x, y)
```

```
minimumcost = 0
print ("Edge in the constructed MST")
for u, v, weight in result:
    minimum cost += weight.
    print ("%d -- %d == %d " % (u, v, weight))
    print ("Minimum Spanning Tree",
                                minimum cost)

if ... name == '__main__':
    g = Graph (7)
    g.addEdge (0, 5, 10)
g.addEdge ( 2, 3, 11)
g.add Edge ( 1, 6, 12)
g. add Edge ( 1, 2, 14)
g. add Edge ( 3, 6, 17)
g. add Edge ( 3, 4, 20)
g. add Edge ( 4, 6 22)
g. add edge ( 4, 5, 23)
g. add edge ( 0, 1, 25)

g. kruskal ()
```

Output:

The Edges in the constructed MST.

0 .... 5 == 10
2 .... 3 == 11
1 .... 6 == 12
1 .... 2 == 14
3 .... 4 == 20
4 .... 5 == 23

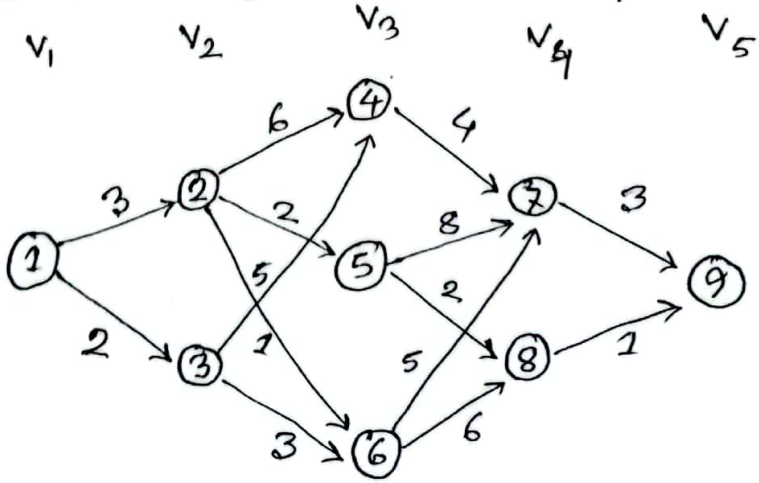
Minimum Spanning Tree 90.

#11. Write a program to find the shortest path from a directed weighted multistage graph using dynamic algorithm.

Theory: A multistage graph is a directed weighted graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage. The vertices of a multistage graph are divided into n number of disjoint subsets

$S = \{S_1, S_2, \cdots S_n\}$

Here we have given a multistage graph, a source to destination, we need to find shortest path from source to distination.



Forward Approach.

$cost(i,j) = \min\{ecost(j,l) + cost(i+1,l)\}$

(i = stage vertex, j = start vertex, l = next vertex of edge)

stage	1	2		3			4		5
Vertex	1	2	3	4	5	6	7	8	9
cost	8	5	10	7	3	7	3	1	0
d	2	5	6	7	8	8	9	9	9

$cost(5,9) = 0$

$cost(4,7) = \min[ecost(7,9) + cost(5,9)]$

$= \min(3+0) = 3$

$cost(4,8) = \min[ecost(8,9) + cost(5,9)]$

$= \min(1+0) = 1$

$cost(3,4) = \min[ecost(4,7) + cost(4,7)]$

$= \min(4+3) = 7$

$cost(3,5) = \min[ecost(5,7) + cost(4,7),$
$ecost(5,8) + cost(4,8)]$

$= \min[8+3, 2+1] = 3$

$cost(3,6) = \min[ecost(6,7) + cost(4,7),$
$ecost(6,8) + cost(4,8)]$

$= \min[5+3, 6+1] = 7$

$cost(2,2) = \min[ecost(2,4) + cost(3,4),$
$ecost(2,5) + cost(3,5),$
$ecost(2,6) + cost(3,6)]$

$= \min[6+7, 2+3, 1+7)$

$= 5$

$cost(2,3) = \min(ecost(3,4) + cost(3,4),$
$ecost(3,6) + cost(3,6)]$

$= \min(5+7, 3+7] = 10$

$cost(1,1) = \min(ecost(1,2) + cos(2,2),$
$ecost(1,3) + cos(2,3)]$

$= \min(3+5, 2+10) = 8$

Shortage path

$d(1,1) = 2$

$d(2,2) = 5$

$d(3,5) = 8$

$d(4,8) = 9$

~~$d(5,9) = 9$~~

Shortage path is $1 \rightarrow 2 \rightarrow 5 \rightarrow 8 \rightarrow 9$

Total weight is $= 3 + 2 + 2 + 1$

$= 8$

Python code:

```
def shortestdistance(graph):
    global INF
    dist = [0]*N
    dist[N-1] = 0
    for i in range(N-2, -1, -1):
    dist[i] = INF
    for j in range(N):
        if graph[i][j] == INF:
            continue
            dist[i] = min(dist[i], graph[i,j]
                          + dist[j])
    return dis[0].

N = 9
INF = 999999999
```

```
graph=[ [INF, 3, 2, INF, INF, INF, INF, INF, INF],
        [INF, INF, INF, 6, 2, 1, INF, INF, INF],
        [INF, INF, INF, 5, INF, 3, INF, INF, INF],
        [INF, INF, INF, INF, INF, INF, 4, INF, INF],
        [INF, INF, INF, INF, INF, INF, 8, 2, INF],
        [INF, INF, INF, INF, INF, INF, INF, 6, INF],
        [INF, INF, INF, INF, INF, INF, INF, INF, 3],
        [INF, INF, INF, INF, INF, INF, INF, INF, 1]]

print("shortest distance is:", shortestDist(graph))
```

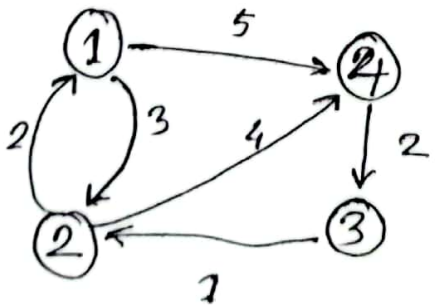
Output: shortest distance is: 8

#12. Write a program to find the all pair shortage path from a graph using Floyd Warshall's Algorithm.

Theory: Floyd Warshall's Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But does not work for the graph with negative cycles.

Floyd-warchshall algorithm is also called as Roy-Floyd algorithm or WFI algorithm.

Here a directed weighted graph.



At first create A° matrix with 4*4 dimension. Here row and culumn are i and j.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

using A^0 matrix create A^1 matrix.

The formula is

$$A^1[i,j] = \min\left[A^0[i,j],\ A^0[i,k] + A^0[k,j]\right]$$

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

similarly

$$A^2 = \begin{vmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{vmatrix}$$

$$A^3 = \begin{vmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{vmatrix}$$

$$A^4 = \begin{vmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{vmatrix}$$

A^4 gives the shortest path between each pair of vertices.

Algorithm:

N = Number of vertices

A = Matrix of dimension $n \times n$.

```
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            A^k[i,j] = min(A^(k-1)[i,j], A^(k-1)[i,k] + A^(k-1)[k,j])
return A.
```

Python code:

```
nV = 4
INF = 999999999
def floyd_warshall(G):
    distance = list(map(lambda i : list(map
                (lambda j: j, i)), G))

    for k in range(nV):
        for i in range(nV):
            for j in range (nV):
                distance[i][j] = min(distance
                [i][j], distance[i][k]+distag[k][j])

    print_solution(distance)
print("The shortage path matrix is:")
def print_solution (distance)
    for i in range (nV):
        for j in range (nV):
            if (distance[i][j] == INF):
                print ("INF", end = " ")
            else print (distance[i][j], end="
        print(" ")
```

```
G = [[ 0, 3, INF, 5],
     [ 2, 0, INF, 4],
     [INF, 1, 0, INF],
     [INF, INF, 2, 0]]
```

floyd_warshall(G)

output:

The shortest path matrix is

```
0   3   7   5
2   0   6   4
3   1   0   5
5   3   2   0
```

#13. The eight queen's puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens attack each other. Thus a solution requires that no two queens share the same row, column, or diagonal. The eight queen's puzzle is an example of the more general n-queens problem of placing n queens on an n×n chessboard, where solutions exist for all natural numbers n with the exception of 2 and 3. Write a program to solve the n queen's problem.

Theory: The N Queen is the problem of placing N chess queens on an n×n chessboard so that no two queens attack ~~each~~ each other. For example the following is a solution for 4 Queen problem.

Algorithm:

1. Start in the left most column.
2. If all queen are placed return true
3. Try all rows in the current column.
   (a) If the queen can be placed safely in the row then mark this [row, column] as a part of the solution and recursively check if placing queens here leads to a solution.
   (b) If placing queens in [row, column] leads to a solution then return true.
   (c) If placing queen does not lead a solution then unmark this [row, column].
4. If all rows have been tried and nothing worked return false to trigger backtracking.

Python code:

```
class QueenchessBoard:
    def __int__(self, size):
        self.size = size
        self.colums = []

    def def place_in_next_row(self, colum):
        self.colums.append(column)

    def remove_in_current_row(self):
        return self.columns.pop()

    def is_this_colum_safe(self, colum):
        row = len(self.columns)
        for queen_colum in self.colums
            if column == queen_colum:
                return false.

        for queen_row, queen_colum in
                enumerate(self.colu)
            if queen_colum = colum_row:
                return false.
```

```
        for queen_row, queen_colum in.
                                enumerate(self.colum):
            if ((self.size - queen_colum) - queen_
                                    _row == (self.size_colum) - row):
                return False.

    return True.

def display (self):
    for row in rang(self.size):
        if column == self.columns[row]:
            print ("Q', end = ' ')
        else:
            print ('.', end = ' ')

    print()

def solve_queen (size):
    board = QueenchessBoard (size)
    number_of_solution = 0

    row = 0
    column = 0

    while True:
        while colum < size:
            if board is this colum (colum):
```

```
            board.place_in_next_row(column)
            row += 1
            column = 0
            break
        else:
            column += 1
    if column == size or row == size:
        if row == size:
            board.display()
            print()
            number_of_solution += 1.
            board.remove_in_current_row()
            row -= 1.

try
print("Number of solutions: ' number
                                of solutio)

n = int(input('Enter n: '))
solve_queen(n)
```

input:

Enter n: 8

Output:

Q - - - - - - -
- - - - Q - - -
- - - - - - - Q
- - Q - - - - -
- - - - - - Q -
- - - Q - - - -
- Q - - - - - -
- - - - - Q - -

. . . . . .

~~like all~~

Number of solution is: 92