# *Title: Toy problems - Camel-Banana Problem*

**Ex. No.: 1** **Reg. No.:**

**Date:** **Name:**

**Aim:** To write a program to implement the camel-banana problem.

**Procedure/Algorithm:**

- 3000 Bananas at Source => 2000 at First Intermediate => 1000 at Second Intermediate
  3000 x km 2000 y km 1000 z km
- Source to Int. Point1, Camel has to take 5 trips => 3 forward, 2 backward (3000 bananas)
- Int. Point1 to Int. Point2, Camel has to take 3 trips => 2 forward, 1 backward (2000 bananas)
- From Int. Point2 to Destination, only one forward trip

**Program:**

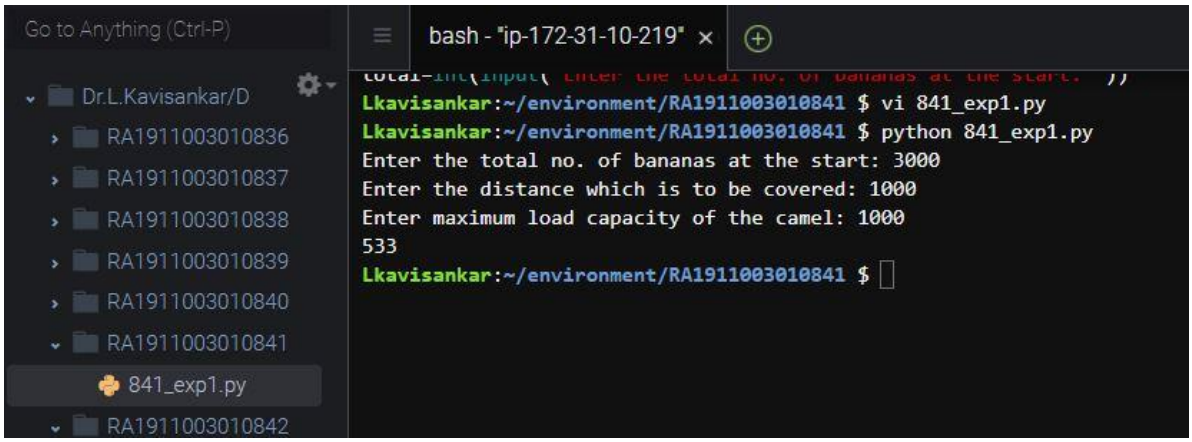
```
total=int(input('Enter the total no. of bananas at the start: '))
distance=int(input('Enter the distance which is to be covered: '))
load_capacity=int(input('Enter maximum load capacity of the camel: '))
lost_bananas=0
bananas=total
for i in range(distance):
   while bananas>0:
      bananas=bananas-load_capacity
      if bananas==1:
         lost_bananas=lost_bananas-1
      lost_bananas=lost_bananas+2
   lost_bananas=lost_bananas-1
   bananas=total-lost_bananas
   if bananas==0:
      break
print(bananas)
```

**Manual Output:**

total = 3000, distance=1000, load_capacity=1000

1. 3000 - 5x = 2000 => x = 200
2. 2000 - 3y = 1000 => y = 333
3. 1000 - x - y = z => 1000 - 200 - 333 = z => z = 467
4. Remaining Bananas => 1000 - 467 = 533

**Screenshot of output:**




**Result:** Thus, the Camel-Banana Problem was implemented successfully.

# *Title:* Real WordProblem (*Graph coloring*)

**Ex. No.: 2** **Reg. No.:**

**Date:** **Name:**

**Aim:** To implement the Graph Coloring Problem.

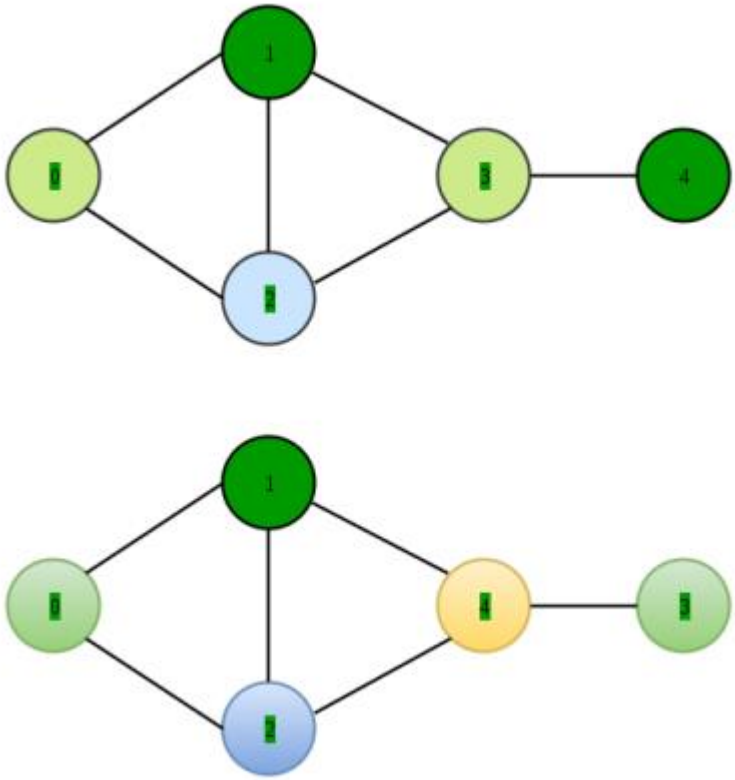
**Procedure/Algorithm:**

1. Color first vertex with first color
2. For the remaining V-1 vertices:
    a. Considering the currently picked vertex and coloring it with lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v, assign a new color to it.

**Program:**

```
def addEdge(adj, v, w):

        adj[v].append(w)
        adj[w].append(v)
        return adj

def greedyColoring(adj, V):

        result = [-1] * V
        result[0] = 0;
        available = [False] * V
        for u in range(1, V):
                for i in adj[u]:
                        if (result[i] != -1):
                                available[result[i]] = True
                cr = 0
                while cr < V:
                        if (available[cr] == False):
                                break

                        cr += 1
                result[u] = cr
                for i in adj[u]:
                        if (result[i] != -1):
                                available[result[i]] = False
        for u in range(V):
                print("Vertex", u, " ---> Color", result[u])

if __name__ == '__main__':

        g1 = [[] for i in range(5)]
        g1 = addEdge(g1, 0, 1)
        g1 = addEdge(g1, 0, 2)
        g1 = addEdge(g1, 1, 2)
        g1 = addEdge(g1, 1, 3)
        g1 = addEdge(g1, 2, 3)
        g1 = addEdge(g1, 3, 4)
        print("Coloring of graph 1 ")
        greedyColoring(g1, 5)

        g2 = [[] for i in range(5)]
        g2 = addEdge(g2, 0, 1)
        g2 = addEdge(g2, 0, 2)
        g2 = addEdge(g2, 1, 2)
        g2 = addEdge(g2, 1, 4)
        g2 = addEdge(g2, 2, 4)
        g2 = addEdge(g2, 4, 3)
        print("\nColoring of graph 2")
        greedyColoring(g2, 5)
```

**Manual Output:**



Considering these 2 graphs, vertices 3 and 4 are swapped. If we consider 0,1,2,3,4 in first graph, the graph can be colored using 3 colors. But considering the vertices 0,1,2,3,4 in the second graph, it can be colored using 4 colors.

**Screenshot of output:**

```
RA1911003010845:~/environment/RA1911003010841 $ vi 841_exp2_gc.py
RA1911003010845:~/environment/RA1911003010841 $ python 841_exp2_gc.py
Coloring of graph 1
('Vertex', 0, ' ---> Color', 0)
('Vertex', 1, ' ---> Color', 1)
('Vertex', 2, ' ---> Color', 2)
('Vertex', 3, ' ---> Color', 0)
('Vertex', 4, ' ---> Color', 1)

Coloring of graph 2
('Vertex', 0, ' ---> Color', 0)
('Vertex', 1, ' ---> Color', 1)
('Vertex', 2, ' ---> Color', 2)
('Vertex', 3, ' ---> Color', 0)
('Vertex', 4, ' ---> Color', 3)
RA1911003010845:~/environment/RA1911003010841 $
```

**Result:**

Thus, Graph coloring problem was implemented successfully.

# *Title:* constraint satisfaction problems

**Ex. No.: 3** **Reg. No.:**

**Date:** **Name:**

**Aim:** To implement the BASE+BALL+GAMES cryptarithm.

**Procedure/Algorithm:**
Systematically substitute digits for letters of the puzzle to form valid calculation.

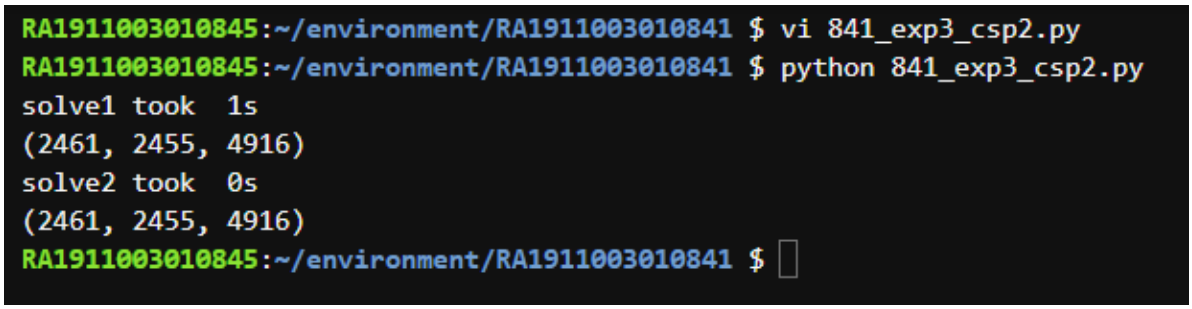
**Program:**

```
import time
import itertools


def timeit(fn):
   def wrapper():
      start = time.clock()
      ret = fn()
      elapsed = time.clock() - start
      print("%s took %2.fs" % (fn.__name__, elapsed))
      return ret
   return wrapper

@timeit
def solve1():
   for b in xrange(1, 10):
      for a in xrange(0, 10):
         for s in xrange(0, 10):
            for e in xrange(0, 10):
               for l in xrange(1, 10):
                  for g in xrange(0, 10):
                     for m in xrange(0, 10):
                        if distinct(b,a,s,e,l,g,m):
                           base = 1000 * b + 100 * a + 10 * s + e
                           ball = 1000 * b + 100 * a + 10 * l + l
                           games = 10000 * g + 1000 * a + 100 * m + 10 * e + s
                           if base +ball == games:
                              return base,ball,games


def distinct(*args):
   return len(set(args)) == len(args)

@timeit
def solve2():
   letters = ('b','a','s','e','l','g','m')
   digits = range(10)
   for perm in itertools.permutations(digits, len(letters)):
      sol = dict(zip(letters, perm))
      if sol['b'] == 0:
         continue
      base  = 1000 * sol['b'] + 100 * sol['a'] + 10 * sol['s'] + sol['e']
      ball  = 1000 * sol['b'] + 100 * sol['a'] + 10 * sol['l'] + sol['l']
      games  = 10000 * sol['g'] + 1000 * sol['a'] + 100 * sol['m'] + 10 * sol['e'] + sol['s']
      if base + ball  == games:
         return base,ball,games


print(solve1())
print(solve2())
```

**Manual Output:**
B=2, A=4, S=6, E=1, L=5, G=0, M=9

```
 2461
+
 2455
 _____
 4916
```

**Screenshot of output:**




**Result:**

Thus, BASE+BALL=GAMES cryptarithmic problem was implemented

successfully.

# *Title: Breadth First Search and Depth First Search*

**Ex. No.: 5** **Reg. No.:**
**Date:** **Name:**

**Aim:** To write a program to implement the Breadth First Search and Depth First Search algorithms.

**Description:** Web scraping is extensively being used in many industrial applications today. Be it in the field of natural language understanding or data analytics, scraping data from websites is one of the main aspects of many such applications. Scraping of data from websites is extracting large amounts of contextual texts from a set of websites for different uses.

**Features:**

1. Given an input URL and a depth upto which the crawler needs to crawl, we will extract all the URLs and categorize them into internal and external URLs.
2. Internal URLs are those which has the same domain name as that of the input URL. External URLs are those which has different domain name as that of the given input URL.
3. We check the validity of the extracted URLs. If the URL has a valid structure, only then it is considered.
4. A depth of 0 means that only the input URL is printed. A depth of 1 means that all the URLs inside the input URL is printed and so on.

**Procedure/Algorithm:**

1. First, we import the installed libraries.
2. Then, we create two empty sets called internal_links and external_links which will store internal and external links separately and ensure that they do not contain duplicates.
3. We then create a method called level_crawler which takes an input URL and crawls it and displays all the internal and external links using the following steps –
   • Define a set called url to temporarily store the URLs.
   • Extract the domain name of the url using urlparse library.
   • Create a beautifulsoup object using HTML parser.
   • Extract all the anchor tags from the beautifulsoup object.
   • Get the href tags from the anchor tags and if they are empty, don't include them.
   • Using urljoin method, create the absolute URL.
   • Check for the validity of the URL.
   • If the url is valid and the domain of the url is not in the href tag and is not in external links set, include it into external links set.
   • Else, add it into internal links set if it is not there and print and put it in temporary url set.
   • Return the temporary url set which includes the visited internal links. This set will be used later on.
4. If the depth is 0, we print the url as it is. If the depth is 1, we call the level_crawler method defined above.
5. Else, we perform a breadth first search (BFS) traversal considered the formation of a URL page as tree structure. At the first level we have the input URL. At the next level, we have all the URLs inside the input URL and so on.
6. We create a queue and append the input url into it. We then pop an url and insert all the urls inside it into the queue. We do this until all the urls at a particular level is not parsed. We repeat the process for the number of times same as the input depth.

**Program:**
BFS:

```
import requests

from bs4 import BeautifulSoup

import re

import time



# Goal: Write web crawling


# the below functions deletes duplicates and adds to the master list.

def deleteDuplicates(uniqueLinks,tempLinks,list_crawled,list_master):

 for a in tempLinks:

  if a not in uniqueLinks:

   if len(a) > 1:

    if a not in list_crawled:

     uniqueLinks.append(a)


# Add unique links to the masterlist

 for a in uniqueLinks:

  if a not in list_master:

   list_master.append(a)


 return uniqueLinks



# the below function crawls the given nextUrl

# Removes references and external links from the fetched Urls
```

```
# Also returns the master list only with the Urls containing 'Rain' and also
Urls with the anchor tags
# containing texts with the string 'rain'
def hitUrl(nextUrl,list_crawled,list_master):

 list_crawled.append(nextUrl)

 uniqueLinks = []
 tempLinks = []

 time.sleep(1)

 data = requests.get(nextUrl)

 data_text = data.text

 soup = BeautifulSoup(data_text,'html.parser')

 dataInFocus = soup.find('div',{'id': 'mw-content-text'})

 for link in dataInFocus.find_all('a', {'href': re.compile("^/wiki")}):
  if ':' not in link.get('href'):
   finalUrl = "https://en.wikipedia.org" + link.get('href')
   listWithoutHash = finalUrl.split('#')
   tempLinks.append(str(listWithoutHash[0]))

 return deleteDuplicates(uniqueLinks,tempLinks,list_crawled,list_master)
```

```
# The below one is a recursive function which traverses to different
depths upto depth 6
def nextLinkToCrawl(listInUse, list_depth1, list_depth2, list_depth3,
list_depth4, list_depth5, list_depth6,list_crawled):

    for a in listInUse:
        if a not in list_crawled:
            return a



    if 0 == cmp(listInUse,list_depth1):
        if len(list_depth1)<1000:
            return nextLinkToCrawl(list_depth2,list_depth1,
list_depth2, list_depth3, list_depth4, list_depth5,
list_depth6,list_crawled)
    if 0 == cmp(listInUse,list_depth2):
        if len(list_depth2)<1000:
            return nextLinkToCrawl(list_depth3,list_depth1,
list_depth2, list_depth3, list_depth4, list_depth5,
list_depth6,list_crawled)
    if 0 == cmp(listInUse,list_depth3):
        if len(list_depth3)<1000:
            return nextLinkToCrawl(list_depth4,list_depth1,
list_depth2, list_depth3, list_depth4, list_depth5,
list_depth6,list_crawled)
    if 0 == cmp(listInUse,list_depth4):
```

```
                if len(list_depth4)<1000:
                        return nextLinkToCrawl(list_depth5,list_depth1,
list_depth2, list_depth3, list_depth4, list_depth5,
list_depth6,list_crawled)
        if 0 == cmp(listInUse,list_depth5):
                if len(list_depth5)<1000:
                        return nextLinkToCrawl(list_depth6,list_depth1,
list_depth2, list_depth3, list_depth4, list_depth5,
list_depth6,list_crawled)

        return 'links not found'

# the below function checks whether nextPageUrl is part of which depth
def
listBelongsTo(nextPageUrl,list_depth1,list_depth2,list_depth3,list_depth
4,list_depth5,list_depth6):

        if nextPageUrl in list_depth1:
                return list_depth1

        elif nextPageUrl in list_depth2:
                return list_depth2

        elif nextPageUrl in list_depth3:
                return list_depth3

        elif nextPageUrl in list_depth4:
```

```
                return list_depth4

        else:
                return list_depth5

def mainWebCrawler(url):

        list_master = []
        list_depth1 = []
        list_depth2 = []
        list_depth3 = []
        list_depth4 = []
        list_depth5 = []
        list_depth6 = []
        list_crawled = []


        list_master.append(url)
        list_depth1.append(url)

        while len(list_master) < 1000:

                nextPageUrl =
nextLinkToCrawl(list_depth1,list_depth1,list_depth2,list_depth3,list_dep
th4,list_depth5,list_depth6,list_crawled)
                if nextPageUrl == 'links not found':
                        print "crawling ends:no further links found"
```

```
                    break

                else:

                    listx =
listBelongsTo(nextPageUrl,list_depth1,list_depth2,list_depth3,list_depth
4,list_depth5,list_depth6)

                    if listx == list_depth1:
                        list_depth1_urls =
hitUrl(nextPageUrl,list_crawled,list_master)
                        for a in list_depth1_urls:
                            list_depth2.append(a)
                    elif listx == list_depth2:
                        list_depth2_urls =
hitUrl(nextPageUrl,list_crawled,list_master)
                        for b in list_depth2_urls:
                            if (b not in list_depth1) and (b not in
list_depth2):
                                list_depth3.append(b)
                    elif listx == list_depth3:
                        list_depth3_urls =
hitUrl(nextPageUrl,list_crawled,list_master)
                        for c in list_depth3_urls:
                            if (c not in list_depth2) and (b not in
list_depth3):
                                list_depth4.append(c)
```

```
                elif listx == list_depth4:
                        list_depth4_urls =
hitUrl(nextPageUrl,list_crawled,list_master)
                        for d in list_depth4_urls:
                                if (d not in list_depth3) and (d not in
list_depth4):
                                        list_depth5.append(d)

                elif listx == list_depth5:
                        list_depth5_urls =
hitUrl(nextPageUrl,list_crawled,list_master)
                        for e in list_depth5_urls:
                                if (e not in list_depth4) and (e not in
list_depth5):
                                        list_depth6.append(e)
        file = open('TASK 1-E.txt', 'w')

        for i,url in enumerate(list_master):
                if i < 1000:
                 file.write(str(url.lower()) + "\n")

        file.close()

url = "https://en.wikipedia.org/wiki/Tropical_cyclone"
mainWebCrawler(url)
```

DFS:

```
import requests
from bs4 import BeautifulSoup
import re
import time
import sys



# Goal: Write focused web crawling


# the below functions deletes duplicates and adds to the master list.


def deleteDuplicates(uniqueLinks,tempLinks,list_crawled,list_master):
 for a in tempLinks:
  if a not in uniqueLinks:
   if len(a) > 1:
    if a not in list_crawled:
     uniqueLinks.append(a)


 # Add unique links to the master list
 for a in uniqueLinks:
  if a not in list_master:
   list_master.append(a)


 return uniqueLinks


# the below function crawls the given nextUrl
```

```
# Removes references and external links from the fetched Urls
# Also returns the master list only with the Urls containing 'Rain' and also
Urls with the anchor tags
# containing texts with the string 'rain'
def hitUrl(nextUrl,list_crawled,list_master,key):

 reExpressionString="^%s| %s$|%s | %s|%s|%s| %s_| %s
"%(key,key,key,key,key,key,key,key)
 reExpression = re.compile(reExpressionString,re.IGNORECASE)

 list_crawled.append(nextUrl)

 uniqueLinks = []
 tempLinks = []

 time.sleep(1)

 data = requests.get(nextUrl)

 data_text = data.text

 soup = BeautifulSoup(data_text,'html.parser')

 dataInFocus = soup.find('div',{'id': 'mw-content-text'})

 # Removes references and links that comes under the image.
 if len(soup.find('ol', attrs={'class': 'references'}) or ()) > 1:
```

```
 soup.find('ol', attrs={'class': 'references'}).decompose()
if len(soup.find('div', attrs={'class': 'thumb tright'}) or ()) > 1:
 soup.find('div', attrs={'class': 'thumb tright'}).decompose()

for link in dataInFocus.find_all('a', {'href': re.compile("^/wiki")}):
 hrefString = link.get('href')
 len1=len(reExpression.findall(hrefString))

 try:
  anchorTextString = str(link.text)
 except UnicodeEncodeError as e:
  error = e
 len2=len(reExpression.findall(anchorTextString))

 if (key.lower() in str(link.get('href')).lower()) or (key.lower() in
anchorTextString.lower()):
  if(len1 > 0) or (len2 > 0):
          if ':' not in link.get('href'):
           finalUrl = "https://en.wikipedia.org" + link.get('href')
           listWithoutHash = finalUrl.split('#')
           tempLinks.append(str(listWithoutHash[0]))

 return deleteDuplicates(uniqueLinks,tempLinks,list_crawled,list_master)


# The below one is a recursive function which traverses to different
depths
```

```
def nextLinkToCrawl(listInUse, list_depth1, list_depth2, list_depth3,
list_depth4, list_depth5, list_depth6,list_crawled):

    for a in listInUse:
        if a not in list_crawled:
            return a



    if 0 == cmp(listInUse,list_depth1):
        if len(list_depth1)<1000:
            return nextLinkToCrawl(list_depth2,list_depth1,
list_depth2, list_depth3, list_depth4, list_depth5,
list_depth6,list_crawled)
    if 0 == cmp(listInUse,list_depth2):
        if len(list_depth2)<1000:
            return nextLinkToCrawl(list_depth3,list_depth1,
list_depth2, list_depth3, list_depth4, list_depth5,
list_depth6,list_crawled)
    if 0 == cmp(listInUse,list_depth3):
        if len(list_depth3)<1000:
            return nextLinkToCrawl(list_depth4,list_depth1,
list_depth2, list_depth3, list_depth4, list_depth5,
list_depth6,list_crawled)
    '''if listInUse==list_depth4:
        if len(list_depth4)<1000:
            return nextLinkToCrawl(list_depth5,list_depth1,
list_depth2, list_depth3, list_depth4, list_depth5,
```

```
list_depth6,list_crawled)
        if listInUse==list_depth5:
                if len(list_depth5)<500:
                        return nextLinkToCrawl(list_depth6,list_depth1,
list_depth2, list_depth3, list_depth4, list_depth5,
list_depth6,list_crawled)'''

        return 'links not found'



# the below function checks whether nextPageUrl is part of which depth
def
listBelongsTo(nextPageUrl,list_depth1,list_depth2,list_depth3,list_depth
4,list_depth5,list_depth6):

        if nextPageUrl in list_depth1:
                return list_depth1

        elif nextPageUrl in list_depth2:
                return list_depth2

        elif nextPageUrl in list_depth3:
                return list_depth3

        elif nextPageUrl in list_depth4:
                return list_depth4
```

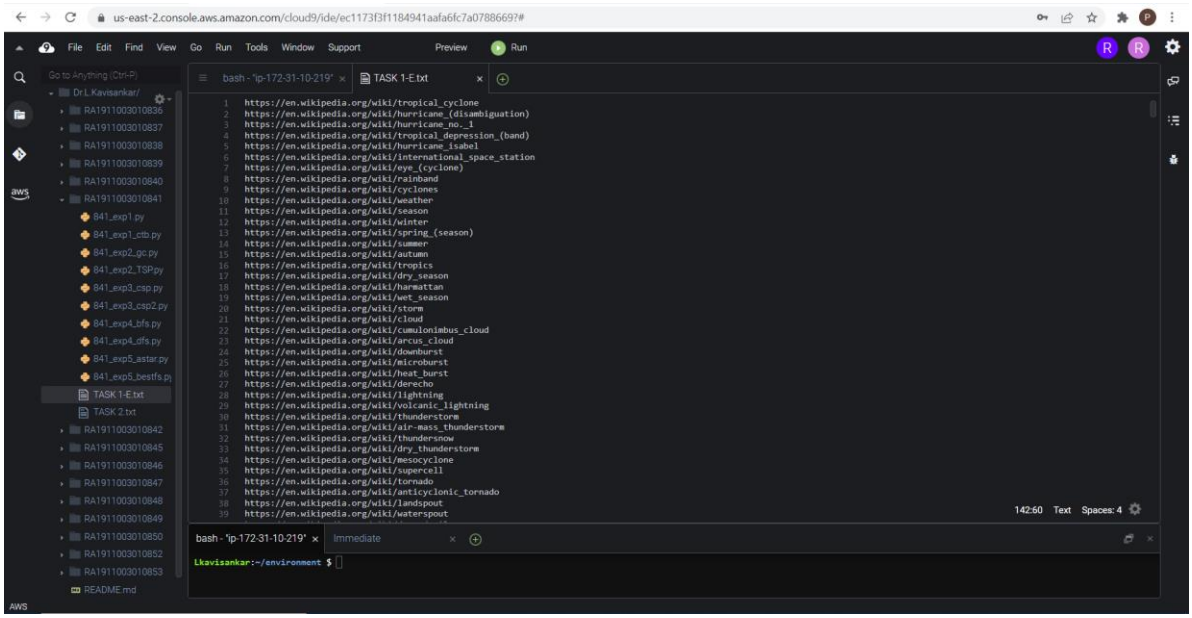
```
        else:
                return list_depth5
def mainWebCrawler(url,key):

        list_master = []
        list_depth1 = []
        list_depth2 = []
        list_depth3 = []
        list_depth4 = []
        list_depth5 = []
        list_depth6 = []
        list_crawled = []



        list_master.append(url)
        list_depth1.append(url)

        while len(list_master) < 1000:

                nextPageUrl =
nextLinkToCrawl(list_depth1,list_depth1,list_depth2,list_depth3,list_dep
th4,list_depth5,list_depth6,list_crawled)
                if nextPageUrl == 'links not found':
                        print "crawling ends:no further links found"
                        break

                else:
```

```
                listx =
listBelongsTo(nextPageUrl,list_depth1,list_depth2,list_depth3,list_depth
4,list_depth5,list_depth6)

                if listx == list_depth1:
                        list_depth1_urls =
hitUrl(nextPageUrl,list_crawled,list_master,key)
                        for a in list_depth1_urls:
                                list_depth2.append(a)
                elif listx == list_depth2:
                        list_depth2_urls =
hitUrl(nextPageUrl,list_crawled,list_master,key)
                        for b in list_depth2_urls:
                                if (b not in list_depth1) and (b not in
list_depth2):
                                        list_depth3.append(b)
                elif listx == list_depth3:
                        list_depth3_urls =
hitUrl(nextPageUrl,list_crawled,list_master,key)
                        for c in list_depth3_urls:
                                if (c not in list_depth2) and (b not in
list_depth3):
                                        list_depth4.append(c)

                elif listx == list_depth4:
                        list_depth4_urls =
```

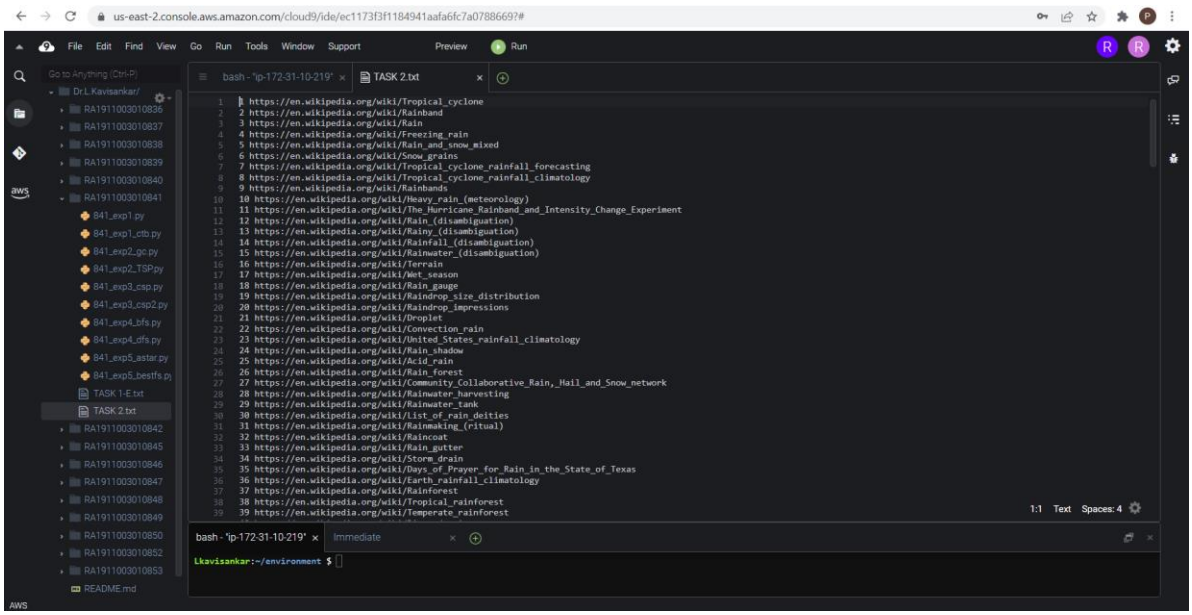
```
hitUrl(nextPageUrl,list_crawled,list_master,key)
                                    for d in list_depth4_urls:
                                            if (d not in list_depth3) and (d not in
list_depth4):
                                                    list_depth5.append(d)


                            elif listx == list_depth5:
                                    list_depth5_urls =
hitUrl(nextPageUrl,list_crawled,list_master,key)
                                    for e in list_depth5_urls:
                                            if (e not in list_depth4) and (e not in
list_depth5):
                                                    list_depth6.append(e)
        file = open('TASK 2.txt', 'w')


        for i,url in enumerate(list_master):
                if i < 1000:
                 file.write(str(i+1) + " " +str(url) + "\n")


        file.close()


# since recusrsion is used in this program, recusrion limit is set to avoid
python from crashing.
sys.setrecursionlimit(20000)
key = "rain"
url = "https://en.wikipedia.org/wiki/Tropical_cyclone"
mainWebCrawler(url,key)
```

**Screenshot of output:**

BFS:




DFS:




**Result:**

Thus, DFS and BFS were traversed, implemented and analyzed

successfully for a simple web page.

# *Title: Best first Search*

**Ex. No.: 5** **Reg. No.:**

**Date:** **Name:**

**Aim:** To write a program to implement the best first search algorithm.

**Procedure/Algorithm:**

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the priority queue.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN list or priority queue. If the node has not been in both, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

**Program:**

```
from Queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]

def best_first_search(source, target, n):
   visited = [False] * n
   visited[0] = True
   pq = PriorityQueue()
   pq.put((0, source))
   while pq.empty() == False:
      u = pq.get()[1]
      print(u)
      if u == target:
         break

      for v, c in graph[u]:
         if visited[v] == False:
            visited[v] = True
            pq.put((c, v))

def addedge(x, y, cost):
   graph[x].append((y, cost))
   graph[y].append((x, cost))

addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)

source = 0
target = 9
best_first_search(source, target, v)
```

**Screenshot of output:**


**Result:**

Thus, the best first search algorithm was implemented successfully.

# *Title: A* Algorithm*

**Ex. No.: 5** **Reg. No.:**

**Date:** **Name:**

**Aim:** To write a program to implement the A* algorithm for real world problems.

**Procedure/Algorithm:**

1. Initialize the open list
2. Initialize the closed list put the starting node on the open list (you can leave its f at zero)
3. while the open list is not empty
    a) find the node with the least f on the open list, call it "q"
    b) pop q off the open list
    c) generate q's 8 successors and set their parents to q
    d) for each successor
        i) if successor is the goal, stop search
        ii) else, compute both g and h for successor
            successor.g = q.g + distance between successor and q
            successor.h = distance from goal to successor (This can be done using many ways, we will discuss three heuristicsManhattan, Diagonal and Euclidean Heuristics)
            successor.f = successor.g + successor.h
        iii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
        iv) if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor otherwise, add the node to the open list end (for loop)
    e) push q on the closed list end (while loop)

**Program:**

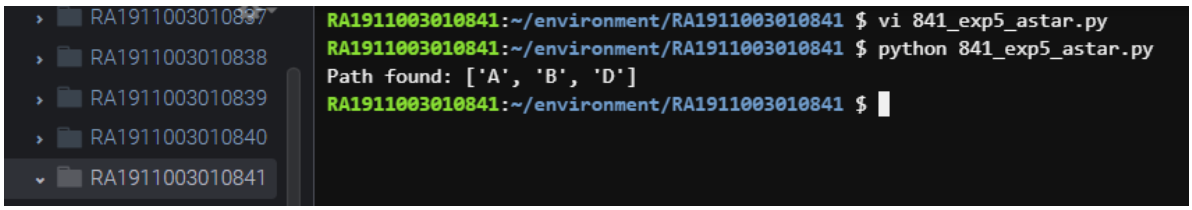
```
from collections import deque

class Graph:

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        open_list = set([start_node])
        closed_list = set([])
        g = {}
        g[start_node] = 0
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None
            if n == stop_node:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)

                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path
```

```
            for (m, weight) in self.get_neighbors(n):
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None

adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

**Manual Output:**

Ratings[]={1,3,4,3,7,1}
Exp => 1+2+3+1+2+1 = 10

**Screenshot of output:**




**Result:**

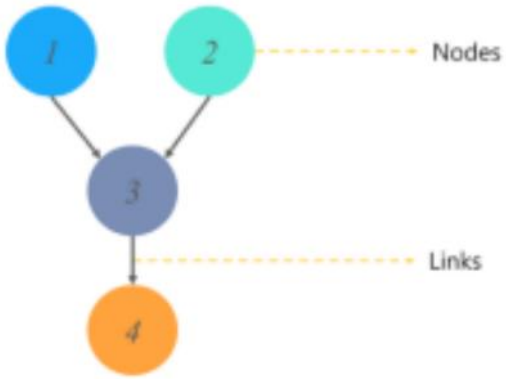
Thus, the program for A* algorithm was implemented successfully.

# *Title: Implementation of uncertain methods*

**Ex. No.: 6** **Reg. No.:**
**Date:** **Name:**

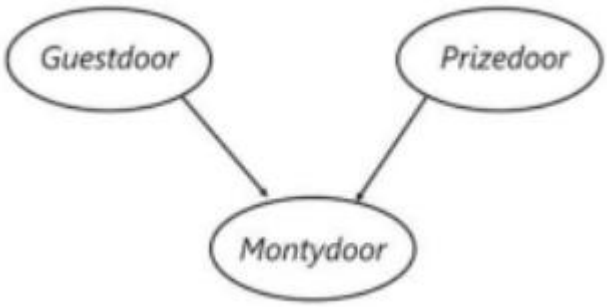
**Aim:** To implement Bayesian Belief Networks to model the problem of Monty.

**Introduction to Bayesian Network:** Bayesian networks are probabilistic models that are especially good at inference given incomplete data. These Belief Networks are used to model uncertainties by using Directed Acyclic Graphs (DAG). A Directed Acyclic Graph is used to represent a Bayesian Network and like any other statistical graph, a DAG contains a set of nodes and links, where the links denote the relationship between the nodes. The nodes here represent random variables and the edges define the relationship between these variables. A DAG models the uncertainty of an event occurring based on the Conditional Probability Distribution (CDP) of each random variable. A Conditional Probability Table (CPT) is used to represent the CPD of each variable in the network.




**The Monty Hall problem:**
The Monty Hall problem is a brain teaser, in the form of a probability puzzle named after the host of the TV series, 'Let's Make A Deal'. The game involves three doors, given that behind one of these doors is a car and the remaining two have goats behind them. So, you start by picking a random door, say #2. On the other hand, the host knows where the car is hidden and he opens another door, say #1 (behind which there is a goat). You are now given a choice; the host will ask you if you want to pick door #3 instead of your first choice i.e., #2. A Bayesian Network is created to understand the probability of winning if the participant decides to switch his choice.

**Directed Acyclic Graph of Monty Hall:**



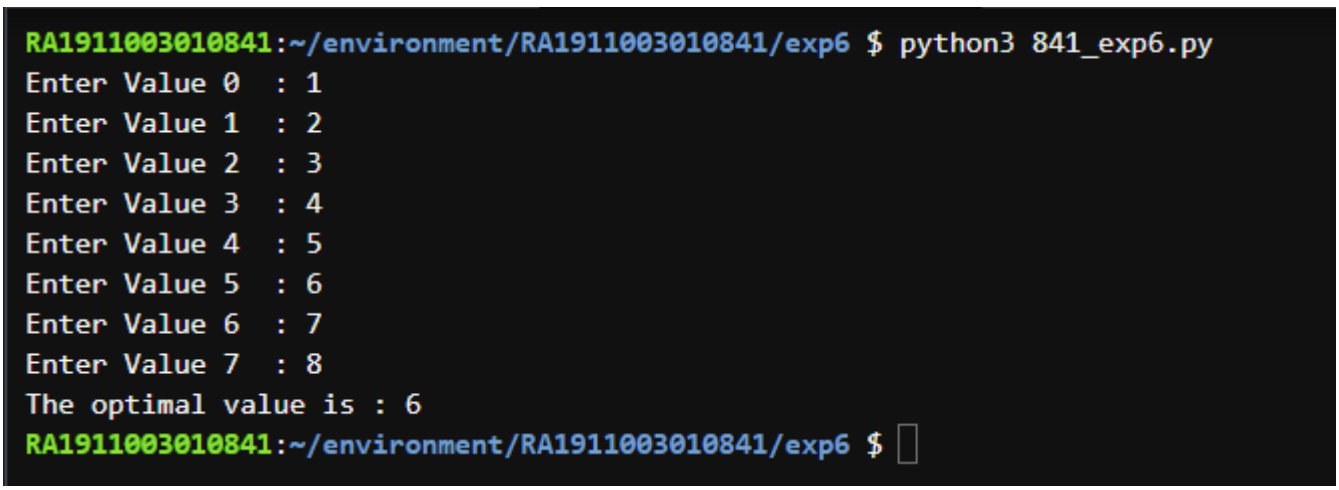

The graph has three nodes, each representing the door chosen by:
1. The door selected by the Guest
2. The door containing the prize (car)
3. The door Monty chooses to open

**Understanding the dependencies:**
Here, the door selected by the guest and the door containing the car are completely random processes. However, the door Monty chooses to open is dependent on both the doors; the door selected by the guest, and the door the prize is behind. Monty has to choose in such a way that the door does not contain the prize and it cannot be the one chosen by the guest.

**Program:**

```
MAX, MIN = 1000, -1000
def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
        if depth == 3:
                return values[nodeIndex]
        if maximizingPlayer:
                best = MIN
                for i in range(0, 2):
                        val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
                        best = max(best, val)
                        alpha = max(alpha, best)
                        if beta <= alpha:
                                break
                return best
        else:
                best = MAX
                for i in range(0, 2):
                        val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
                        best = min(best, val)
                        beta = min(beta, best)
                        if beta <= alpha:
                                break
                return best

if __name__ == "__main__":
   values = []
   for i in range(0, 8):
      x = int(input(f"Enter Value {i}  : "))
      values.append(x)
   print ("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

**Screenshot of output:**




**Result:**

Thus, Bayesian Belief Networks to model the problem of Monty Hall was implemented successfully.

# *Title: Implementation of unification and resolution on real world problems.*

**Ex. No.: 7** **Reg. No.:**
**Date:** **Name:**

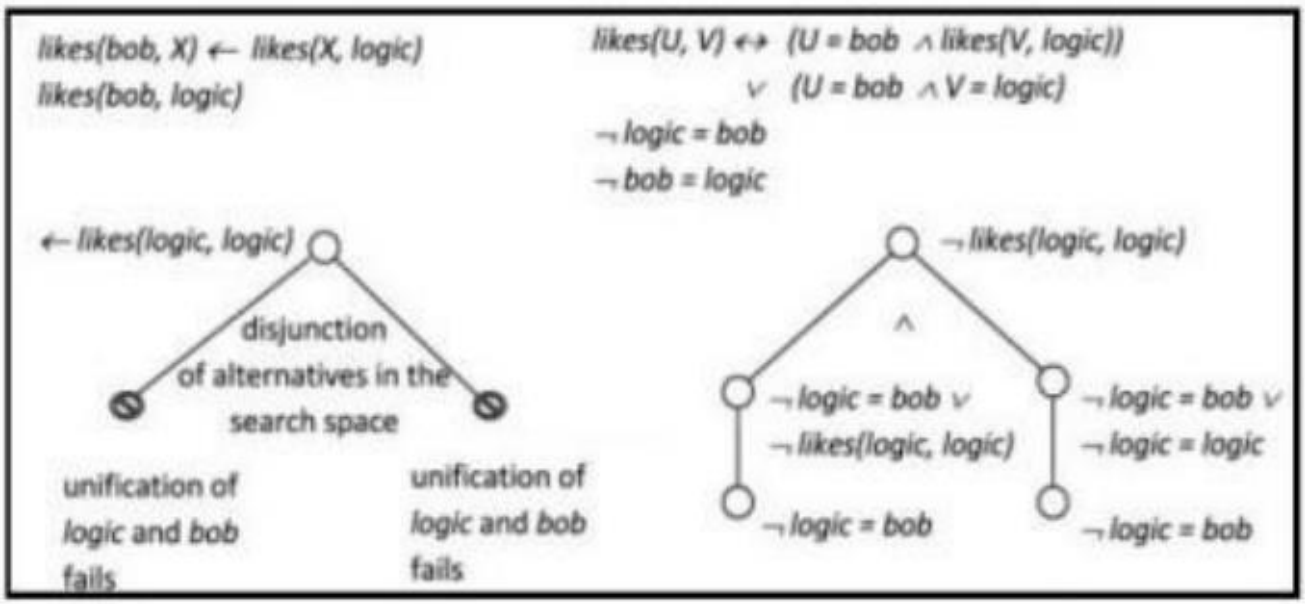
**Aim:** Develop a program to unify expressions and direct the output of resolution to output.txt after taking input from input.txt file in same directory

**Procedure/Algorithm:**

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF.
3. Negate the statement which needs to prove (proof by contradiction).
4. Draw resolution graph (unification).

Traugott Rules:

$$
\begin{array}{ccc}
a \rightarrow b \wedge (true \rightarrow d) & \Longrightarrow & a \rightarrow b \wedge d \\
a \rightarrow (true \rightarrow e) & \Longrightarrow & a \rightarrow e \\
\neg true & \Longrightarrow & false
\end{array}
$$



**Program:**

```
import copy
import time

class Parameter:
    variable_count = 1

    def __init__(self, name=None):
        if name:
            self.type = "Constant"
            self.name = name
        else:
            self.type = "Variable"
            self.name = "v" + str(Parameter.variable_count)
            Parameter.variable_count += 1

    def isConstant(self):
        return self.type == "Constant"

    def unify(self, type_, name):
        self.type = type_
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

    def __str__(self):
        return self.name

class Predicate:
    def __init__(self, name, params):
        self.name = name
        self.params = params

    def __eq__(self, other):
        return self.name == other.name and all(a == b for a, b in zip(self.params,
other.params))

    def __str__(self):
        return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

    def getNegatedPredicate(self):
        return Predicate(negatePredicate(self.name), self.params)


class Sentence:
    sentence_count = 0

    def __init__(self, string):
        self.sentence_index = Sentence.sentence_count
        Sentence.sentence_count += 1
        self.predicates = []
```

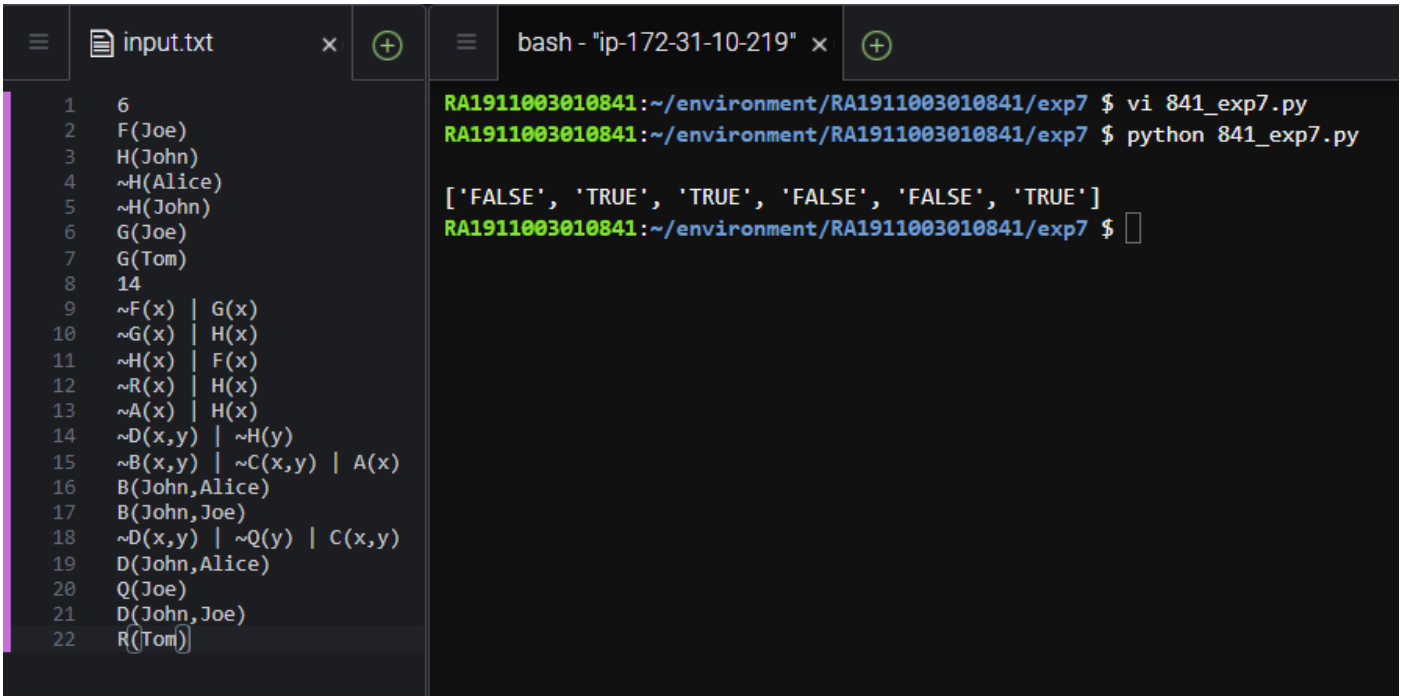
```
        self.variable_map = {}
        local = {}

        for predicate in string.split("|"):
            name = predicate[:predicate.find("(")]
            params = []

            for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):
                if param[0].islower():
                    if param not in local:  # Variable
                        local[param] = Parameter()
                        self.variable_map[local[param].name] = local[param]
                    new_param = local[param]
                else:
                    new_param = Parameter(param)
                    self.variable_map[param] = new_param

                params.append(new_param)

            self.predicates.append(Predicate(name, params))

    def getPredicates(self):
        return [predicate.name for predicate in self.predicates]

    def findPredicates(self, name):
        return [predicate for predicate in self.predicates if predicate.name == name]

    def removePredicate(self, predicate):
        self.predicates.remove(predicate)
        for key, val in self.variable_map.items():
            if not val:
                self.variable_map.pop(key)

    def containsVariable(self):
        return any(not param.isConstant() for param in self.variable_map.values())

    def __eq__(self, other):
        if len(self.predicates) == 1 and self.predicates[0] == other:
            return True
        return False

    def __str__(self):
        return "".join([str(predicate) for predicate in self.predicates])

class KB:
    def __init__(self, inputSentences):
        self.inputSentences = [x.replace(" ", "") for x in inputSentences]
        self.sentences = []
        self.sentence_map = {}

    def prepareKB(self):
        self.convertSentencesToCNF()
```

```
        for sentence_string in self.inputSentences:
            sentence = Sentence(sentence_string)
            for predicate in sentence.getPredicates():
                self.sentence_map[predicate] = self.sentence_map.get(predicate, []) +
[sentence]

    def convertSentencesToCNF(self):
        for sentenceIdx in range(len(self.inputSentences)):
            if "=>" in self.inputSentences[sentenceIdx]:  # Do negation of the Premise and add
them as literal
                self.inputSentences[sentenceIdx] =
negateAntecedent(self.inputSentences[sentenceIdx])

    def askQueries(self, queryList):
        results = []

        for query in queryList:
            negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))
            negatedPredicate = negatedQuery.predicates[0]
            prev_sentence_map = copy.deepcopy(self.sentence_map)
            self.sentence_map[negatedPredicate.name] =
self.sentence_map.get(negatedPredicate.name, []) + [negatedQuery]
            self.timeLimit = time.time() + 40

            try:
                result = self.resolve([negatedPredicate], [False]*(len(self.inputSentences) + 1))
            except:
                result = False

            self.sentence_map = prev_sentence_map

            if result:
                results.append("TRUE")
            else:
                results.append("FALSE")

        return results

    def resolve(self, queryStack, visited, depth=0):
        if time.time() > self.timeLimit:
            raise Exception
        if queryStack:
            query = queryStack.pop(-1)
            negatedQuery = query.getNegatedPredicate()
            queryPredicateName = negatedQuery.name
            if queryPredicateName not in self.sentence_map:
                return False
            else:
                queryPredicate = negatedQuery
                for kb_sentence in self.sentence_map[queryPredicateName]:
                    if not visited[kb_sentence.sentence_index]:
                        for kbPredicate in kb_sentence.findPredicates(queryPredicateName):
```

```
                canUnify, substitution =
performUnification(copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

                if canUnify:
                    newSentence = copy.deepcopy(kb_sentence)
                    newSentence.removePredicate(kbPredicate)
                    newQueryStack = copy.deepcopy(queryStack)

                    if substitution:
                        for old, new in substitution.items():
                            if old in newSentence.variable_map:
                                parameter = newSentence.variable_map[old]
                                newSentence.variable_map.pop(old)
                                parameter.unify("Variable" if new[0].islower() else "Constant",
new)
                                newSentence.variable_map[new] = parameter

                        for predicate in newQueryStack:
                            for index, param in enumerate(predicate.params):
                                if param.name in substitution:
                                    new = substitution[param.name]
                                    predicate.params[index].unify("Variable" if new[0].islower()
else "Constant", new)

                    for predicate in newSentence.predicates:
                        newQueryStack.append(predicate)

                    new_visited = copy.deepcopy(visited)
                    if kb_sentence.containsVariable() and len(kb_sentence.predicates) > 1:
                        new_visited[kb_sentence.sentence_index] = True

                    if self.resolve(newQueryStack, new_visited, depth + 1):
                        return True
            return False
        return True


def performUnification(queryPredicate, kbPredicate):
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params, kbPredicate.params):
            if query == kb:
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
```

```
                query.unify("Constant", kb.name)
            else:
                return False, {}
        else:
            if not query.isConstant():
                if kb.name not in substitution:
                    substitution[kb.name] = query.name
                elif substitution[kb.name] != query.name:
                    return False, {}
                kb.unify("Variable", query.name)
            else:
                if kb.name not in substitution:
                    substitution[kb.name] = query.name
                elif substitution[kb.name] != query.name:
                    return False, {}
    return True, substitution

def negatePredicate(predicate):
    return predicate[1:] if predicate[0] == "~" else "~" + predicate

def negateAntecedent(sentence):
    antecedent = sentence[:sentence.find("=>")]
    premise = []

    for predicate in antecedent.split("&"):
        premise.append(negatePredicate(predicate))

    premise.append(sentence[sentence.find("=>") + 2:])
    return "|".join(premise)

def getInput(filename):
    with open(filename, "r") as file:
        noOfQueries = int(file.readline().strip())
        inputQueries = [file.readline().strip() for _ in range(noOfQueries)]
        noOfSentences = int(file.readline().strip())
        inputSentences = [file.readline().strip() for _ in range(noOfSentences)]
        return inputQueries, inputSentences

def printOutput(filename, results):
    print(results)
    with open(filename, "w") as file:
        for line in results:
            file.write(line)
            file.write("\n")
    file.close()

if __name__ == '__main__':
    inputQueries_, inputSentences_ = getInput("input.txt")
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("output.txt", results_)
```

**Screenshot of output:**




**Result:** Thus, unification and resolution for real world problems was implemented successfully.

# *Title: Unsupervised Learning Methods*

**Ex. No.: 8** **Reg. No.:**
**Date:** **Name:**

**Aim:** To implement Unsupervised Learning Models (K-means clustering and K-Nearest Neighbours).

**Procedure/Algorithm:**

**K-Means Clustering:**

The working of the K-Means algorithm is explained in the below steps:

Step-1: Select the number K to decide the number of clusters.
Step-2: Select random K points or centroids. (It can be other from the input dataset).
Step-3: Assign each data point to their closest centroid, which will form the predefined K clusters.
Step-4: Calculate the variance and place a new centroid of each cluster.
Step-5: Repeat the third steps, which means reassign each datapoint to the new closest centroid of each cluster.
Step-6: If any reassignment occurs, then go to step-4 else go to FINISH.
Step-7: The model is ready.

**K-Nearest Neighbors:**

The K-NN working can be explained on the basis of the below algorithm:

Step-1**:** Select the number K of the neighbours.
Step-2**:** Calculate the Euclidean distance of **K number of neighbours**
Step-3**:** Take the K nearest neighbours as per the calculated Euclidean distance.
Step-4**:** Among these k neighbours, count the number of the data points in each category.
Step-5**:** Assign the new data points to that category for which the number of the neighbour is maximum.
Step-6**:** Our model is ready.

**Program:**

**K-Means Clustering:**

```
import numpy as np
from sklearn.datasets import make_blobs
from sklearn import datasets
import matplotlib.pyplot as plt

np.random.seed(42)

X, y = make_blobs(centers=3, n_samples=500, n_features=2, shuffle=True, random_state=40)

def euclidean_distance(x1,x2):
  return np.sqrt(np.sum((x1 - x2)**2))

class KMeans:
  def __init__(self, K=5, max_iters=100, plot_steps=False):
    self.K = K
    self.max_iters = max_iters
    self.plot_steps = plot_steps

    #list of samples endices for each cluster
    self.clusters = [[] for _ in range(self.K)]

    #mean feature vector for each cluster
    self.centroids = []
```

```
def predict(self, X):
  self.X = X
  self.n_samples, self.n_features = X.shape

  #initialize centroids
  random_sample_idxs = np.random.choice(self.n_samples, self.K, replace = False)
  self.centroids = [self.X[idx] for idx in random_sample_idxs]

  #optimization
  for _ in range(self.max_iters):
    #update clusters
    self.clusters = self._create_clusters(self.centroids)
    if self.plot_steps:
      self.plot()

    #update centroids
    centroids_old = self.centroids
    self.centroids = self._get_centroids(self.clusters)


    #check if converged
    if self._is_converged(centroids_old, self.centroids):
      break

    if self.plot_steps:
      self.plot()

  #return cluster labels
  return self._get_cluster_labels(self.clusters)

def _get_cluster_labels(self, clusters):
  labels = np.empty(self.n_samples)
  for cluster_idx, cluster in enumerate(clusters):
    for sample_idx in cluster:
      labels[sample_idx] = cluster_idx
  return labels


def _create_clusters(self, centroids):
  clusters = [[] for _ in range(self.K)]
  for idx, sample in enumerate(self.X):
    centroid_idx = self._closest_centroid(sample, centroids)
    clusters[centroid_idx].append(idx)
  return clusters

def _closest_centroid(self, sample, centroids):
  distances = [euclidean_distance(sample,point) for point in centroids]
  closest_idx = np.argmin(distances)
  return closest_idx

def _get_centroids(self, clusters):
  centroids = np.zeros((self.K, self.n_features))
  for cluster_idx, cluster in enumerate(clusters):
    cluster_mean = np.mean(self.X[cluster], axis=0)
    centroids[cluster_idx] = cluster_mean
  return centroids
```

```
  def _is_converged(self, centroids_old, centroids):
    distances = [euclidean_distance(centroids_old[i], centroids[i]) for i in range(self.K)]
    return sum(distances) == 0

  def plot(self):
    fig, ax = plt.subplots(figsize=(12,8))

    for i, index in enumerate(self.clusters):
      point = self.X[index].T
      ax.scatter(*point)
    for point in self.centroids:
      ax.scatter(*point,marker='x',color='black',linewidth=2)

    plt.show()

clusters = len(np.unique(y))
print(clusters)
km = KMeans(K= clusters, max_iters=150, plot_steps=False)
y_pred = km.predict(X)
km.plot()
```

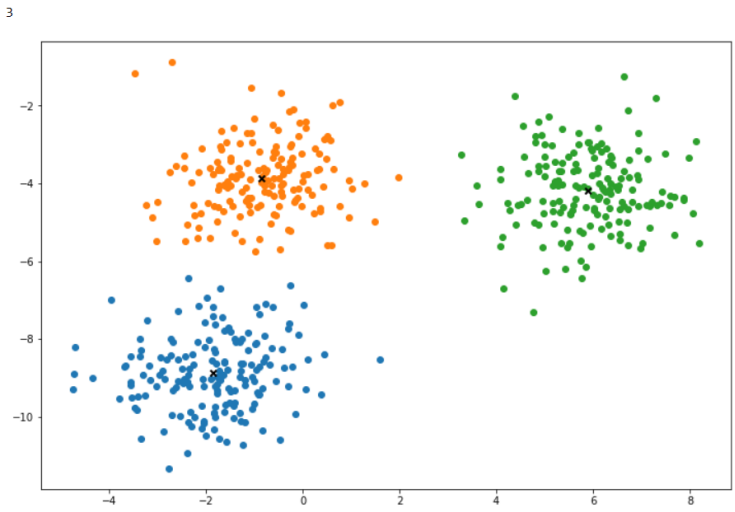
**K-Nearest Neighbors:**

```
from sklearn import datasets
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np
iris = datasets.load_iris()
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.33, random_state=42)

#Sepal Plot
plt.scatter(iris.data[:,:1],iris.data[:,1:2],c=iris.target, cmap=plt.cm.Dark2)
plt.title('Sepal plot')
plt.xlabel('sepal length')
plt.ylabel('sepal width')
plt.show()

#Petal Plot
plt.scatter(iris.data[:,2:3],iris.data[:,3:4],c=iris.target, cmap=plt.cm.Dark2)
plt.title('Petal plot')
plt.xlabel('petal length')
plt.ylabel('petal width')
plt.show()

from sklearn.neighbors import KNeighborsClassifier
#getting classfier using k = 9 and trained with training dataset
knn = KNeighborsClassifier(9)
knn.fit(X_train,y_train)

#now testing and check the accuracy at k = 9
from sklearn.metrics import accuracy_score
from sklearn.metrics import mean_squared_error
pred = knn.predict(X_test)
print (accuracy_score(y_test, pred))

#thats function getting classfier 1 to 30 and trained that classfier using training dataset and then testing
```

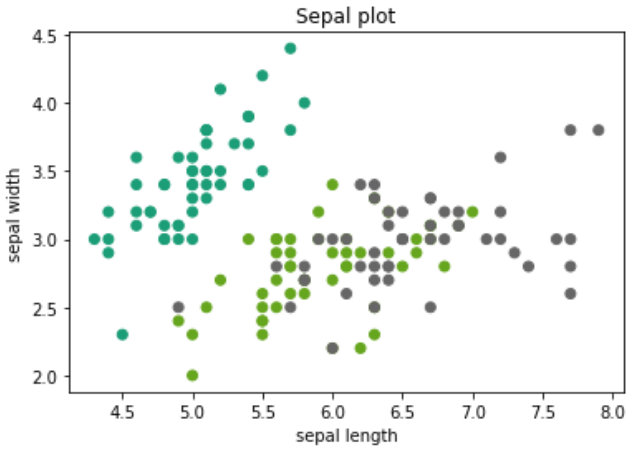
```
#at the end its drawing plots of accuracy and error with k 1 to 30

def compute(x_input,y_input,x_test):
    index = []
    accuracy = []
    error = []
    for K in range(30):
     K = K+1
     neigh = KNeighborsClassifier(n_neighbors = K)
     neigh.fit(x_input, y_input)
     y_pred = neigh.predict(x_test)
     index.append(K)
     accuracy.append(accuracy_score(y_test,y_pred)*100)
     error.append(mean_squared_error(y_test,y_pred)*100)
    plt.subplot(2,1,1)
    plt.plot(index,accuracy)
    plt.title('Accuracy')
    plt.xlabel('Value of K')
    plt.ylabel('Accuracy')

    plt.subplot(2,1,2)
    plt.plot(index,error,'r')
    plt.title('Error')
    plt.xlabel('Value of K')
    plt.ylabel('Error')
    plt.show()

compute(X_train,y_train,X_test)
```

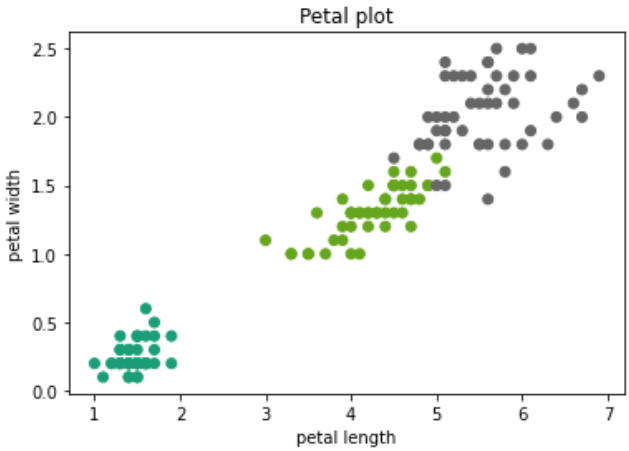
**Screenshot of output:**

**K-means Clustering:**

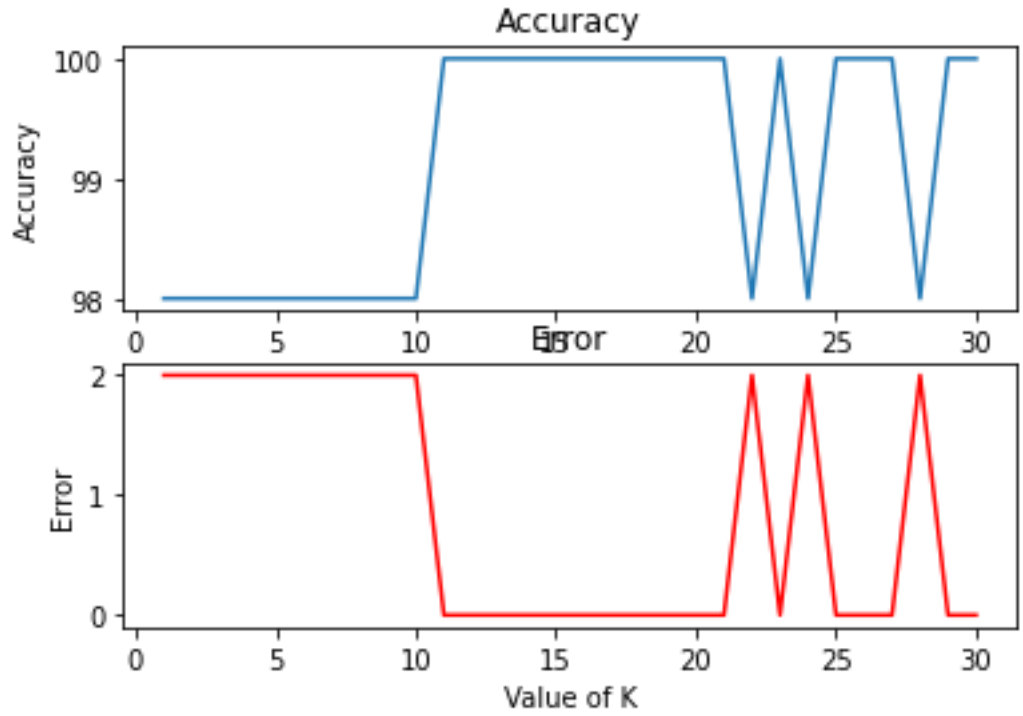


**K-Nearest Neighbors:**




Accuracy: 0.98




**Result:** Thus, the Unsupervised Learning Models were implemented successfully.

# *Title: Implementation of NLP Programs*

**Ex. No.: 9** **Reg. No.:**
**Date:** **Name:**

**Aim:** To implement RandomForestClassifier on SMSSpamCollection Dataset.

**Procedure/Algorithm:**
Random Forest works in two-phase first is to create the random forest by combining N decision trees, and second is to make predictions for each tree created in the first phase. The Working process can be explained in the below steps and diagram:

Step-1: Select random K data points from the training set.
Step-2: Build the decision trees associated with the selected data points (Subsets).
Step-3: Choose the number N for decision trees that you want to build.
Step-4: Repeat Step 1 & 2.
Step-5: For new data points, find the predictions of each decision tree, and assign the new data points to the category that wins the majority votes.

**Program:**

```
import nltk
import pandas as pd
import re
from sklearn.feature_extraction.text import TfidfVectorizer
import string

stopwords = nltk.corpus.stopwords.words('english')
ps = nltk.PorterStemmer()

data = pd.read_csv("SMSSpamCollection.tsv", sep='\t')
data.columns = ['label', 'body_text']

def count_punct(text):
   count = sum([1 for char in text if char in string.punctuation])
   return round(count/(len(text) - text.count(" ")), 3)*100

data['body_len'] = data['body_text'].apply(lambda x: len(x) - x.count(" "))
data['punct%'] = data['body_text'].apply(lambda x: count_punct(x))

def clean_text(text):
   text = "".join([word.lower() for word in text if word not in string.punctuation])
   tokens = re.split('\W+', text)
   text = [ps.stem(word) for word in tokens if word not in stopwords]
   return text

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(data[['body_text', 'body_len', 'punct%']], data['label'], test_size=0.2)

tfidf_vect = TfidfVectorizer(analyzer=clean_text)
tfidf_vect_fit = tfidf_vect.fit(X_train['body_text'])

tfidf_train = tfidf_vect_fit.transform(X_train['body_text'])
tfidf_test = tfidf_vect_fit.transform(X_test['body_text'])

X_train_vect = pd.concat([X_train[['body_len', 'punct%']].reset_index(drop=True),
```

```
        pd.DataFrame(tfidf_train.toarray())], axis=1)
X_test_vect = pd.concat([X_test[['body_len', 'punct%']].reset_index(drop=True),
        pd.DataFrame(tfidf_test.toarray())], axis=1)

from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.metrics import precision_recall_fscore_support as score
import time

rf = RandomForestClassifier(n_estimators=150, max_depth=None, n_jobs=-1)

start = time.time()
rf_model = rf.fit(X_train_vect, y_train)
end = time.time()
fit_time = (end - start)

start = time.time()
y_pred = rf_model.predict(X_test_vect)
end = time.time()
pred_time = (end - start)

precision, recall, fscore, train_support = score(y_test, y_pred, pos_label='spam', average='binary')
print('Fit time: {} / Predict time: {} ---- Precision: {} / Recall: {} / Accuracy: {}'.format(
   round(fit_time, 3), round(pred_time, 3), round(precision, 3), round(recall, 3),
round((y_pred==y_test).sum()/len(y_pred), 3)))

gb = GradientBoostingClassifier(n_estimators=150, max_depth=11)

start = time.time()
gb_model = gb.fit(X_train_vect, y_train)
end = time.time()
fit_time = (end - start)

start = time.time()
y_pred = gb_model.predict(X_test_vect)
end = time.time()
pred_time = (end - start)

precision, recall, fscore, train_support = score(y_test, y_pred, pos_label='spam', average='binary')
print('Fit time: {} / Predict time: {} ---- Precision: {} / Recall: {} / Accuracy: {}'.format(
   round(fit_time, 3), round(pred_time, 3), round(precision, 3), round(recall, 3),
round((y_pred==y_test).sum()/len(y_pred), 3)))
```

**Screenshot of output:**

In [6]:
```
f = RandomForestClassifier(n_estimators=150, max_depth=None, n_jobs=-1)

tart = time.time()
f_model = rf.fit(X_train_vect, y_train)
nd = time.time()
it_time = (end - start)

tart = time.time()
_pred = rf_model.predict(X_test_vect)
nd = time.time()
ed_time = (end - start)

ecision, recall, fscore, train_support = score(y_test, y_pred, pos_label='spam', average='binary')
int('Fit time: {} / Predict time: {} ---- Precision: {} / Recall: {} / Accuracy: {}'.format(
  round(fit_time, 3), round(pred_time, 3), round(precision, 3), round(recall, 3), round((y_pred==y_test).sum()/len(y_pred), 3)))
```

```
Fit time: 4.97 / Predict time: 0.191 ---- Precision: 1.0 / Recall: 0.87 / Accuracy: 0.984
```

In [7]:
```
) = GradientBoostingClassifier(n_estimators=150, max_depth=11)

tart = time.time()
)_model = gb.fit(X_train_vect, y_train)
nd = time.time()
it_time = (end - start)

tart = time.time()
_pred = gb_model.predict(X_test_vect)
nd = time.time()
ed_time = (end - start)

ecision, recall, fscore, train_support = score(y_test, y_pred, pos_label='spam', average='binary')
int('Fit time: {} / Predict time: {} ---- Precision: {} / Recall: {} / Accuracy: {}'.format(
  round(fit_time, 3), round(pred_time, 3), round(precision, 3), round(recall, 3), round((y_pred==y_test).sum()/len(y_pred), 3)))
```

```
Fit time: 225.224 / Predict time: 0.199 ---- Precision: 0.91 / Recall: 0.884 / Accuracy: 0.975
```

**Result:** Thus, the RandomForestClassifier was successfully implemented on SMSSpamCollection Dataset.

# *Title: Application of deep learning methods*

**Ex. No.: 10** **Reg. No.:**
**Date:** **Name:**

**Aim:** To implement CNN to classify Cifar-10 Images.

**Procedure/Algorithm:**

Step 1: Prepare Dataset for Training
Step 2: Create Training Data
Step 3: Assigning Labels and Features
Step 4: Normalizing X and converting labels to categorical data
Step 5: Split X and Y for use in CNN
Step 6: Define, compile and train the CNN Model

**Program:**

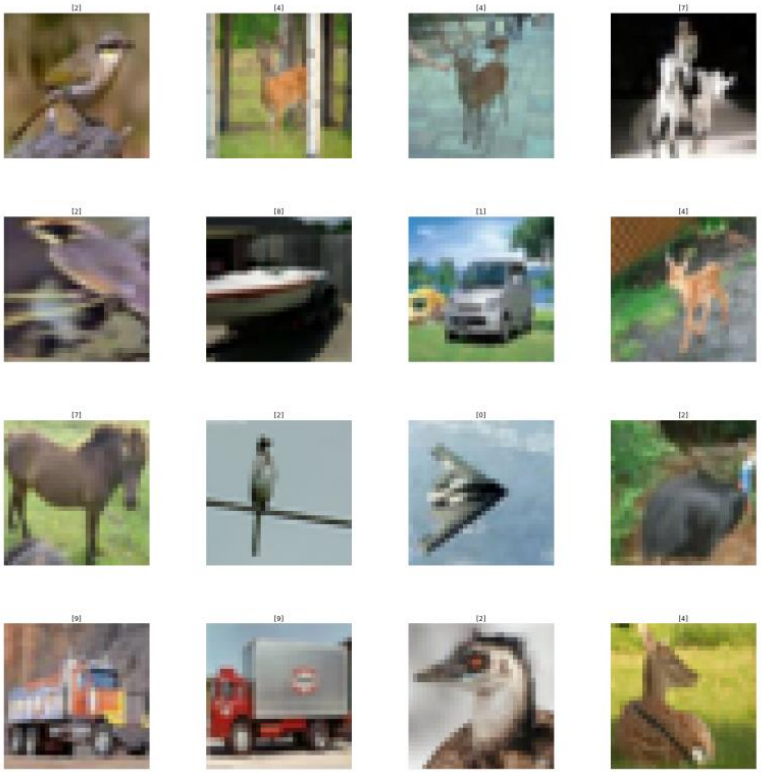
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn
from keras.datasets import cifar10
(X_train, y_train) , (X_test, y_test) = cifar10.load_data()

i = 30009
plt.imshow(X_train[i])
print(y_train[i])

W_grid = 4
L_grid = 4

fig, axes = plt.subplots(L_grid, W_grid, figsize = (25, 25))
axes = axes.ravel()

n_training = len(X_train)

for i in np.arange(0, L_grid * W_grid):
   index = np.random.randint(0, n_training) # pick a random number
   axes[i].imshow(X_train[index])
   axes[i].set_title(y_train[index])
   axes[i].axis('off')

plt.subplots_adjust(hspace = 0.4)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
number_cat = 10

from keras.utils.np_utils import to_categorical
y_train = to_categorical(y_train, number_cat)

y_test = to_categorical(y_test, number_cat)
y_test
```

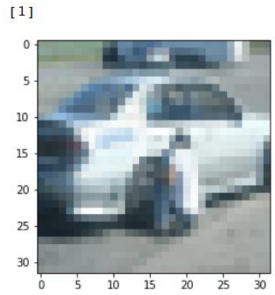
```
X_train = X_train/255
X_test = X_test/255

Input_shape = X_train.shape[1:]

import tensorflow as tf
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, AveragePooling2D, Dense, Flatten, Dropout
from tensorflow.keras.optimizers import Adam
from keras.callbacks import TensorBoard

cnn_model = Sequential()
cnn_model.add(Conv2D(filters = 64, kernel_size = (3,3), activation = 'relu', input_shape = Input_shape))
cnn_model.add(Conv2D(filters = 64, kernel_size = (3,3), activation = 'relu'))
cnn_model.add(MaxPooling2D(2,2))
cnn_model.add(Dropout(0.4))
cnn_model.add(Conv2D(filters = 128, kernel_size = (3,3), activation = 'relu'))
cnn_model.add(Conv2D(filters = 128, kernel_size = (3,3), activation = 'relu'))
cnn_model.add(MaxPooling2D(2,2))
cnn_model.add(Dropout(0.4))
cnn_model.add(Flatten())
cnn_model.add(Dense(units = 1024, activation = 'relu'))
cnn_model.add(Dense(units = 1024, activation = 'relu'))
cnn_model.add(Dense(units = 10, activation = 'softmax'))

cnn_model.compile(loss = 'categorical_crossentropy', optimizer = tf.keras.optimizers.RMSprop(lr = 0.001), metrics =
['accuracy'])

history = cnn_model.fit(X_train, y_train, batch_size = 32, epochs = 1, shuffle = True)

evaluation = cnn_model.evaluate(X_test, y_test)
print('Test Accuracy: {}'.format(evaluation[1]))

predicted_classes = cnn_model.predict(X_test)
predicted_classes
classes_x = np.argmax(predicted_classes,axis=1)
classes_x

y_test = y_test.argmax(1)

L = 7
W = 7
fig, axes = plt.subplots(L, W, figsize = (12, 12))
axes = axes.ravel()
for i in np.arange(0, L*W):
   axes[i].imshow(X_test[i])
   axes[i].set_title('Prediction = {}\n True = {}'.format(classes_x[i], y_test[i]))
   axes[i].axis('off')
plt.subplots_adjust(wspace = 1)

from sklearn.metrics import confusion_matrix
import seaborn as sns
cm = confusion_matrix(y_test, classes_x)
cm
plt.figure(figsize = (10, 10))
sns.heatmap(cm, annot = True)
```

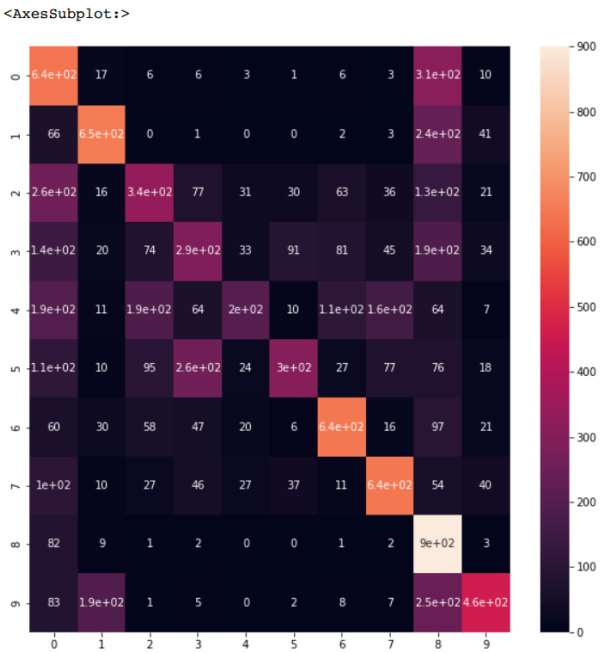
```
import os
directory = os.path.join(os.getcwd(), 'saved_models')
if not os.path.isdir(directory):
   os.makedirs(directory)
model_path = os.path.join(directory, 'keras_cifar10_trained_model.h5')
cnn_model.save(model_path)

import keras
from keras.datasets import cifar10
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

n = 8
X_train_sample = X_train[:n]

from keras.preprocessing.image import ImageDataGenerator
dataget_train = ImageDataGenerator(brightness_range=(1,3))
dataget_train.fit(X_train_sample)

from PIL import Image
fig = plt.figure(figsize=(20,2))
for x_batch in dataget_train.flow(X_train_sample,batch_size=n):
   for i in range(0,n):
      ax=fig.add_subplot(1,n,i+1)
      ax.imshow(Image.fromarray(np.uint8(x_batch[i])))
   fig.suptitle('Augmented Images (rotated by 90 degrees)')
   plt.show()
   break

from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
                     rotation_range = 90,
                     width_shift_range = 0.1,
                     horizontal_flip = True,
                     vertical_flip = True
                      )

datagen.fit(X_train)
cnn_model.fit_generator(datagen.flow(X_train, y_train, batch_size = 32), epochs = 2)

score = cnn_model.evaluate(X_test, y_test)
print('Test accuracy', score[1])

# save the model
directory = os.path.join(os.getcwd(), 'saved_models')
if not os.path.isdir(directory):
   os.makedirs(directory)
model_path = os.path.join(directory, 'keras_cifar10_trained_model_Augmentation.h5')
cnn_model.save(model_path)
```

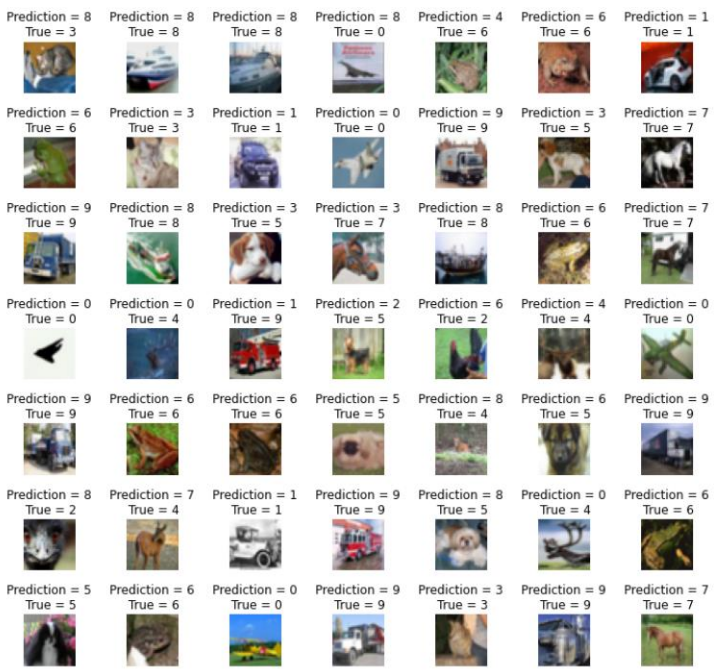
## Screenshots of output:

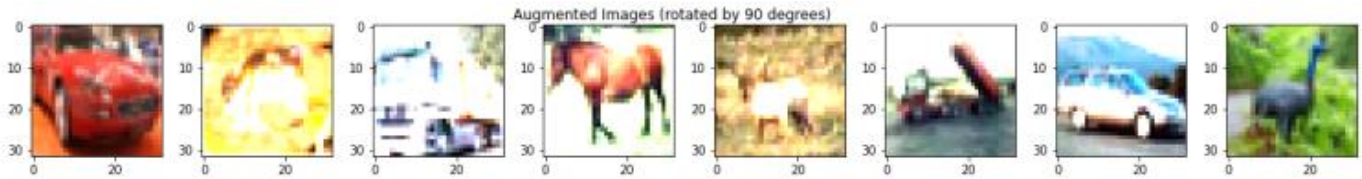
```
/opt/homebrew/Caskroom/miniforge/base/envs/tensorflow/lib/python3.9/site-packages/matplotlib/text.py:1223: FutureWarning: elementwise comparison failed; returning scalar instead, but in the future will perform elementwise comparison
  if s != self._text:
```






**Result:** Thus, CNN algorithm was successfully applied on Cifar-10 Images dataset.