EXPERIMENT NUMBER- 01

AIM: Given N items with their corresponding weights and values, and a package of capacity C, choose either the entire item or fractional part of the item among these N unique items to fill the package such that the package has maximum value.

DESCRIPTION: A greedy algorithm is the most straightforward approach to solving the knapsack problem, in that it is a one-pass algorithm that constructs a single final solution.

Consider the following instance of the Knapsack problem: N=3, W=50, (v1,v2,v3)=(60,100,120), (w1,w2,w3)=(10,20,30).

Applying greedy strategy to solve,

$\sum Wixi = 30*1 + 20*1 + 10*0 <=50$

$\sum Pixi = 120*1 + 100*1 + 60*0 = 120+100 = 220$

Thus, the optimal solution is 220.

ALGORITHM: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

1. for i = 1 to n
   1.1. do x[i] = 0
2. weight = 0
3. for i = 1 to n
   3.1. if weight + w[i] ≤ W then
      3.1.1. x[i] = 1
      3.1.2. weight = weight + w[i]
   3.2. else
      3.2.1. x[i] = (W - weight) / w[i]
      3.2.2. weight = W
      3.2.3. break
4. return x

PROGRAM:

```
from numpy import array

class Item:
    def __init__(self, price, weight, x):
        self.price = price
        self.weight = weight
        self.x = x

def merge(l, low, high, mid):
    temp = sorted(l[low:high+1], key=lambda
item: item.price/item.weight)
    l[low:high+1] = temp
```

```
def merge_sort(l, low, high):
    if low < high:
        mid = (low + high) // 2
        merge_sort(l, low, mid)
        merge_sort(l, mid + 1, high)
        merge(l, low, high, mid)
    return l

def knapsack(l, m):
    u = m
    vector = [0] * len(l)

    for i in range(len(l)):
        if u - l[i].weight > 0:
            vector[i] = 1
            l[i].x = 1
            u -= l[i].weight
        else:
            if i < len(l) and u > 0:
                s = (u) / l[i].weight
                vector[i] = round(s, 2)
                l[i].x = vector[i]
                break

    return vector

# Input
n = int(input("Input the size of array: "))
l = [Item(int(input("Price: ")),
int(input("Weight: ")), 0) for _ in range(n)]
s = list(l)
m = int(input("Enter the max capacity: "))

# Execution
merge_sort(l, 0, len(l) - 1)
result = knapsack(l, m)

profit = sum(item.price * result[i] for i, item in
enumerate(l))
arr = [item.x for item in s]

# Output
print("The result vector x is:", arr)
print("The max profit is obtained:", profit)
```

TEST CASES:

Successful test cases:

```
Input the size of array: 3
Price: 60
Weight: 10
Price: 100
Weight: 20
Price: 120
Weight: 30
Enter the max capacity: 50
The result vector x is: [1, 1, 0.67]
The max profit is obtained:  240.4

Process finished with exit code 0
```

```
Input the size of array: 3
Price: 25
Weight: 18
Price: 24
Weight: 15
Price: 15
Weight: 10
Enter the max capacity: 20
The result vector x is: [0, 1, 0.5]
The max profit is obtained:  31.5

Process finished with exit code 0
```

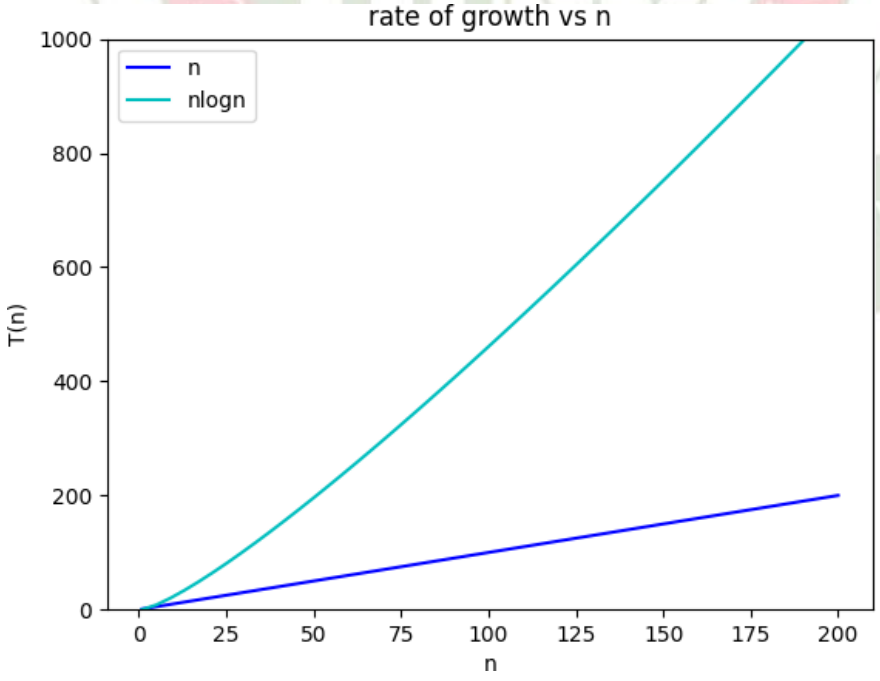
Unsuccessful test cases:

```
Input the size of array: 3
Price: 23
Weight: 8
Price: 35
Weight: 9
Price: 62
Weight: 12
Enter the max capacity: 24
Traceback (most recent call last):
  File "C:\Users\saphalya peta\OneDrive\Desktop\knapsack.py", line 74, in <module>
    merge_sort(l,0,len(l)-1)
  File "C:\Users\saphalya peta\OneDrive\Desktop\knapsack.py", line 34, in merge_sort
    merge_sort(l,low,mid)
  File "C:\Users\saphalya peta\OneDrive\Desktop\knapsack.py", line 36, in merge_sort
    merge(l,low,high,mid)
  File "C:\Users\saphalya peta\OneDrive\Desktop\knapsack.py", line 16, in merge
    if ((l[i].price/l[i].weight)>(l[j].price/l[j].weight)):
ZeroDivisionError: division by zero

Process finished with exit code 1
```

```
Input the size of array: 2
Price: 12
Weight: 9999999999
Price: 8
Weight: 545454678
Enter the max capacity: 20
The result vector x is: [0, 0.0]
The max profit is obtained:  0.0

Process finished with exit code 0
```

ANALYSIS: If the items are already sorted in decreasing order of Vi/wi, then the loop takes a time O(n), therefore, the total time including the sort is O(n log n).If we keep the items in heap with largest Vi/wi at the root. Then,

1. Creating the heap takes O(n) time.
2. Loop takes O(n log n) time.



CONCLUSION: Fractional knapsack generates an optimal solution when we consider the ratio of profits and weights, using the greedy approach. The code successfully executes the fractional knapsack of greedy algorithm with maximum profit.

EXPERIMENT NUMBER- 02

AIM: A Test has 'N' questions with a heterogeneous distribution of points. The test-taker has a choice as to which questions can be answered. Each question Qi has points Pi and time Ti to answer the question, where $1 \leq i \leq N$. The students are asked to answer the possible subsets of problems whose total point values add up to a maximum score within the time limit 'T'. Determine which subset of questions gives student the highest possible score.

DESCRIPTION:
In 0-1 knapsack problems, either an item is completely filled in or no item is filled in at all. Fractions of items are not considered. So here, either a question is answered or not, no other possibility is considered. We take questions with highest value and adding them and their points as long as the points is less than or equal to the maximum score. If the score is less than the maximum score but we cannot afford to add the next question as a whole, then we stop there and find the maximum score obtained. We can use greedy approach to solve such problems as well. In this approach, we calculate the time by value ratio for each question and sort them according to this ratio. We then start taking questions with highest values of this ratio and start adding them until we can't add the next question.

Consider the following instance:
M=60, n=3, Pi= {100,280,120}, Wi= {10,40,20}

Pi/Wi= (10,7,6)

So, ∑Wi.xi= 10*1 + 40*1 + 20*0
=10+40
=50 <=60=M

So, ∑Pi.xi= 100*1 + 280*1
=380

Therefore, (x1,x2,x3)= (1,1,0) and optimal profit= 380

ALGORITHM:
1. Start
2. Input the values of n(no of questions) and time_limit.
3. Input the points and time for the question from the user.
4. For I in n: 4.1. Ptratio.append(p[i][0]/p[i]p[1])
5. X=[0]*n
6. While time_limit>0 then
    6.1. Max_value=max(ptratio)
    6.2. max1=ptratio.index(max_value)
    6.3. if time_limit>pt[max1][1] then
        6.3.1. x[max]=1
    6.4. else
        6.4.1. ptratio[max1]=0
        6.4.2. time_limit=time_limit-pt[max1][1]
7. for I in n
    7.1. p=p+x[i]*pt[i][0]
8. print total profit and feasible solution

9. stop.

PROGRAM:

```
class Test:
    def __init__(self, p, t, x):
        self.p, self.t, self.x = p, t, x

def merge(l, low, high, mid):
    temp = sorted(l[low:high + 1], key=lambda
item: item.p / item.t)
    l[low:high + 1] = temp

def merge_sort(l, low, high):
    mid = (low + high) // 2
    if low < high:
        merge_sort(l, low, mid)
        merge_sort(l, mid + 1, high)
        merge(l, low, high, mid)
    return l

def knapsack(l, m):
    u, total = m, 0
    vector = [0] * len(l)
    for i, item in enumerate(l):
        if u - item.t > 0:
            vector[i], item.x, u = 1, 1, u - item.t
        else:
            item.x = 0
            break
    return vector

# Input
l = [Test(int(input("Points: ")), int(input("Time:
")), 0) for _ in range(int(input("Input the size of
array: ")))]
s, m = list(l), int(input("Enter the max marks: "))

# Execution
merge_sort(l, 0, len(l) - 1)
result = knapsack(l, m)

score = sum(item.p * result[i] for i, item in
enumerate(l))
arr = [item.x for item in s]

# Output
print("The result vector x is:", arr)
print("The highest possible score is:", score)
```

TEST CASES:

Successful test cases:

```
Input the size of array: 3
Points: 100
Time: 10
Points: 280
Time: 40
Points: 120
Time: 20
Enter the max marks: 60
The result vector x is: [1, 1, 0]
The highest possible score is:  380
```

```
Input the size of array: 3
Points: 60
Time: 10
Points: 100
Time: 20
Points: 120
Time: 30
Enter the max marks: 50
The result vector x is: [1, 1, 0]
The highest possible score is:  160
```

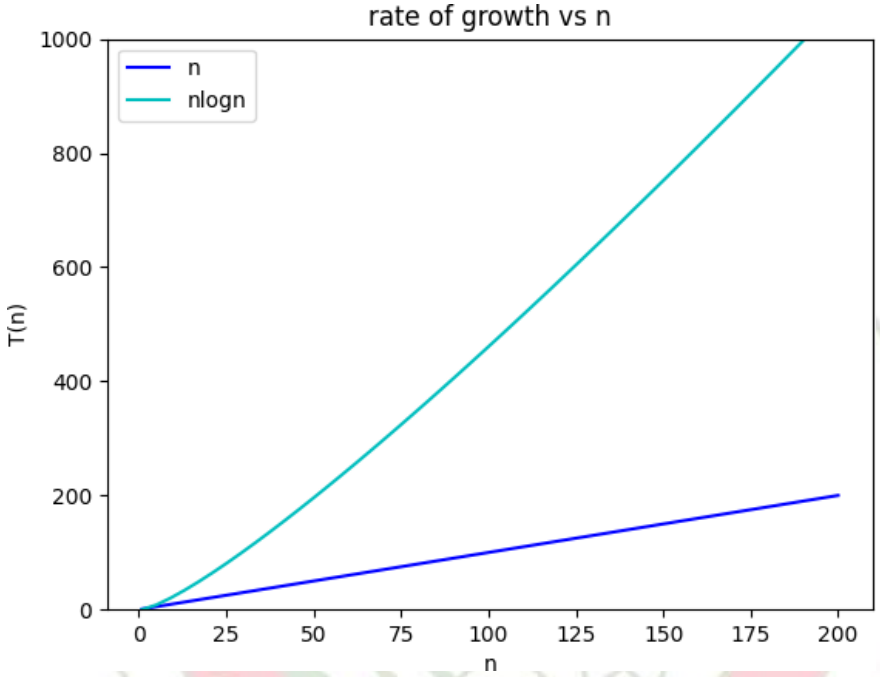
Unsuccessful test cases:

```
Input the size of array: 2
Points: 0
Time: 12
Points: 0
Time: 9
Enter the max marks: 1
The result vector x is: [0, 0]
The highest possible score is:  0
```

```
Input the size of array: 2
Points: 12
Time: 1300
Points: 24
Time: 500
Enter the max marks: 100
The result vector x is: [0, 0]
The highest possible score is:  0
```

ANALYSIS: If the items are already sorted in decreasing order of Vi/wi, then the loop takes a time O(n), therefore, the total time including the sort is O(n log n).If we keep the items in heap with largest Vi/wi at the root. Then,

1. Creating the heap takes O(n) time.
2. Loop takes O(n log n) time.



CONCLUSION: 0/1 knapsack generates an optimal solution when we consider the ratio of profits and weights, using the greedy approach. The code successfully executes the 0/1 knapsack of greedy algorithm with the highest possible score.

EXPERIMENT NUMBER- 03

AIM: Given a bunch of projects, where every project has a deadline and associated profit if the project is finished before the deadline. It is also given that every project takes one month duration, so the minimum possible deadline for any project is 1 month. In what way the total profits can be maximized if only one project can be scheduled at a time.

DESCRIPTION: Let us consider, a set of n given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline by greedy method. These jobs need to be ordered in such a way that there is maximum profit.

There are n jobs to be processed on a machine.
• Each job i has a deadline di≥ 0 and profit pi≥0.
• Pi is earned if the job is completed by its deadline.
• The job is completed if it is processed on a machine for unit time.
• Only one machine is available for processing jobs.
• Only one job is processed at a time on the machine.
The optimal solution of this algorithm is a feasible solution with maximum profit=∑Pi. Thus, D(i)>0 for 1⩽i⩽n.
Initially, these jobs are ordered according to profit, i.e., p1⩾p2⩾p3⩾...⩾pn.

Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	**J2**	**J1**	**J4**	**J3**	**J5**
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, from Gantt chart,
First we select J_2, as it can be completed within its deadline and contributes maximum profit. Next, J_1 is selected as it gives more profit compared to J_4.
In the next clock, J_4 cannot be selected as its deadline is over, hence J_3 is selected as it executes within its deadline.
The job J_5 is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (J_2, J_1, J_3), which are being executed within their deadline and gives maximum profit.

Therefore, Total profit of this sequence is 100 + 60 + 20 = 180.

ALGORITHM:

1. Start
2. Function JS(d,j,n)
3. D[0]=J[0]=0
4. J[1]=1
5. k=1
6. for i=2 to n then
   6.1. r=k
   6.2. while ((d[J[r]]>d[i]) and (d[J[r]] != r)) then
      6.2.1. r=r-1
      6.2.2. if ((d[J[r]]<+d[i]) and (d[i]>r)) then
         6.2.2.1. for q=k to r+1 step -1 then
         6.2.2.2. J[q+1]=J[q]
         6.2.2.3. J[r+1]=i
         6.2.2.4. k=k+1
7. return k
8. Stop

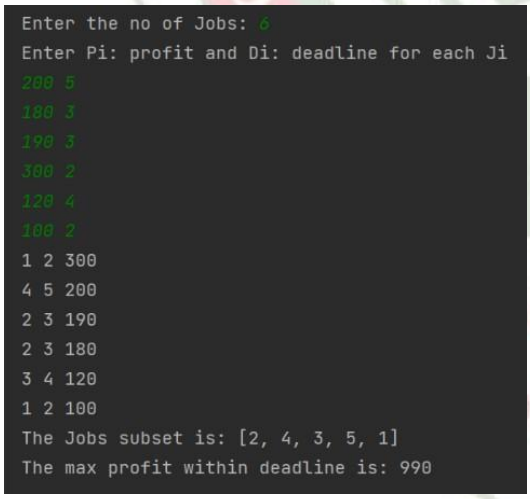
PROGRAM:

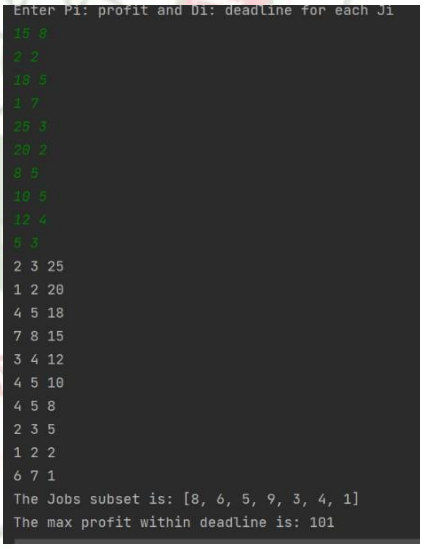
```
class Job:
    def __init__(self, profit, deadline, q_no):
        self.profit, self.deadline, self.q = profit, deadline, q_no

def merge(l, low, high, mid):
    i, j, temp = low, mid + 1, []
    while i <= mid and j <= high:
        temp.append(l[i] if l[i].profit > l[j].profit else l[j])
        i += 1 if l[i].profit > l[j].profit else 0
        j += 1 if l[j].profit > l[i].profit else 0
    temp.extend(l[i:mid + 1])
    temp.extend(l[j:high + 1])
    for i, item in enumerate(temp):
        l[low + i] = item
    return l

def merge_sort(l, low, high):
    mid = (low + high) // 2
    if low < high:
        merge_sort(l, low, mid)
        merge_sort(l, mid + 1, high)
        merge(l, low, high, mid)
    return l

def job_scheduling(l):
    l = merge_sort(l, 0, len(l) - 1)
    max_deadline = max((job.deadline for job in l), default=0)
    x = [None] * max_deadline
    i, k = 0, 0
    while k < len(l):
        ind = l[i].deadline - 1
        k += 1
```

```
if x[ind] is None:
            x[ind] = l[i]
        else:
            count = ind
            for j in range(count, -1, -1):
                if x[j] is None:
                    x[j] = l[i]
                    break
        i += 1
    return x

# Input
n = int(input("Enter the number of Jobs: "))
qs = [Job(*map(int, input("Enter profit, deadline, q_no: ").split())) for _ in range(n)]

# Execution
result = job_scheduling(qs)
total, l = sum(job.profit for job in result if job), [job.q for job in result if job]

# Output
print("The Jobs subset is:", l)
print("The max profit within deadline is:", total)
```

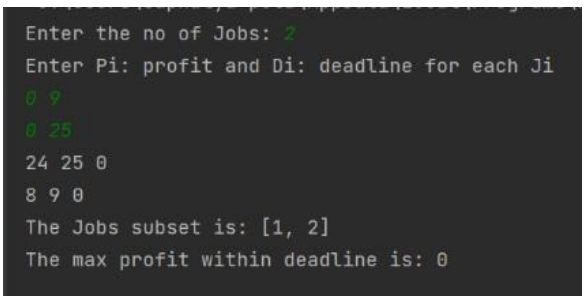
TEST CASES:

Successful test cases:

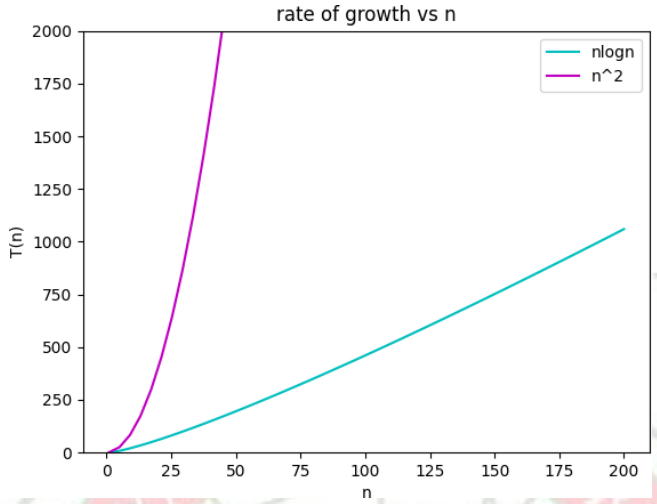




Unsuccessful test cases:



ANALYSIS: In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is O(n^2).



CONCLUSION: Job Scheduling with deadlines generates an optimal solution when we consider the sorted profits, using the greedy approach. The code successfully executes the Job Scheduling with deadlines of greedy algorithm with the maximum possible earned profit.

EXPERIMENT NUMBER- 04

AIM: A Test has 'N' questions with a heterogeneous distribution of points. The test-taker has a choice as to which questions can be answered. Each question Qi has points Pi and time Ti to answer the question, where $1 \leq i \leq N$. The students are asked to answer the possible subsets of problems whose total point values add up to a maximum score within the time limit 'T'. Determine which subset of questions gives student the highest possible score using Dynamic Programming.

DESCRIPTION: We use 0/1 Knapsack algorithm to solve this problem using dynamic programming approach. Given time and points of n questions, put these questions in a knapsack of capacity T to get the maximum total value in the knapsack. In other words, given two integer arrays p[0..n-1] and t[0..n-1] which represent points and times associated with n questions respectively. Also given an integer T which represents knapsack capacity or maximum time limit, find out the maximum value subset of p[] such that sum of the times of this subset is smaller than or equal to T. You cannot break an questions, either pick the complete item or don't pick it (0/1 property). The best way to solve 0/1 knapsack is using dynamic programming. The following are the steps involved in dynamic approach.

Step 1: Decompose the problem into smaller problems.

Step 2: Recursively define the value of an optimal solution in terms of solutions to smaller problems.

Step 3: Compute the value of an optimal solution, typically in a bottom-up fashion using a table structure.

ALGORITHM:

1. START
2. Input Time limit and points
3. Create a matrix B [1…...m+1,1… n+1] where n is the number of questions and m is the max time limit
4. for i in 1 to n+1 then
   - 4.1. for j in 0 to m + 1 then
     - 4.1.1. if T[i – 1] <= j
       - 4.1.1.1. B[i][j] ← max{B[i – 1, j], B[i – 1, j – Ti] + Pi}
     - 4.1.2.else
       - 4.1.2.1. B[i][j] ← B[i – 1][j]
5. After creating a table, the logic below is used to get the solution vector.
6. Max points = B[i][j]
7. k=j
8. while i>0 and k>0 then
   - 8.1. if B[i][k]!=B[i-1][k] then
     - 8.1.1. x[i – 1] ← 1
     - 8.1. 2. i ← 1
     - 8.1.3. k- ← T[i – 1]
   - 8.2. else
     - 8.2.1 i- ← 1
9. print(x)
10. STOP

PROGRAM:

```
def dp_knapsack(l, m):
    B = [[0 for i in range(m + 1)] for j in range(len(l) + 1)] for i in
    range(len(l)):
        for t in range(1, m + 1): if t < 0:
                B[i][t] = 0
            else:
                if l[i][0] > t:
                    B[i][t] = B[i - 1][t]
                elif l[i][0] <= t:
                    B[i][t] = max(B[i - 1][t], B[i - 1][t - l[i][0]] + l[i][1])
    print("The table we get from tabulation method is:") for i in
    range(-1, n - 1):
        print(B[i])
    x = [0 for i in range(len(l))] i =
    len(l) - 1
    k = m
    max_pts = B[i][k]
    while (i >= 0 and k > 0): if B[i][k]
        != B[i - 1][k]:
            x[i] = 1
            k -= l[i][0]
    # print(i,B[i][k],i-1,B[i-1][k]) i = i - 1
    print("The max points is:", max_pts) print("The
    solution vector is:", x)

l = []
m = int(input("Enter the max time: "))
n = int(input("Enter the number of questions: ")) print("Enter the
time and points of each question: ") for i in range(n):
    a, b = [int(x) for x in input().split()] l.append([a, b])
# print(l)
dp_knapsack(l,m)
```

TEST CASES:
Successful test cases:

```
Enter the max time: 5
Enter the number of questions: 4
Enter the time and points of each question:
2 3
3 4
4 5
5 6
The table we get from tabulation method is:
[0, 0, 0, 0, 0, 0]
[0, 0, 3, 3, 3, 3]
[0, 0, 3, 4, 4, 7]
[0, 0, 3, 4, 5, 7]
The max points is: 7
The solution vector is: [1, 1, 0, 0]
```

```
Enter the max time: 6
Enter the number of questions: 3
Enter the time and points of each question:
2 1
3 2
4 5
The table we get from tabulation method is:
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 1, 1, 1, 1]
[0, 0, 1, 2, 2, 3, 3]
The max points is: 6
The solution vector is: [1, 0, 1]
```

Unsuccessful test cases:

```
Enter the max time: 2
Enter the number of questions: 2
Enter the time and points of each question:
21 0
37 0
The table we get from tabulation method is:
[0, 0, 0]
[0, 0, 0]
The max points is: 0
The solution vector is: [0, 0]
```

ANALYSIS: Time Complexity of 0/1 knapsack using dynamic programming is O(N*W), where 'N' is the number of weight element and 'W' is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.

CONCLUSION: The algorithm solves the 0/1 knapsack using tabulation method of dynamic programming, it consumes more memory and time but it will always give the optimal solution compared to the greedy approach. 0/1 knapsack generates an optimal solution when we consider the ratio of profits and weights, using the dynamic programming. The code successfully executes the 0/1 knapsack using dynamic programming with the highest possible score**.**

EXPERIMENT NUMBER- 05

AIM: An X-ray telescope (XRT) is a telescope that is designed to observe remote objects in the X-ray spectrum. In order to get above the Earth's atmosphere, which is opaque to X-rays, X-ray telescopes must be mounted on high altitude rockets, balloons or artificial satellites.
Planets, stars and galaxies and the observations are to be made with telescope. Here the process of rotating equipment into position to observe the objects is called slewing. Slewing is a complicated and time-consuming procedure handled by computer driven motors. The problem is to find the tour of the telescope that moves from one object to other by observing each object exactly once with a minimum total slewing time.

DESCRIPTION:
In the traveling salesman Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost c (i, j) to travel from the city i to city j. The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called Traveling-salesman problem (TSP).
Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For n number of vertices in a graph, there are (n - 1)! number of possibilities.
Example
In the following example, we will illustrate the steps to solve the travelling salesman problem.
Analysis
From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$S = \Phi$
$Cost(2,\Phi,1)=d(2,1)=5Cost(2,\Phi,1)=d(2,1)=5$
$Cost(3,\Phi,1)=d(3,1)=6Cost(3,\Phi,1)=d(3,1)=6$
$Cost(4,\Phi,1)=d(4,1)=8Cost(4,\Phi,1)=d(4,1)=8$

$S = 1$
$Cost(i,s)=min\{Cost(j,s–(j))+d[i,j]\}Cost(i,s)=min\{Cost(j,s)−(j))+d[i,j]\}$
$Cost(2,\{3\},1)=d[2,3]+Cost(3,\Phi,1)=9+6=15cost(2,\{3\},1)=d[2,3]+cost(3,\Phi,1)=9+6=15$
$Cost(2,\{4\},1)=d[2,4]+Cost(4,\Phi,1)=10+8=18cost(2,\{4\},1)=d[2,4]+cost(4,\Phi,1)=10+8=18$
$Cost(3,\{2\},1)=d[3,2]+Cost(2,\Phi,1)=13+5=18cost(3,\{2\},1)=d[3,2]+cost(2,\Phi,1)=13+5=18$
$Cost(3,\{4\},1)=d[3,4]+Cost(4,\Phi,1)=12+8=20cost(3,\{4\},1)=d[3,4]+cost(4,\Phi,1)=12+8=20$
$Cost(4,\{3\},1)=d[4,3]+Cost(3,\Phi,1)=9+6=15cost(4,\{3\},1)=d[4,3]+cost(3,\Phi,1)=9+6=15$
$Cost(4,\{2\},1)=d[4,2]+Cost(2,\Phi,1)=8+5=13cost(4,\{2\},1)=d[4,2]+cost(2,\Phi,1)=8+5=13$

$S = 2\ Cost(2,\{3,4\},1)=$

d[2,3]+Cost(3,{4},1)=9+20=29
d[2,4]+Cost(4,{3},1)=10+15=25=25Cost(2,{3,4},1
)
{d[2,3]+cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25
=25
Cost(3,{2,4},1)=
d[3,2]+Cost(2,{4},1)=13+18=31
d[3,4]+Cost(4,{2},1)=12+13=25=25Cost(3,{2,4},1)
{d[3,2]+cost(2,{4},1)=13+18=31d[3,4]+Cost(4,{2},1)=12+13=25
=25
Cost(4,{2,3},1)=
d[4,2]+Cost(2,{3},1)=8+15=23
d[4,3]+Cost(3,{2},1)=9+18=27=23Cost(4,{2,3},1)
{d[4,2]+cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23

S = 3 Cost(1,{2,3,4},1)=
d[1,2]+Cost(2,{3,4},1)=10+25=35
d[1,3]+Cost(3,{2,4},1)=15+25=40
d[1,4]+Cost(4,{2,3},1)=20+23=43=35cost(1,{2,3,4}),1)
d[1,2]+cost(2,{3,4},1)=10+25=35
d[1,3]+cost(3,{2,4},1)=15+25=40
d[1,4]+cost(4,{2,3},1)=20+23=43=35

The minimum cost path is 35.
Start from cost {1, {2, 3, 4}, 1}, we get the minimum value for d [1, 2]. When s = 3, select the path from 1 to 2 (cost is 10) then go backwards. When s = 2, we get the minimum value for d [4, 2]. Select the path from 2 to 4 (cost is 10) then go backwards.
When s = 1, we get the minimum value for d [4, 3]. Selecting path 4 to 3 (cost is 9), then we shall go to then go to s = Φ step. We get the minimum value for d [3, 1] (cost is 6).
1---2---4---3---1 is path obtained.

ALGORITHM:

1. C ({1}, 1) = 0
2. for s = 2 to n do
3. for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
4. C (S, 1) = ∞
5. for all j Є S and j ≠ 1
6. C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
7. Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

PROGRAM:

```
def g(vertex, rem_v):
    if not rem_v:
        return w[vertex][0]
    elif (vertex, tuple(rem_v)) in memo:
        return memo[(vertex, tuple(rem_v))]
    else:
        mi = float('inf')
        ind = 0
        for i in rem_v:
            temp = rem_v.copy()
```

```
 temp.remove(i)
            val = w[vertex][i] + g(i, temp)
            if val < mi:
                mi = val
                ind = i
        memo[(vertex, tuple(rem_v))] = mi
        index[(vertex, tuple(rem_v))] = ind
        return mi

def path(vertex, rem_v, r):
    if not rem_v:
        r.append(vertex + 1)
        r.append(1)
    else:
        r.append(vertex + 1)
        ind = index[(vertex, tuple(rem_v))]
        rem_v.remove(ind)
        path(ind, rem_v, r)
    return r

v = int(input('Number of objects: '))
w = []

print('Enter slewing time for travel from each object to all other objects:')
for i in range(v):
    l = input('{}->'.format(i + 1))
    l = list(map(int, l.split(',')))
    w.append(l)

memo = {}
index = {}

for i in range(1, len(w)):
    memo[(i, ())] = w[i][0]

print("The min cost: ", g(0, [x for x in range(1, len(w))]))
print("The required path is: ", path(0, [x for x in range(1, len(w))], []))
```

OUTPUT:

Successful test cases:

```
Number of objects: 4
Enter slewing time for travel from each object to all other objects
1->0,10,15,20
2->10,0,35,25
3->15,35,0,30
4->20,25,30,0
The min cost:  80
The required path is:  [1, 2, 4, 3, 1]
```

```
Number of objects: 5
Enter slewing time for travel from each object to all other objects
1->0,15,30,20,25
2->15,0,50,45,30
3->30,50,0,25,45
4->20,45,25,0,35
5->25,30,45,35,0
The min cost:  135
The required path is:  [1, 2, 5, 3, 4, 1]
```

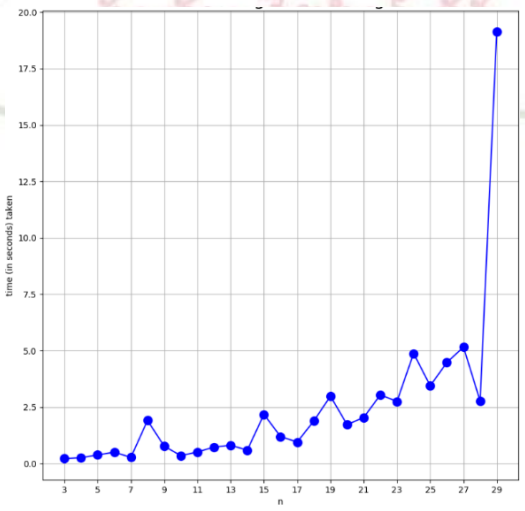
Unsuccessful test cases: When we enter the number of objects as 0 then we are getting a key error.

```
Number of objects: 0
Enter slewing time for travel from each object to all other objects

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-7-fc9b923871c1> in <module>
     43     memo[i]=w[i][0]
     44
---> 45 print("The min cost: ",g(0,[int(x) for x in range(1,len(w))]))
     46 print("The required path is: ",path(0,[int(x) for x in range(1,len(w))],[]))

<ipython-input-7-fc9b923871c1> in g(vertex, rem_v)
      1 def g(vertex,rem_v):
      2     if rem_v==[]:
----> 3         return memo[vertex]
      4     #below elif test, no need
      5     # elif (vertex,tuple(rem_v)) in memo:

KeyError: 0
```

ANALYSIS:

Time Complexity: O(2^n*n^2)

There are at the most 2n.n sub-problems and each one takes linear time to solve. Therefore, the total running time is O(2^n*n^2).



CONCLUSION:

TSP is a popular DP problem, but depending on the size of the input cities, it is possible to find an optimal or a near-optimal solution using various algorithms.

EXPERIMENT NUMBER- 06

AIM: N-Queen is the problem of placing 'N' chess queens on an N×N chessboard. Design a solution for this problem so that no two queens attack each other. Note: A queen can attack when an opponent is on the same row, column or diagonal.

DESCRIPTION:

- Given an n*n chessboard, we need to place n queens in the board such that no two queens are in attacking position i.e they are not in same row, column or diagonal.
- Let us take an example of 4*4 chessboard for instance.
- To solve this problem, we use the backtracking approach.
- Now, we need to place q1, q2, q3 and q4 in the chessboard such that ith queen is placed in ith row.
- We start off by placing q1 in [1,1]
- We need to place q2 in row2 where it is not attacking. Therefore, the next possible solution is [2,3].
- But, when we place q2 in [2,3] there is no safe place for q3 in the chessboard.
- Hence, we backtrack back to q2 and change its position to [2,4]
- Now, q3 can be placed in [3,2] but then q4 will not have a safe place
- Now we backtrack back to q1 and change its position to [1,2]
- Now q2 can be in [2,4]
- q3 in [3,1] and q4 in [4,3] which is the actual solution
- So the solution, x= [2,4,1,3]

ALGORITHM:

- place(k,i)

```
{
        For j=1 to k-1 do
                If ((x[j]==I or (abs(x[j]-i) = abs(j-k)) Then
                        return false
        Return true
}
```

- Nqueens(k,n)

```
{
        For i=1 to n do
                {
If place(k,i) then
{
                                        X[k]=i
                                        If (k==n) then write (x[1:n]) Else
                                        Nqueens(k+1,n)
                                }
                }
```

}

Where,

K is the kth queen to be placed and X[ ] is the solution vector

PROGRAM:

```
def place(row, col):
    for i in range(row):
        if x[i] == col or abs(x[i] - col) == abs(i - row):
            return False
    return True

def print_board(x):
    for i in range(len(x)):
        temp = [0] * len(x)
        temp[x[i]] = i + 1
        print(temp)
    exit(0)

def n_queens(row, n):
    for i in range(n):
        if place(row, i):
            x[row] = i
            if row == n - 1:
                print_board(x)
                break
            else:
                n_queens(row + 1, n)

n = int(input("Enter the number of Queens: "))
x = [-1] * n
print("The possible arrangements of Queens on the chessboard are: ")
n_queens(0, n)
```

TEST CASES:

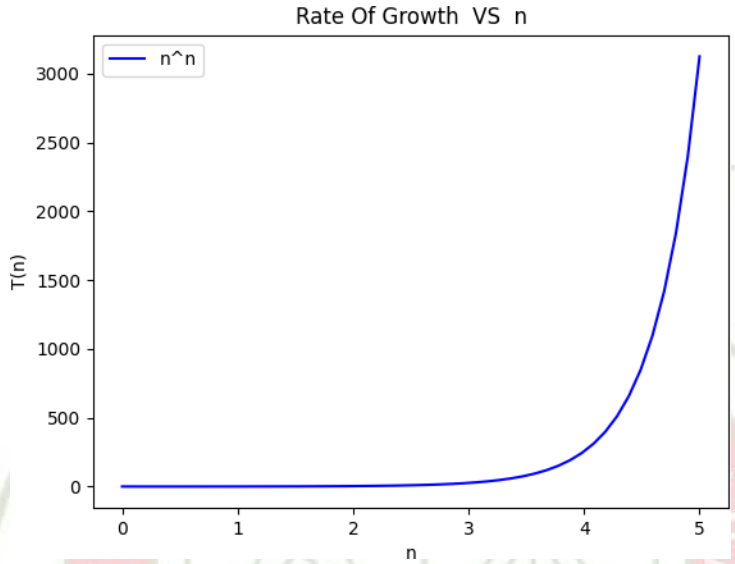
Successful test cases:

```
Enter the number of Queens: 4
The possible arrangement of Queens in chessboard are:
[0, 1, 0, 0]
[0, 0, 0, 2]
[3, 0, 0, 0]
[0, 0, 4, 0]
```

```
Enter the number of Queens: 6
The possible arrangement of Queens in chessboard are:
[0, 1, 0, 0, 0, 0]
[0, 0, 0, 2, 0, 0]
[0, 0, 0, 0, 0, 3]
[4, 0, 0, 0, 0, 0]
[0, 0, 5, 0, 0, 0]
[0, 0, 0, 0, 6, 0]
```

Unsuccessful test cases: When the number of queens are 3, there is no solution for the N- Queen problem as there's no possible way to fit 3 queens in a 3X3 chess board without creating any conflict

ANALYSIS: The time complexity for NQueens is O(n!) i.e., O(n^n) as the first queen has ‘n’ placements, the second queen must not be in the same column as the first as well as at an oblique angle, so the second queen has n-1 possibilities, and so on. Similarly, this algorithm uses O(n^2) positions to represent the 2d chess board and store where the queen is placed.



CONCLUSION: N-Queen is the problem of placing N chess queens on an N*N chessboard so that no two queens attack each other. Using backtracking, we generate all the possible configurations of queens on the board and prints the configuration that satisfies the given constraints.

EXPERIMENT NUMBER- 07

AIM: CSE department of CBIT wants to generate a time table for 'N' subjects. The following information is given – subject name, subject code and list of subjects codes with clashes with this subject. The problem is to identify the list of subjects which can be scheduled on the same time line such that clashes among them do not exist.

DESCRIPTION:
The given problem statement implies graph coloring concept where, the nodes of a graph must be colored with minimum number of possible colors such that no two adjacent nodes have the same color.

- Here we define two functions named as **timescheduling** and **nextvalue**.
- First, we input the total number of subjects, subject names and subject codes.
- We take the clashes as input for each subject and form an adjacency matrix.
- We take a for loop that runs until it finds the minimum number of periods required to schedule the subjects so that clashes do not exist.
- Now we call the function timescheduling with first subject as its argument.
- Timescheduling calls the **nextvalue** function to assign a period and also check if there are any clashes.
- **Nextvalue** return True if it assigned a valid period and there are no clashes otherwise it backtracks to its calling function that is timeacheduling.
- If the period assigned is zero then also it backtracks to its calling function.
- If all the subjects are assigned their periods respectively then it prints the schedule otherwise it calls itself with next node as argument.

ALGORITHM:

- **Algorithm timescheduling(k,i):**

```
While i==True do
        Nextvalue(k)
        If x[k]==0 then
                return If
        k==n-1 then
                Print x
        Else call timescheduling(k+1)
```

- **Algorithm nextvalue(k):**

```
While (True):
        X[k] = (x[k]+1)mod (m+1) If
        x[k]==0 then
                Return For
        j=1 to n do
                If G[k,j]!=0 and x[k]==x[j] then break
        If j==n+1 then return
```

n= no. of subjects
G = adjacency matrix

K= subject to be schedule
m = no. of periods
x = solution vector

PROGRAM:

```
n = int(input("Enter the number
   of subjects: "))
sub = []

for i in range(n):
   print("Enter subject_code and
   subject_name for {}".format(i
   + 1))
   l = list(map(str,
   input().split()))
   sub.append(l)

print(sub)

x = [0 for _ in range(n)]
G = [[0 for _ in range(n)] for _ in
   range(n)]

for i in range(n):
   print("Enter clashes for
   subject {}".format(i + 1))
   l = list(map(int,
   input().split()))

   for j in l:
      G[i][j - 1] = 1
print(G)

def time_scheduling(k, i):
   while i:
      next_value(k)
      if x[k] == 0:
         return
      if k == n - 1:
         print("Time line schedule
   sequence is:", *x)
```

```
exit(0)
        i = time_scheduling(k + 1,
    i)


def next_value(k):
    while True:
        x[k] = (x[k] + 1) % (max(x)
    + 1)
        if x[k] == 0:
            return
        for j in range(n):
            if G[k][j] != 0 and x[k]
    == x[j]:
                break
        if j == n - 1:
            return


time_scheduling(0, True)
```

TEST CASES:

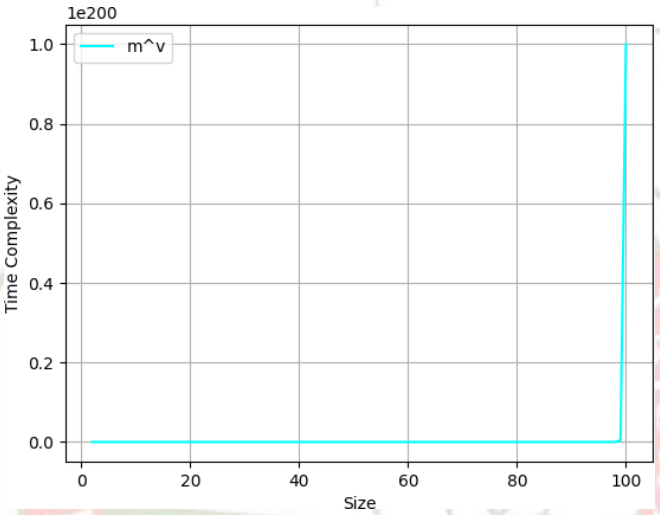
Successful test cases:

```
no of subjects: 4
Enter subject_code and subject_name for 1
121 eng
Enter subject_code and subject_name for 2
122 chem
Enter subject_code and subject_name for 3
123 dm
Enter subject_code and subject_name for 4
124 bee
[['121', 'eng'], ['122', 'chem'], ['123', 'dm'], ['124', 'bee']]
Enter clashes for subject 1
2 3
Enter clashes for subject 2
1 4
Enter clashes for subject 3
1 4
Enter clashes for subject 4
2 3
[[0, 1, 1, 0], [1, 0, 0, 1], [1, 0, 0, 1], [0, 1, 1, 0]]
Time line schedule seqquence is :  1 2 2 1
```

```
no of subjects: 6
Enter subject_code and subject_name for 1
154 camp
Enter subject_code and subject_name for 2
155 daa
Enter subject_code and subject_name for 3
156 mfds
Enter subject_code and subject_name for 4
157 dbms
Enter subject_code and subject_name for 5
158 iwt
Enter subject_code and subject_name for 6
159 eea
[['154', 'camp'], ['155', 'daa'], ['156', 'mfds'], ['157', 'dbms'], ['158', 'iwt'], ['159', 'eea']]
Enter clashes for subject 1
1 2
Enter clashes for subject 2
1 3
Enter clashes for subject 3
2 5
Enter clashes for subject 4
4 5 6
Enter clashes for subject 5
4 6
Enter clashes for subject 6
5 2
[[0, 0, 0, 1, 1, 0], [1, 0, 0, 1, 0, 1], [0, 1, 0, 1, 0, 0], [1, 0, 1, 0, 1, 0], [0, 0, 1, 1, 0, 0], [1, 0, 0, 0, 0, 1]]
Time line schedule seqquence is :  1 2 1 2 3 2
```

Unsuccessful test cases: When a graph with self-loop is introduced, the code with keep on running as it keeps on searching for the exact color to add without collision but due to self- loop, there is no possibility

ANALYSIS: Time complexity for this is algorithm is O(m^v) where m is the chromatic number of the given graph and v is the number of vertices. And the space complexity is O(v^2) for the adjacency matrix of the graph.

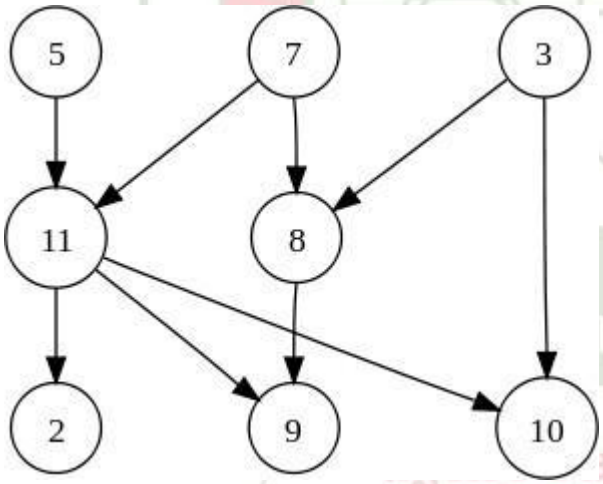


CONCLUSION: The above problem of graph coloring was based on backtracking where the functions backtrack when there is a bounding function to obtain the appropriate value. Through this we can find the minimum number of colors required to color a graph also called chromatic number.

EXPERIMENT NUMBER- 08:

AIM: Consider a source code structure where you are building several libraries DLLs (Dynamic-Link Library) and they have dependencies on each other. For example, to build DLL , you must have built DLLs B, C and D (Maybe you have a reference of B,C and D in the project that builds A).

DESCRIPTION: In computer science, a topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. Precisely, a topological sort is a graph traversal in which each node v is visited only after all its dependencies are visited. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering,
and algorithms are known for constructing a topological ordering of any DAG in linear time. Topological sorting has many applications especially in ranking problems such as feedback arc set. Topological sorting is possible even when the DAG has disconnected components.



The graph shown has many valid topological sorts, including:

- 5, 7, 3, 11, 8, 2, 9, 10 (visual top-to-bottom, left-to-right)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

ALGORITHM: Topological sorting can be done by using both stack and queue. By using stack

1) Create a empty stack.
2) Call DFS on the graph whose indegree is 0.
3) Push elements to stack if all the adjacent nodes are visited.
4) Pop all elemets from stack we got topological order.

By using Queue
Here we use 2 arrays and one queue
1) Add all the vertices with no dependencies into a queue(An array stores indegree of all vertices)
2) Then delete that node from the graph and add into the soritng array and change the indegree of other vertices accordingly.

PROGRAM:

```
class Graph:
    def __init__(self, V, adj):
        self.V = V
        self.Matrix = adj


    def topological_sort(self):
        indegree = [0] * self.V
        for x in range(self.V):
            for y in range(self.V):
                indegree[y] += self.Matrix[x][y]

        queue = []
        order = []
        visited = []

        for x in range(self.V):
            if indegree[x] == 0:

queue.append(x)

visited.append(x)
```

```
    while queue:
        u = queue.pop(0)
        order.append(u)

        for x in
range(self.V):
            indegree[x] -=
self.Matrix[u][x]

self.Matrix[u][x] = 0

        for x in
range(self.V):
            if indegree[x]
== 0 and x not in visited:

queue.append(x)

visited.append(x)

      print("Topological
order is:", order)

# Example usage
n = int(input("Enter the
number of vertices: "))
adj = [[0] * n for _ in
range(n)]

for i in range(n):
   print("Enter the
dependencies of %d DLL
with other DLLs (if no
dependency, click enter)"
% (i))
   l = [int(x) for x in
input().split()]

   for j in l:
      adj[j][i] = 1

G = Graph(n, adj)
print(G.Matrix)
```

G.topological_sort()

OUTPUT:

Successful test cases:

```
Enter no of vertices: 6
Enter the dependencies of 0 DLL with other DLLs (if no dependency(no indegree), click enter)
4 5
Enter the dependencies of 1 DLL with other DLLs (if no dependency(no indegree), click enter)
3 4
Enter the dependencies of 2 DLL with other DLLs (if no dependency(no indegree), click enter)
5
Enter the dependencies of 3 DLL with other DLLs (if no dependency(no indegree), click enter)
2
Enter the dependencies of 4 DLL with other DLLs (if no dependency(no indegree), click enter)

Enter the dependencies of 5 DLL with other DLLs (if no dependency(no indegree), click enter)

[[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0], [0, 1, 0, 0, 0, 0], [1, 1, 0, 0, 0, 0], [1, 0, 1, 0, 0, 0]]
Topological order is: [4, 5, 0, 2, 3, 1]
```

```
Enter no of vertices: 5
Enter the dependencies of 0 DLL with other DLLs (if no dependency(no indegree), click enter)

Enter the dependencies of 1 DLL with other DLLs (if no dependency(no indegree), click enter)

Enter the dependencies of 2 DLL with other DLLs (if no dependency(no indegree), click enter)

Enter the dependencies of 3 DLL with other DLLs (if no dependency(no indegree), click enter)

Enter the dependencies of 4 DLL with other DLLs (if no dependency(no indegree), click enter)

[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
Topological order is: [0, 1, 2, 3, 4]
```

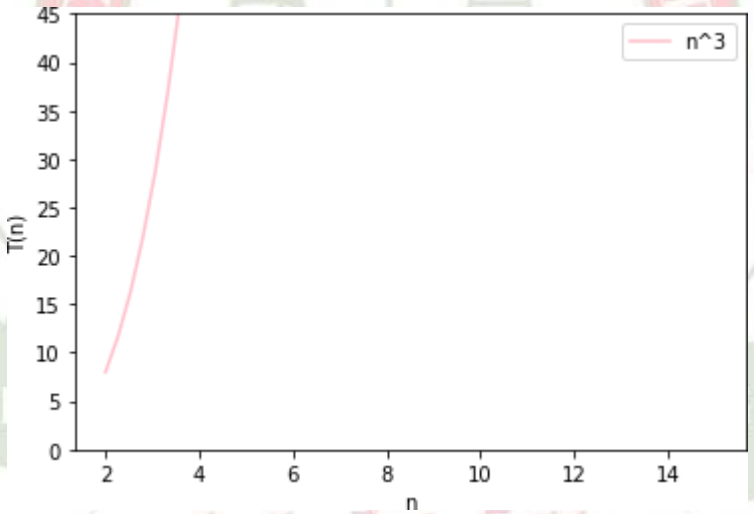
Unsuccessful test cases: Topological sorting is not possible if the graph is not DAG.
Here the graph which is given as input is not DAG, so the topological order which we got contains only 2 vertices.

The below graph has self-loops so the topological sorting is not possible for that kind of graphs.

```
Enter no of vertices: 5
Enter the dependencies of 0 DLL with other DLLs (if no dependency(no indegree), click enter)
0 3
Enter the dependencies of 1 DLL with other DLLs (if no dependency(no indegree), click enter)
1 4
Enter the dependencies of 2 DLL with other DLLs (if no dependency(no indegree), click enter)

Enter the dependencies of 3 DLL with other DLLs (if no dependency(no indegree), click enter)

Enter the dependencies of 4 DLL with other DLLs (if no dependency(no indegree), click enter)

[[1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 0, 0, 0], [1, 0, 0, 0, 0], [0, 1, 0, 0, 0]]
Topological order is: [2, 3, 4]
```

ANALYSIS:

Time Complexity: O(V+E)
The time complexity of topological sort is O(V+E), where V = Vertices, E = Edges.
The queue needs to store all the vertices of the graph. So the space required is O(V)
No. edges in a DAG is n(n-1)^2 where n is number of vertices.
So the time complexity is O(n^3)



CONCLUSION:
Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. A graph that contains a cycle (cyclic graph) cannot have valid sorting. All rooted trees have topological ordering since there are no cycles. Topological orderings are not unique for a particular graph.

EXPERIMENT NUMBER- 09

AIM: Bi-connected graphs are used in the design of power grid networks. Consider the nodes as cities and the edges as electrical connections between them, you would like the network to be robust and a failure at one city should not result in a loss of power in other cities.

DESCRIPTION:

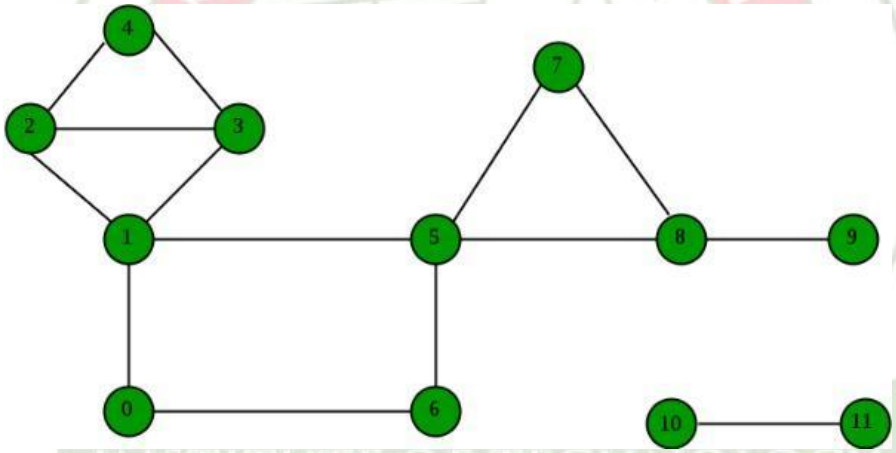
In graph theory, a biconnected component (sometimes known as a 2-connected component) is a maximal biconnected subgraph. Any connected graph decomposes into a tree of biconnected components called the block-cut tree of the graph. The blocks are attached to each other at shared vertices called cut vertices or separating vertices or articulation points.

Specifically, a cut vertex is any vertex whose removal increases the number of connected components.

A graph is said to be Biconnected if:

It is connected, i.e. it is possible to reach every vertex from every other vertex, by a simple path.

Even after removing any vertex the graph remains connected.



In above graph, following are the biconnected components:

- 4–2 3–4 3–1 2–3 1–2
- 8–9
- 8–5 7–8 5–7
- 6–0 5–6 1–5 0–1
- 10–11

ALGORITHM:

```
1. DFS_Visit(v):
2.          {
            color(v)=GREY;time=time+1;d[v]=time;
            } low[v]=d[v]
            for each w in Adj[v]{
                  if(color[w]==WHITE){
```

```
                                prev[w]=u;
                                DFS_Visit(w); if
                                low[w]>=d[v]
                                        record that vertex v is in articulation
                                if(low[w]<low[v])
                                        low[v]=low[w]
                        }
                        else if w is not the parent of v then
                                //--(v,w) is a BACK edge
                                If(d[w]<low[v])
                                        low[v]=d[w] ;
                }
                color[v]=BLACK;time=time+1;f[v]=time;
        3.  }
```

PROGRAM:

```
class Graph:
    def __init__(self, vertices, graph):
        # No. of vertices
        self.V = vertices
        self.Time = 0
        self.count = 0
        self.time_ap = 0
        self.graph = graph

    def AP_Visit(self, u, visited, ap, parent, low, disc):
        children = 0
        visited[u] = True
        disc[u] = self.time_ap
        low[u] = self.time_ap
        self.time_ap += 1

        for v in self.graph[u]:
            if visited[v] == False:
                parent[v] = u
                children += 1
                self.AP_Visit(v, visited, ap, parent, low, disc)
                low[u] = min(low[u], low[v])

                if parent[u] == -1 and children > 1:
                    ap[u] = True
                if parent[u] != -1 and low[v] >= disc[u]:
                    ap[u] = True
            elif v != parent[u]:
                low[u] = min(low[u], disc[v])

    def Articulation_Point(self):
        visited = [False] * self.V
        d_ap = [-1] * self.V
        low_ap = [-1] * self.V
        parent_ap = [-1] * self.V
```

```
        ap = [False] * self.V

        for i in range(self.V):
            if visited[i] == False:
                self.AP_Visit(i, visited, ap, parent_ap, low_ap, d_ap)

        a = ""
        for i in range(self.V):
            if ap[i] == True:
                a += str(i) + " "
        print("The articulation points are: " + a)

graph = {}
n = int(input("Enter the number of cities: "))
for i in range(n):
    key = int(input("Enter the key: "))
    value = list(map(int, input("Enter the value: ").split(",")))
    graph[key] = value

print(graph)
g = Graph(n, graph)
g.Articulation_Point()
```

OUTPUT:

Successful test cases:

```
Enter the number of cities: 7
Enter the key: 0
Enter the value: 1,2
Enter the key: 1
Enter the value: 0,3
Enter the key: 2
Enter the value: 0,3
Enter the key: 3
Enter the value: 1,2,5
Enter the key: 4
Enter the value: 3,6
Enter the key: 5
Enter the value: 3,6
Enter the key: 6
Enter the value: 4,5
{0: [1, 2], 1: [0, 3], 2: [0, 3], 3: [1, 2, 5], 4: [3, 6], 5: [3, 6], 6: [4, 5]}
The articulation points are: 3
```

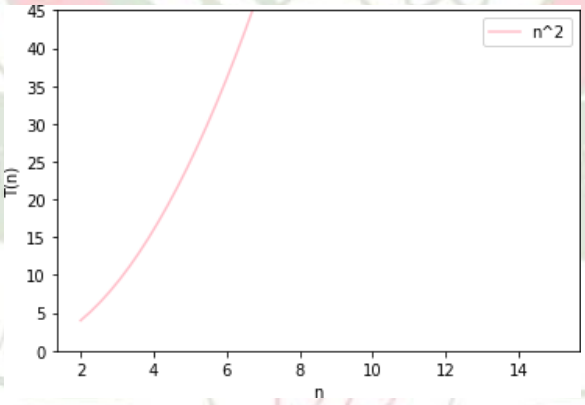
```
Enter the number of cities: 14
Enter the key: 0
Enter the value: 0,1
Enter the key: 1
Enter the value: 1,2
Enter the key: 2
Enter the value: 1,3
Enter the key: 3
Enter the value: 2,3
Enter the key: 4
Enter the value: 2,4
Enter the key: 5
Enter the value: 3,4
Enter the key: 6
Enter the value: 1,5
Enter the key: 7
Enter the value: 0,6
Enter the key: 8
Enter the value: 5,6
Enter the key: 9
Enter the value: 5,7
Enter the key: 10
Enter the value: 5,8
Enter the key: 11
Enter the value: 7,8
Enter the key: 12
Enter the value: 8,9
Enter the key: 13
Enter the value: 10,11
{0: [0, 1], 1: [1, 2], 2: [1, 3], 3: [2, 3], 4: [2, 4], 5: [3, 4], 6: [1, 5], 7: [0, 6], 8: [5, 6], 9: [5, 7], 10: [5, 8], 11:
[7, 8], 12: [8, 9], 13: [10, 11]}
The articulation points are: 1 2
```

ANALYSIS:

Time Complexity: O(V+E)

The time complexity of topological sort is O(V+E), where V = Vertices, E = Edges. No. edges in a graph is n(n-1) where n is number of vertices.

So the time complexity is O(n^2)



CONCLUSION:

A graph is biconnected if it has no arculation points, so there will be one biconnected component which is the graph itself

EXPERIMENT NUMBER- 10

AIM: You are given the task of choosing the optimal path to connect 'N' devices. The devices are connected with the minimum required N-1 wires into a tree structure, and each device is connected with the other with a wire of length 'L' ie 'D1'connected to 'D2' with a wire of length 'L1'. This information will be available for all 'N' devices.

a) Determine the minimum length of the wire which consists of N-1 wires that will connect all devices.

b) Determine the minimum length of the wire which connects Di and Dj

c) Determine the minimum length of the wire which connects Di to all other devices.

DESCRIPTION:

(a)**:** Minimum Spanning Tree:

A minimum spanning tree is the one that contains the least weight among all the other spanning trees of a connected weighted graph. There can be more than one minimum spanning tree for a graph.

There are two most popular algorithms that are used to find the minimum spanning tree in a graph. They include: Kruskal's algorithm, Prim's algorithm.

Kruskal's algorithm is an algorithm to find the MST in a connected graph. Kruskal's algorithm finds a subset of a graph G such that:

- It forms a tree with every vertex in it.
- The sum of the weights is the minimum among all the spanning trees that can be formed from this graph.

The sequence of steps for Kruskal's algorithm is given as follows:

- First sort all the edges from the lowest weight to highest.
- Take edge with the lowest weight and add it to the spanning tree. If the cycle is created, discard the edge.
- Keep adding edges like in step 1 until all the vertices are considered.

Prim's algorithm is yet another algorithm to find the minimum spanning the tree of a graph. In contrast to Kruskal's algorithm that starts with graph edges, Prim's algorithm starts with a vertex. We start with one vertex and keep on adding edges with the least weight till all the vertices are covered.

The sequence of steps for Prim's Algorithm is as follows:

- Choose a random vertex as starting vertex and initialize a minimum spanning tree.
- Find the edges that connect to other vertices. Find the edge with minimum weight and add it to the spanning tree.
- Repeat step 2 until the spanning tree is obtained.

(b)**:** Floyd Warshall Algorithm:

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

(c). Bellman/Dijkstras:

Bellman Ford's algorithm

Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edge in any simple path, that is why the outer loop runs |v| – 1 time. The idea is, assuming that there is no negative weight cycle if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give the shortest path with at-most (i+1) edges

Dijkstra's algorithm

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate an SPT (shortest path tree) with a given source as root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

ALGORITHM:

```
Procedure kruskal(G)
    G – input graph    :
begin
A = Ø
For each vertex v ∈ G.V:
   MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by weight in increasing order (u, v):
   if FIND-SET(u) ≠ FIND-SET(v):
   A = A U {(u, v)}
   UNION(u, v)
return A
end kruskal;
```

```
Procedure prims
        G – input graph
        U – random vertex
        V – vertices in graph G
begin
        T = Ø;
        U = { 1 };
        while (U ≠ V)
          let (u, v) be the least cost edge such that u ∈ U and v ∈ V - U;
          T = T U {(u, v)}
          U = U U {v}
end procedure
```

Floyd-Warshall algorithm

```
0   Algorithm AllPaths(cost, A, n)
1   // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices; A[i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i := 1 to n do
6           for j := 1 to n do
7               A[i, j] := cost[i, j]; // Copy cost into A.
8       for k := 1 to n do
9           for i := 1 to n do
10              for j := 1 to n do
11                  A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12  }
```

Dijkstra's Algorithm

```
1   Algorithm ShortestPaths(v, cost, dist, n)
2   // dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
3   // path from vertex v to vertex j in a digraph G with n
4   // vertices. dist[v] is set to zero. G is represented by its
5   // cost adjacency matrix cost[1 : n, 1 : n].
6   {
7       for i := 1 to n do
8       { // Initialize S.
9           S[i] := false; dist[i] := cost[v, i];
10      }
11      S[v] := true; dist[v] := 0.0; // Put v in S.
12      for num := 2 to n − 1 do
13      {
14          // Determine n − 1 paths from v.
15          Choose u from among those vertices not
16          in S such that dist[u] is minimum;
17          S[u] := true; // Put u in S.
18          for (each w adjacent to u with S[w] = false) do
19              // Update distances.
20              if (dist[w] > dist[u] + cost[u, w])) then
21                  dist[w] := dist[u] + cost[u, w];
22      }
23  }
```

Greedy algorithm to generate shortest paths

```
1   Algorithm BellmanFord(v, cost, dist, n)
2   // Single-source/all-destinations shortest
3   // paths with negative edge costs
4   {
5       for i := 1 to n do // Initialize dist.
6           dist[i] := cost[v, i];
7       for k := 2 to n − 1 do
8           for each u such that u ≠ v and u has
9                   at least one incoming edge do
10              for each ⟨i, u⟩ in the graph do
11                  if dist[u] > dist[i] + cost[i, u] then
12                      dist[u] := dist[i] + cost[i, u];
13  }
```

Bellman and Ford algorithm to compute shortest paths

PROGRAM (a):

#Must be solved using Prims/Kruskal Algorithm
A Python program for Prim's Minimum Spanning Tree (MST) algorithm.
The program is for adjacency matrix representation of the graph

Using Prims

```
n = int(input("Enter the number of Vertices: ")) matrix
= [ [0, 2, float('inf'), 6, float('inf')],
        [2, 0, 3, 8, 5],
        [float('inf'), 3, 0, float('inf'), 7],
        [6, 8, float('inf'), 0, 9],
        [float('inf'), 5, 7, 9, 0]]

class Graph():
    def _init_(self,vertices,graph):
        self.V=vertices self.graph=graph

    def printMST(self, parent):
        print("Edge\tWeight") total=0
        for i in range(1,self.V):
            print(parent[i], "-", i, "\t", self.graph[i][parent[i]])
            total+=self.graph[i][parent[i]]
        print("Min length connecting all devices: ",total)

    def minKey(self,key,mstSet):
        min=999999
        for v in range(self.V):
            if key[v]<min and mstSet[v] == False: min
                = key[v]
                min_index = v
        return min_index
    def primMST(self):
        key = [999999]*self.V
        parent = [None]*self.V
        key[0]=0
        mstSet = [False]*self.V
        parent[0] = -1
        for cout in range(self.V):
            u = self.minKey(key,mstSet)
            mstSet[u]=True
            for v in range(self.V):
                if self.graph[u][v]>0 and mstSet[v] == False and key[v]>self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u
        self.printMST(parent)
g=Graph(n,matrix)
g.primMST()
```

TEST CASES (a):

```
Enter the number of Vertices: 5
Edge    Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
Min length connecting all devices:  16
```

```
Enter the number of Vertices: 2
Edge    Weight
0 - 1    2
Min length connecting all devices:  2
```

PROGRAM (b):

```
# Must be solved using Floyd Warshal to retrive Di, Dj from the adj matrix a=[]
for i in matrix:
    a.append(i)
for k in range(n): for i
    in range(n):
        for j in range(n):
            a[i][j]=min(a[i][j],a[i][k]+a[k][j])
print("Min distance between two devices using Floyd Warshal: ") print(a)
```

TEST CASES (b):

```
Enter the number of Vertices: 5
Edge    Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
Min length connecting all devices:  16
```

```
Enter the number of Vertices: 2
Edge    Weight
0 - 1    2
Min length connecting all devices:  2
```

PROGRAM (c):

```
# Must be solved using Bellman/Dijkstras #
Dijkstra's Algorithm in Python
# Providing the graph

# Find which vertex is to be visited next def
to_be_visited():
    global visited_and_distance v =
    -10
    for index in range(n):
        if visited_and_distance[index][0] == 0 \
            and (v < 0 or visited_and_distance[index][1] <=
                visited_and_distance[v][1]):
            v = index return
    v
```

```
# n = len(vertices[0])

visited_and_distance = [[0, 0]] for i
in range(n-1):
    visited_and_distance.append([0, 999999]) for
vertex in range(n):
    # Find next vertex to be visited
    to_visit = to_be_visited()
    for neighbor_index in range(n):

        # Updating new distances
        if matrix[to_visit][neighbor_index] > 0 and \
                visited_and_distance[neighbor_index][0] == 0:
            new_distance = visited_and_distance[to_visit][1] \
                + matrix[to_visit][neighbor_index]
            if visited_and_distance[neighbor_index][1] > new_distance:
                visited_and_distance[neighbor_index][1] = new_distance

        visited_and_distance[to_visit][0] = 1

i = 0
print("Dijkstra Algo output: ") #
Printing the distance
for distance in visited_and_distance:
    print("Distance of ", i,
          " from source vertex: ", distance[1]) i = i
    + 1
```

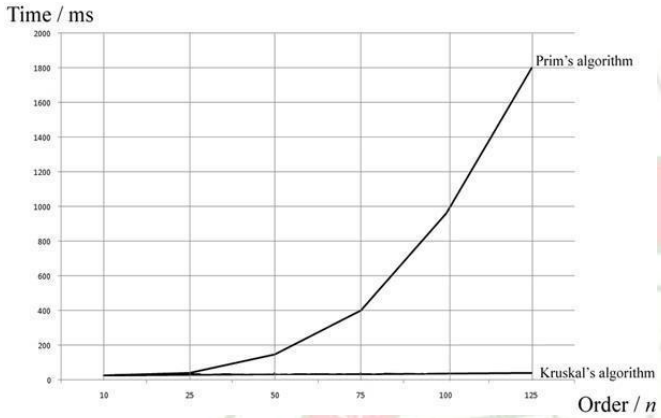
TEST CASES (c):

```
Enter the number of Vertices: 5
Edge    Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
Min length connecting all devices:  16
```

```
Enter the number of Vertices: 2
Edge    Weight
0 - 1    2
Min length connecting all devices:  2
```

ANALYSIS:
The time complexity of the Prim's Algorithm is O((V+E)logV) because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

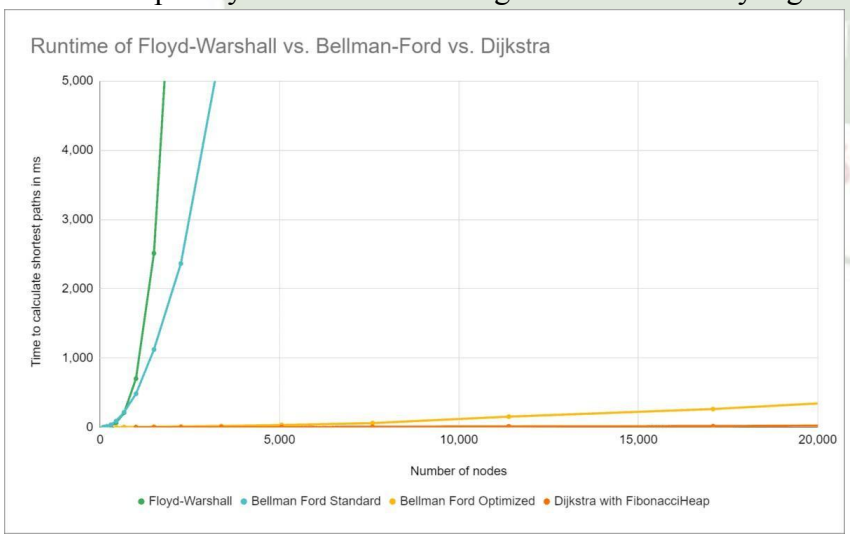
The time complexity of the Kruskal's algorithm is O(E log E) or O(E log V), where E is a number of edges and V is a number of vertices.



The Floyd-Warshall algorithm is a graph-analysis algorithm that calculates shortest paths between all pairs of nodes in a graph. It is a dynamic programming algorithm with $O(|V|^3)$ time complexity and $O(|V|^2)$ space complexity.

Dijkstra Algorithm has a time complexity of O(V^2) O(V2) using the adjacency matrix representation of graph. The time complexity can be reduced to O((V+E)logV)O((V+E)logV) using adjacency list representation of the graph, where E is the number of edges in the graph and V is the number of vertices in the graph.

Time Complexity of Bellman Ford algorithm is relatively high $O(V \cdot E)$, in case E=V2, O(V3).



CONCLUSION: We generalized deterministic shortest path problems in different ways like Bellman-ford, Dijkstra's, Floyd-Warshall and Johnson's Algorithms and also minimum spanning trees. Deterministic shortest path problems assume that the successor state is completely determined by the current state and the action executed in it.