SRM INSTITUTE OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND

ENGINEERING

18CSC305J - ARTIFICIAL INTELLIGENCE

SEMESTER – 6

BATCH – 1

REGISTRATION NUMBER	**RA1811028010040**
NAME	**ISH CHAUHAN**

B.tech-CSE-CC, Third Year (Section: J2)

Faculty Incharge: Vaishnavi Moorthy, Asst Prof/Dept of

CSE YEAR 2020-21

INDEX

Experiment No: 1

DATE : 20/01/21

LANGUAGE USED : PYTHON

EXPERIMENT TITLE : Maximum Subarray Problem

PROBLEM STATEMENT : Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

ALGORITHM :

Initialize:

max_so_far = 0

max_ending_here = 0

Loop for each element of the array:

(a) max_ending_here = max_ending_here + a[i]

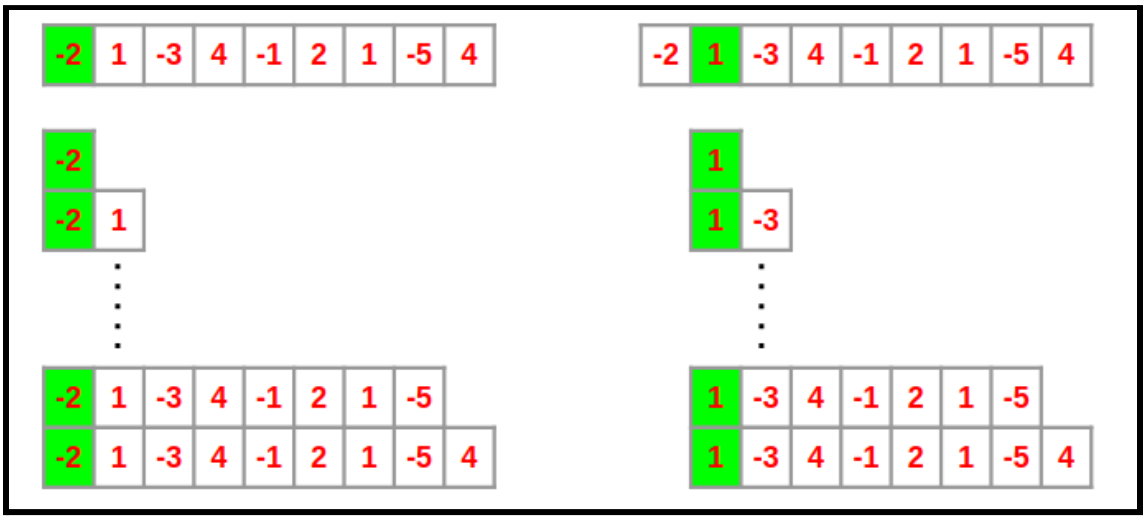
(b) if(max_so_far < max_ending_here)

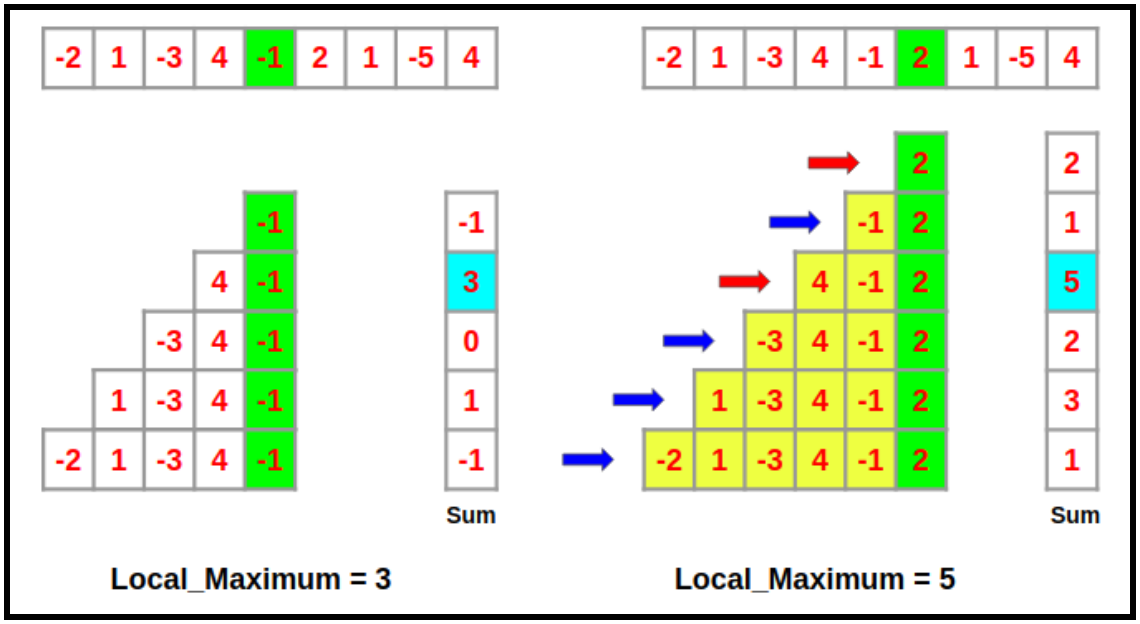
max_so_far = max_ending_here

(c) if(max_ending_here < 0)

max_ending_here = 0

return max_so_far





OPTIMIZATION TECHNIQUE:

1. Apparently, this is an optimization problem, which can be usually solved by Dynamic Programming (DP). So when it comes to DP, the first thing for us to figure out is the format of the sub problem(or the state of each sub problem).
2. The format of the sub problem can be helpful when we are trying to come up with the recursive relation.
3. At first, I think the sub problem should look like: maxSubArray(int A[], int i, int j), which means the maxSubArray for A[i: j]. In this way, our goal is to figure out what maxSubArray(A, 0, A.length - 1) is.
4. However, if we define the format of the sub problem in this way, it's hard to find the connection from the sub problem to the original problem(at least for me). In other words, I can't find a way to divide the original problem into the sub problems and use the solutions of the sub problems to somehow create the solution of the original one.
5. maxSubArray(int A[], int i), which means the maxSubArray for A[0:i] which must have A[i] as the end element. Note that now the sub problem's format is less flexible and less powerful than the previous one because there's a limitation that A[i] should be contained in that sequence and we have to keep track of each solution of the sub problem to update the global optimal value.

6. However, now the connect between the sub problem & the original one becomes clearer:

maxSubArray(A, i) = maxSubArray(A, i - 1) > 0 ? maxSubArray(A, i - 1) : 0 + A[i];

PROGRAM :

```
def maxSubArraySum(a,size):

  max_so_far = float("-inf")
  max_ending_here = 0

  for i in range(0, size):
     max_ending_here = max_ending_here + a[i]
     if (max_so_far < max_ending_here):
        max_so_far = max_ending_here
     if max_ending_here < 0:
        max_ending_here = 0
  return max_so_far
a = [-13, -3, -25, -20, -3, -16, -23, -12, -5, -22, -15, -4, -7]
print("Maximum contiguous sum is", maxSubArraySum(a,len(a)))
```

OUTPUT :

```
def maxSubArraySum(a,size):

   max_so_far = float("-inf")
   max_ending_here = 0

   for i in range(0, size):
       max_ending_here = max_ending_here + a[i]
       if (max_so_far < max_ending_here):
           max_so_far = max_ending_here
       if max_ending_here < 0:
           max_ending_here = 0
   return max_so_far

a = [-13, -3, -25, -20, -3, -16, -23, -12, -5, -22, -15, -4, -7]
print("Maximum contiguous sum is", maxSubArraySum(a,len(a)))
```

```
Maximum contiguous sum is -3
```

RESULT : A program via which a user can find the maximum subarray has been created.

Experiment No: 2

DATE : 28/01/21

LANGUAGE USED : PYTHON

EXPERIMENT TITLE : Graph Coloring Problem

PROBLEM STATEMENT : Given a graph color its edges such that no two adjacent have the same color using minimum number of colors and return the Chromatic number.

ALGORITHM :

Initialize:

1. Color first vertex with first color.

Loop for remaining V-1 vertices.:

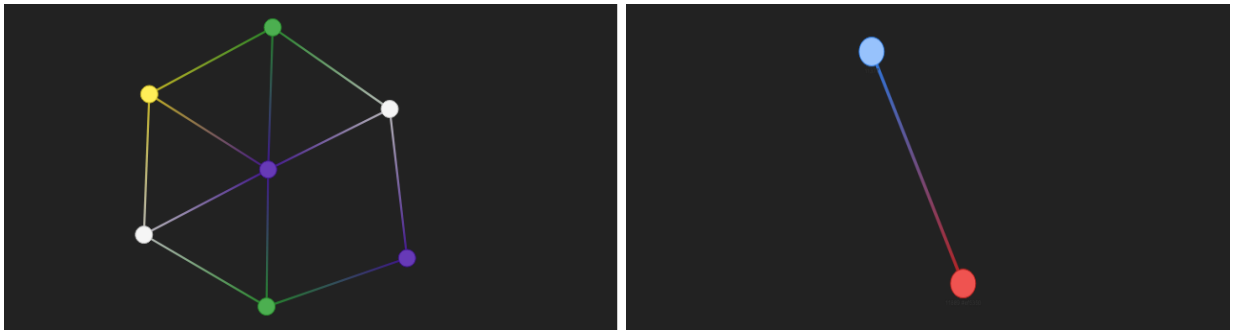
1. Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it.
2. If all previously used colors appear on vertices adjacent to v, assign a new color to it.
3. Repeat the following for all edges.
4. Index of color used is the chromatic number.

OPTIMIZATION TECHNIQUE:

Graph coloring problem is to assign colors to certain elements of a graph subject to certain constraints.

Vertex coloring is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using the same color. The other graph coloring problems like Edge Coloring (No vertex is incident to two edges of same color) and Face Coloring (Geographical Map Coloring) can be transformed into vertex coloring.

Chromatic Number: The smallest number of colors needed to color a graph G is called its chromatic number. For example, the following can be colored at least 2 colors.

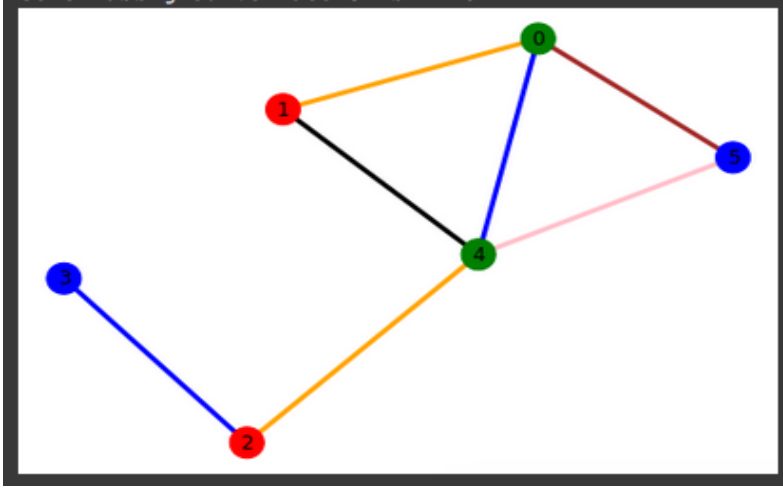


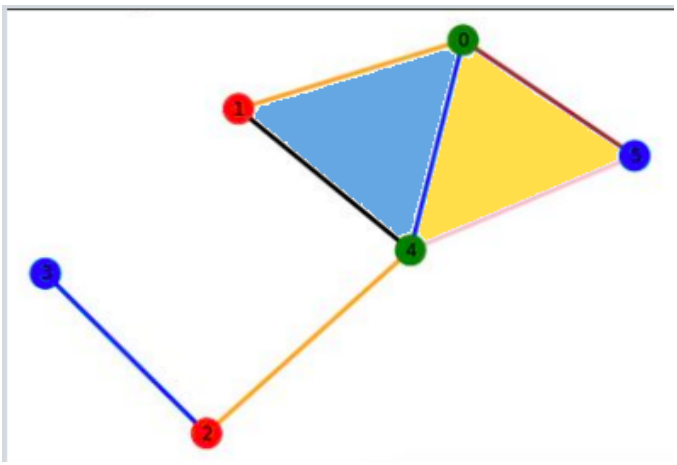
PROGRAM :

```
import networkx as nx
import matplotlib.pyplot as plt
from random import randint
 g = nx.Graph()

edges = [(0, 1), (0, 4), (0, 5), (4, 5), (1, 4), (1, 3), (2, 3), (2, 4)]
colors = ["BLUE", "GREEN", "RED", "YELLOW", "ORANGE", "PINK","BLACK",
"BROWN", "WHITE", "PURPLE"]

for edge in edges:
 g.add_edge(edge[0],edge[1])

g = edge_allocated(g, edges)

color = nx.coloring.greedy_color(g, strategy = "largest_first")

v_color = [colors[color[edge]] for edge in sorted(color)]
e_color = [colors[randint(0,9)] for edge in sorted(color)]
f_color = [colors[color[edge]+4] for edge in sorted(color)]
nx.draw(g, node_color = v_color, edge_color=e_color ,with_labels = True,
width=3, node_size=400)

for index,color in enumerate(v_color):
  print("Color assigned to vertex", index, "is", color)
print("*-------------------------------------*")
for index,color in enumerate(e_color):
 print("Color assigned to Edge", index, "is", color)
print("*-------------------------------------*")
for index,color in enumerate(f_color):
 print("Color assigned to Face", index, "is", color)
```

OUTPUT :

```
Color assigned to vertex 0 is GREEN
Color assigned to vertex 1 is RED
Color assigned to vertex 2 is GREEN
Color assigned to vertex 3 is BLUE
Color assigned to vertex 4 is BLUE
Color assigned to vertex 5 is RED
*-----------------------------------*
Color assigned to Edge 0 is ORANGE
Color assigned to Edge 1 is BLUE
Color assigned to Edge 2 is BROWN
Color assigned to Edge 3 is BLACK
Color assigned to Edge 4 is WHITE
Color assigned to Edge 5 is PINK
*-----------------------------------*
Color assigned to Face 0 is PINK
Color assigned to Face 1 is BLACK
Color assigned to Face 2 is PINK
Color assigned to Face 3 is ORANGE
Color assigned to Face 4 is ORANGE
Color assigned to Face 5 is BLACK
```





RESULT : A program via which a user can color edges of graph such that no two adjacent vertices have the same color using minimum number of colors and return the Chromatic number.

Experiment No: 3

DATE : 04/02/21

EXPERIMENT TITLE : CONSTRAINT SATISFACTION PROBLEMS

Question : AA + BB + CC = ABC

```
   A A
+  B B
+  C C
----------
 A B C
----------
```

- The digits are distinct and positive.
- Let's first focus on the value A, when we add three 2 digit numbers the most you get is in the 200's (ex: AA + BB + CC = ABC à 99 + 88 + 77 = 264).
- From this, we can tell that the largest value of A can be 2. So Either A = 1 or A = 2.
- Now focus on value B, let's take the unit digit of the given question: A + B +C = C (units).
- This can happen only if A + B = 0 (in the units) à A and B add up to 10.
- Two possibilities: 11 + 99 + CC = 19C à (1) or 22 + 88 + CC = 28C (2)
- Take equation (2), 110 + CC = 28C
- Focus on ten's place, 1 + C = 8, here C = 7. Then 22 + 88 + 77 = 187
- Thus, Equation (2) is not possible.
- From Equation (1), 11+99+CC = 19C à 110 + CC =19C à 1 + C = 9, then C = 8.
- 11 + 99 + 88 = 198. Hence, solved A = 1, B = 9 and C = 8

Question : TWO + TWO = FOUR

T W O

+T W O

FOUR

- F = 1 for carry over T >= 5.
- ‘O’ can’t be 0 as R will be 0. So IT can't be 5 so let T>=6
- If T = 6, O = 2 and R = 4 and W + W = U for W can’t be 1,2,6,4. W < 4 as to avoid carry over. WE can't be 3 as U will be 6.
- So T = 7, so, O can be 4 or 5 depending on whether or W + W > 10. If O is 4 then R = 8. WE can't be 1,2. So W = 3
- If W = 3 then U = 6 hence

Here is the answer:

7 3 4

\+ 7 3 4

---

1 4 6 8

Question : SEND + MORE = MONEY

```
          S  E  N  D
+         M  O  R  E
---------------------
       M  O  N  E  Y
---------------------
```

- From Column 5, M=1, since it is the only carry-over possible from the sum of 2 single digit numbers in column 4.
- To produce a carry from column 4 to column 5 'S + M' is at least 9 so 'S=8or9' so 'S+M=9or10' & so 'O = 0 or 1'. But 'M=1', so 'O = 0'.
- If there is carry from Column 3 to 4 then 'E=9' & so 'N=0'. But 'O = 0' so there is no carry & 'S=9' & 'c3=0'.
- If there is no carry from column 2 to 3 then 'E=N' which is impossible, therefore there is carry & 'N=E+1' & 'c2=1'.
- If there is carry from column 1 to 2 then 'N+R=E mod 10' & 'N=E+1' so 'E+1+R=E mod 10', so 'R=9' but 'S=9', so there must be carry from column 1 to 2. Therefore 'c1=1' & 'R=8'.
- To carry 'c1=1' from column 1 to 2, we must have 'D+E=10+Y' as Y cannot be 0/1 so D+E is at least 12. As D is at most 7 & E is atleast 5 (D cannot be 8 or 9 as it is already assigned). N is at most 7 & 'N=E+1' so 'E=5or6'.
- If E were 6 & D+E atleast 12 then D would be 7, but 'N=E+1' & N would also be 7 which is impossible. Therefore 'E=5' & 'N=6'. D+E is at least 12 so that we get 'D=7' & 'Y=2'.

SOLUTION:

```
   9  5  6  7
+  1  0  8  5
-----------------
1  0  6  5  2
-----------------
```

VALUES: S=9 E=5 N=6 D=7 M=1 O=0 R=8 Y=2

Question : BASE + BALL = GAMES

B A S E

+ B A L L

G A M E S

- Assuming numbers can't start with 0, G is 1 because two four-digit numbers can't sum to 20000 or more.
- SE+LL=ES or 1ES.
- If it is ES, then LL must be a multiple of 9 because SE and ES are always congruent mod 9. But LL is a multiple of 11, so it would have to be 99, which is impossible.
- So SE+LL=1ES. LL must be congruent to 100 mod 9. The only multiple of 11 that works is 55, so L is 5.
- SE+55=1ES. This is possible when E+5=S. The possibilities for ES are 27, 38, or 49.
- BA+BA+1=1AM. B must be at least 5 because B+B (possibly +1 from a carry) is at least 10.
- If A is less than 5, then A+A+1 does not carry, and A must be even. Inversely, if A is greater than 5, it must be odd. The possibilities for A are 0, 2, 4, 7, or 9.
    * 0 does not work because M would have to be 1.
    * 2 and 7 don't work because M would have to be 5.
    * 9 doesn't work because M would also have to be 9.
- So A is 4, M is 9, and B is 7. This leaves 38 as the only possibility for ES. The full equation is:

7483

+ 7455

14938

Question : N O + G U N + N O = H U N T

```
    N O
+ G U N
+   N O
  -------
  H U N T
  -------
```

- Here H = 1, from the NUNN column we must have "carry 1," so G = 9, U = zero.
- Since we have "carry" zero or 1 or 2 from the "ONOT" column, correspondingly we have N + U = 10 or 9 or 8. But duplication is not allowed, so N = 8 with "carry 2" from ONOT.
- Hence, O + O = T + 20 – 8 = T + 12. Testing for T = 2, 4 or 6, we find only T = 2 acceptable, O = 7.
- So we have,

```
    87
+  908
+   87
  -------
  1082
  -------
```

Question : CROSS + ROADS= DANGER

C R O S S

+ R O A D S

D A N G E R

- Since it is already mentioned that the carry value of resultant cannot be 0 then let's presume that the carry value of D is 1. We know that the sum of two similar values is even, hence R will have an even valueHence S+S= R So R is an even number for sure. So the value of R can b (0, 2, 4, 6, 8)
- Value of R cannot be 0 as two different values cannot be allotted the same digit,(if S=10 then their sum = 20 carry forward 2, then the value of R= 0) which is not possible.IF S= 1 :Not possible since D has the same value.IF S = 2
- Then R= 4 which is possible Hence S= 2 and R= 4C4+C+R= A+10C4+C+4= A+10C4+C>5 (Being the value of carry will be generated when the value of C is greater than 5
- C= 9+S+D = E +2+1= E
- Therefore E= 3+C+R= A+10C4+9+4= A+10Therefore A= 3 but it cannot be possible as E= 3.
- Now let's Consider S+D+C1= E2+1+0= 3. Therefore E= 3 making C2= 0 since 2+1=3.
- Now let's consider the equation again:C+R+C4= A+109+4+0= A+1013= A+10.
- Therefore A= 3 but E= 3.
- So, A is not equal to 3.
- Again considering A= 5 and S+D+C1= E3+1+0= E therefore E= 4 and C2= 0.
- Now considering the equation R+O = N + 0= N. So, 6+0+C3 <= t 3.
- Let C3= 1, then O< or equal to 2. That is O= 0, 1, 2.
- Let O =2. Again considering R+O+C3= N6+2+1= N. Hence, N= 9 but C= 9 so N cannot be equal to 9. Now let N= 8 and C3= 0. Let us consider equation G= 7. Hence, D= 1 S= 3 A= 5 G= 7 C= 9 O= 2 E= 4 R= 6 N= 8.

```
    9 6 2 3 3
+   6 2 5 1 3
---------------
  1 5 8 7 4 6
---------------
```

Program:

```
def solutions():
    # letters = ('s', 'e', 'n', 'd', 'm', 'o', 'r', 'y')
    all_solutions = list()
    for s in range(9, -1, -1):
        for e in range(9, -1, -1):
            for n in range(9, -1, -1):
                for d in range(9, -1, -1):
                    for m in range(9, 0, -1):
                        for o in range(9, -1, -1):
                            for r in range(9, -1, -1):
                                for y in range(9, -1, -1):
                                    if len(set([s, e, n, d, m, o, r, y])) == 8:
                                        send = 1000 * s + 100 * e + 10 * n + d
                                        more = 1000 * m + 100 * o + 10 * r + e
                                        money = 10000 * m + 1000 * o + 100 * n + 10 * e + y

                                        if send + more == money:
                                            all_solutions.append((send, more, money))
    return all_solutions

print(solutions())
```

Output:

```
SEND :  9567
MORE :  1085
MONEY :  10652
```

Result:

A program via which a user can solve constraint satisfaction problems has been created .

Experiment No: 4

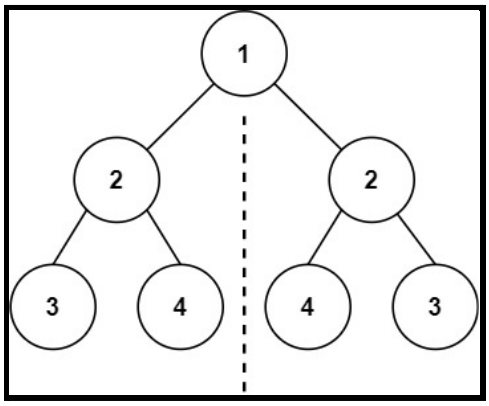
DATE : 14/02/21

LANGUAGE USED : PYTHON

EXPERIMENT TITLE : DFS and BFS

PROBLEM STATEMENT : Implementation and Analysis of DFS and BFS for an application

PROBLEM BFS: Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).



OPTIMIZATION TECHNIQUE :

- The set visits every node you put in queue true right after you put them in queue or else you will revisit some nodes a lot of time.
- You are putting a node in queue but not setting its visited to true and delaying this until you take that node out of queue then you set its visited to true.
- So what happens? Consider this Graph V={a,b,c,d} , E={a->b, a->c, a->d, b->d, c->d}. First a pushes b,c,d in the queue but doesn't set their visits.
- Then b pushes d to the queue, and after that c also pushes d to the queue. So your queue looks like this q={d,d,d}.
- So you are going to visit d three times. And this was a very small case, you can see for large cases how could this result in a big issue.

BINARY TREE DEFINITION :

```
# Definition for a binary tree node.
class TreeNode:
   def __init__(self, val=0, left=None, right=None):
       self.val = val
       self.left = left
       self.right = right
```

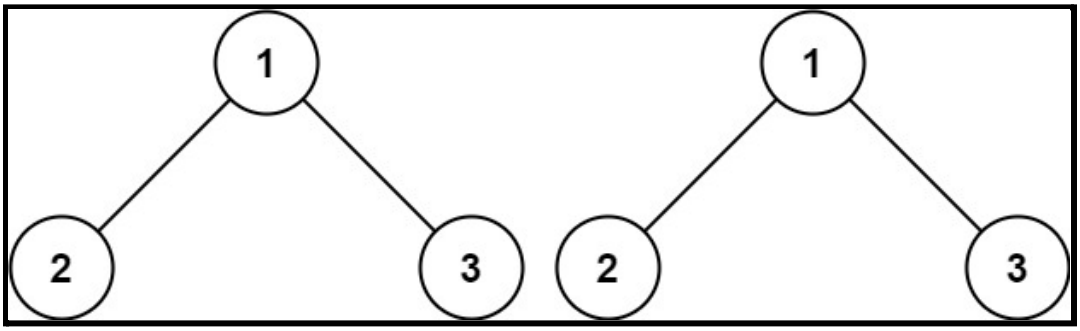
ALGORITHM :

```
def isSymmetric(self, root):
   if root is None:
       return True
   stack = [(root.left, root.right)]
   while stack:
       left, right = stack.pop()
       if left is None and right is None:
           continue
       if left is None or right is None:
           return False
       if left.val == right.val:
           stack.append((left.left, right.right))
           stack.append((left.right, right.left))
       else:
           return False
   return True
```

OUTPUT:

```
Root : [1, 2, 2, 3, 4, 4, 3]
True
```

PROBLEM DFS : Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).



OPTIMIZATION TECHNIQUE :

- Implement adjacency matrix when memory is not an issue, and it would be a requirement to see if a particular edge between nodes exists (not quite the case for BFS)
- Implement adjacency list to reduce memory usage and add elements quickly (ideal for BFS)
- Alternatively an incidence matrix could be used for unweighted or even weighed graphs if there's more memory so it could combine best of adjacency list and adjacency matrix
- Implement dynamic programming to store optimal solutions if required (ex: if a node has already been visited and the path so far has travelled less distance, why add it to the queue?)
- Implement ArrayDeque operation for 0-1 BFS graphs, because graphs with a weight of 0 on edges would be pushed to the front of the Queue as no distance happened. This would be more effective than a PriorityQueue or another implementation of a Queue on a weighted graph
- Implement an array for visited nodes to prevent visiting the same nodes repeatedly

BINARY TREE DEFINITION :

```
# Definition for a binary tree node.
class TreeNode:
   def __init__(self, val=0, left=None, right=None):
       self.val = val
       self.left = left
       self.right = right
```

ALGORITHM :

```
def isSameTree(self, p, q):
    if not p or not q: return p == q
    if p.val != q.val: return False

    left = self.isSameTree(p.left, q.left)
    right = self.isSameTree(p.right, q.right)
    return left and right
```

OUTPUT :

```
p = [1,2,3], q = [1,2,3]
True
```

RESULT :

A program via which a user can solve respective BFS and DFS problems..

Experiment No: 5

DATE : 28/02/21

LANGUAGE USED : PYTHON

EXPERIMENT TITLE : BFS AND A* SEARCHING ALGORITHM

PROBLEM STATEMENT : Developing Best first search and A* Algorithm for real world problems.

ALGORITHM:

1. Best First Search:

a. Take an input of the maze in binary format.

b. Taking the starting point, find all adjacent paths that can be taken.

c. Keep traversing through the array while taking the adjacent cells closest to the destination while avoiding cells with value 0.

d. If the final point is reached, save the length of the path.

e. Compare lengths of all paths that reach the destination and print the length of the shortest path.

2. A* Search Algorithm:

a. Take an input of the maze in binary format.

b. Taking the starting point, find all adjacent and diagonal paths that can be taken.

c. Finding the closest cell to the most optimised path to reach the destination, pick a cell to move ahead.

d. Check for adjacent turning paths to cut diagonals and save 1 step.

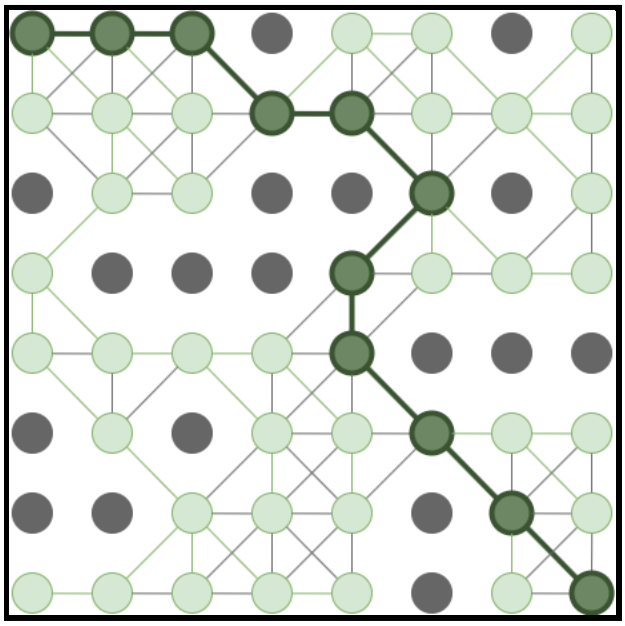
e. Compare lengths of all paths that reach the destination and print the length of the shortest path.

OPTIMIZATION TECHNIQUE :

An A* search is like a breadth-first search, except that in each iteration, instead of expanding the cell with the shortest path from the origin, we expand the cell with the lowest overall estimated path length.

This is the distance so far, plus a heuristic (rule-of-thumb) estimate of the remaining distance. As long as the heuristic is consistent, an A* graph-search will find the shortest path.

This can be somewhat more efficient than breadth-first-search as we typically don't have to visit nearly as many cells. Intuitively, an A* search has an approximate sense of direction, and uses this sense to guide it towards the target.



We perform an A* search to find the shortest path, then return it's length, if there is one. Note: I chose to deal with the special case, that the starting cell is a blocking cell, here rather than complicate the search implementation.

This implementation is somewhat general and will work for other constant-cost search problems, as long as you provide a suitable goal function, successor function, and heuristic.

BFS CODE :

```
from collections import deque
ROW = 9
COL = 10

class Point:
    def __init__(self,x: int, y: int):
        self.x = x
        self.y = y

class queueNode:
    def __init__(self,pt: Point, dist: int):
        self.pt = pt
        self.dist = dist

def isValid(row: int, col: int):
    return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)

rowNum = [-1, 0, 0, 1]
colNum = [0, -1, 1, 0]
def BFS(mat, src: Point, dest: Point):
    if mat[src.x][src.y]!=1 or mat[dest.x][dest.y]!=1:
        return -1
    visited = [[False for i in range(COL)] for j in range(ROW)]
    visited[src.x][src.y] = True
    q = deque()
    s = queueNode(src,0)
    q.append(s)
    while q:
        curr = q.popleft()
        pt = curr.pt
        if pt.x == dest.x and pt.y == dest.y:
            return curr.dist
        for i in range(4):
            row = pt.x + rowNum[i]
            col = pt.y + colNum[i]
            if (isValid(row,col) and
            mat[row][col] == 1 and
                not visited[row][col]):
                visited[row][col] = True
                Adjcell = queueNode(Point(row,col), curr.dist+1)
                q.append(Adjcell)
```

```
    return -1
def main():
    mat = [[ 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 ],
        [ 1, 0, 1, 0, 1, 1, 1, 0, 1, 1 ],
        [ 1, 1, 1, 0, 1, 1, 0, 1, 0, 1 ],
        [ 0, 0, 0, 0, 1, 0, 0, 0, 0, 1 ],
        [ 1, 1, 1, 0, 1, 1, 1, 0, 1, 0 ],
        [ 1, 0, 1, 1, 1, 1, 0, 1, 0, 0 ],
        [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 1 ],
        [ 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 ],
        [ 1, 1, 0, 0, 0, 0, 1, 0, 0, 1 ]]
    source = Point(0,0)
    dest = Point(3,4)

    dist = BFS(mat,source,dest)

    if dist!=-1:
        print("Shortest Path is",dist)
    else:
        print("Shortest Path doesn't exist")
main()
```

OUTPUT :

```
Shortest Path is 11
```

A* CODE :

```
def a_star_graph_search(
          start,
          goal_function,
          successor_function,
          Heuristic ):
   visited = set()
   came_from = dict()
   distance = {start: 0}
   frontier = PriorityQueue()
   frontier.add(start)
   while frontier:
       node = frontier.pop()
       if node in visited:
           continue
       if goal_function(node):
           return reconstruct_path(came_from, start, node)
       visited.add(node)
       for successor in successor_function(node):
           frontier.add(
               successor,
               priority = distance[node] + 1 + heuristic(successor)
           )
           if (successor not in distance
               or distance[node] + 1 < distance[successor]):
               distance[successor] = distance[node] + 1
               came_from[successor] = node
   return None

def reconstruct_path(came_from, start, end):
   reverse_path = [end]
   while end != start:
       end = came_from[end]
       reverse_path.append(end)
   return list(reversed(reverse_path))
```

RESULT :

Developed best first search and A* Algorithm for real world problems.

Experiment No: 6

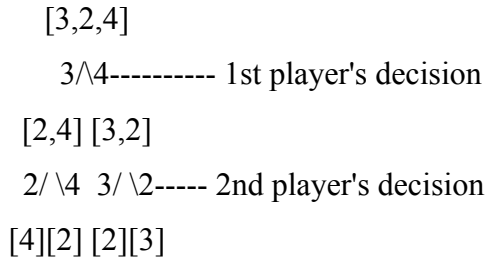
DATE : 04/03/21
LANGUAGE USED : PYTHON
EXPERIMENT TITLE : MIN-MAX ALGORITHM

PROBLEM STATEMENT : Developing min max algorithm for real world problem.

PROBLEM : Predict the Winner

Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by player 2 and then player 1 and so on. Each time a player picks a number, that number will not be available for the next player. This continues until all the scores have been chosen. The player with the maximum score wins. Given an array of scores, predict whether player 1 is the winner. You can assume each player plays to maximize his score.

ALGORITHM :



- Currently 1st with choosable i, j,

  1.if 1st picks nums[i], 2nd can pick either ends of nums[i + 1, j]

  a.if 2nd picks nums[i + 1], 1st can pick either ends of nums[i + 2, j]

  b.if 2nd picks nums[j], 1st can pick either ends of nums[i + 1, j - 1]

  since 2nd plays to maximize his score, 1st can get nums[i] + min(1.a, 1.b)

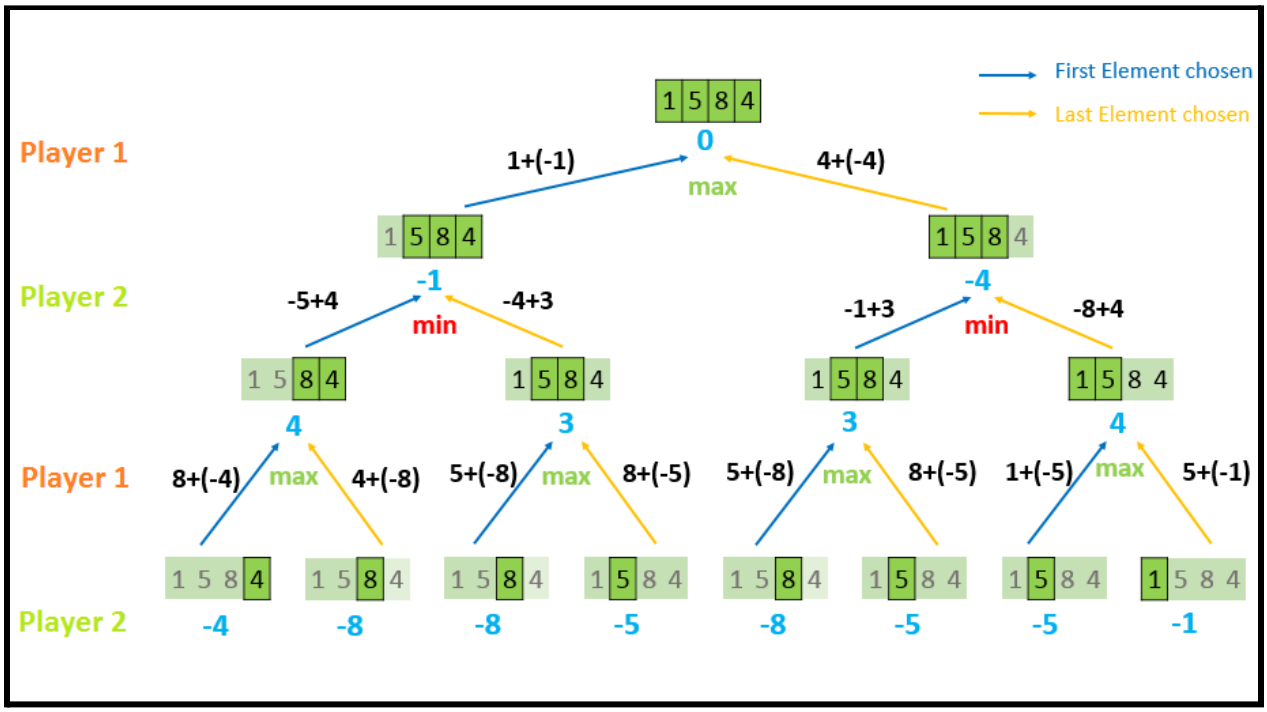
  2.if 1st picks nums[j], 2nd can pick either ends of nums[i, j - 1]

  a.if 2nd picks nums[i], 1st can pick either ends of nums[i + 1, j - 1];

  b.if 2nd picks nums[j - 1], 1st can pick either ends of nums[i, j - 2];

  since 2nd plays to maximize his score, 1st can get nums[j] + min(2.a, 2.b)

OPTIMIZATION TECHNIQUE :



Optimize for a sample test case :

- Input: [1, 5, 233, 7] : Output: True
- Explanation: Player 1 first chooses 1.
- Then player 2 has to choose between 5 and 7.
- No matter which number player 2 choose, player 1 can choose 233.
- Finally, player 1 has more score (234) than player 2 (12), so you need to return True representing player 1 can win.
- Since the 1st plays to maximize his score, 1st can get max(nums[i] + min(1.a, 1.b), nums[j] + min(2.a, 2.b));

CODE :

```
class Solution:
   def PredictTheWinner(self, nums: List[int]) -> bool:
       # minMax -- recursion with memoization
       n = len(nums)
       if n == 1 or n % 2 == 0:
           return True

       self.dp = [0 for _ in range(n * n)]
       return self.checkWin(nums, 0, n-1) >= 0

   def checkWin(self, nums, left, right):
       if left >= right:
           return nums[left]

       k = left * len(nums) + right
       if self.dp[k] > 0:
           return self.dp[k]

       self.dp[k] = max(nums[left] - self.checkWin(nums, left + 1, right), nums[right]
- self.checkWin(nums, left, right - 1))
       return self.dp[k]
```

OUTPUT :

```
nums : [1, 5, 2, 3, 7, 233, 8, 9, 12, 100]
Player 1 is the winner
```

RESULT :

Developed a min-max Algorithm for real world problems.

Experiment No: 7

DATE : 12/04/21

LANGUAGE USED : PYTHON

EXPERIMENT TITLE : Implementation of Unification and Resolution

PROBLEM STATEMENT : Developing an optimized technique using an appropriate artificial intelligence algorithm to solve the Unification and Resolution.

ALGORITHM :

function PL-RESOLUTION (KB, @) returns true or false inputs: KB,

the knowledge base, group of sentences/facts in propositional logic

@, the query, a sentence in propositional logic

clauses → the set of clauses in the CNF representation of KB ^ @

new → { }

loop do

for each Ci, Cj in clauses do

resolvents → PL-RESOLVE (Ci, Cj)

if resolvents contains the empty clause the return true

new → new union resolvents

if new is a subset of clauses then return false

clauses → clauses union true

OPTIMIZATION TECHNIQUE:

Resolution basically works by using the principle of proof by contradiction. To find the conclusion we should negate the conclusion. Then the resolution rule is applied to the resulting clauses. Each clause that contains complementary literals is resolved to produce a2.

new clause, which can be added to the set of facts (if it is not already present). This process continues until one of the two things happen:•There are no new clauses that can be added•An application of the resolution rule derives the empty clauseAn empty clause shows that the

negation of the conclusion is a complete contradiction,hence the negation of the conclusion is invalid or false or the assertion is completely valid or true.

1. Convert the given statements in Predicate/Propositional Logic
2. Convert these statements into Conjunctive Normal Form
3. Negate the Conclusion (Proof by Contradiction)
4. Resolve using a Resolution Tree (Unification)

CODE :

Unification

```
def get_index_comma(string):
    """
    Return index of commas in string
    """

    index_list = list()
    # Count open parentheses
    par_count = 0

    for i in range(len(string)):
        if string[i] == ',' and par_count == 0:
            index_list.append(i)
        elif string[i] == '(':
            par_count += 1
        elif string[i] == ')':
            par_count -= 1

    return index_list


def is_variable(expr):
    """
    Check if expression is variable
    """

    for i in expr:
        if i == '(' or i == ')':
            return False
```

```
    return True


def process_expression(expr):
    """
    input:  - expression:
            'Q(a, g(x, b), f(y))'
    return: - predicate symbol:
            Q
            - list of arguments
            ['a', 'g(x, b)', 'f(y)']
    """

    # Remove space in expression
    expr = expr.replace(' ', '')

    # Find the first index == '('
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break

    # Return predicate symbol and remove predicate symbol in expression
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')

    # Remove '(' in the first index and ')' in the last index
    expr = expr[1:len(expr) - 1]

    # List of arguments
    arg_list = list()

    # Split string with commas, return list of arguments
    indices = get_index_comma(expr)

    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
```

```
        arg_list.append(expr[indices[len(indices) - 1] + 1:])

    return predicate_symbol, arg_list


def get_arg_list(expr):
    """
    input:  expression:
            'Q(a, g(x, b), f(y))'
    return: full list of arguments:
            ['a', 'x', 'b', 'y']
    """

    _, arg_list = process_expression(expr)

    flag = True
    while flag:
        flag = False

        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)
                for j in tmp:
                    if j not in arg_list:
                        arg_list.append(j)
                arg_list.remove(i)

    return arg_list


def check_occurs(var, expr):
    """
    Check if var occurs in expr
    """

    arg_list = get_arg_list(expr)
    if var in arg_list:
        return True

    return False
```

```
def unify(expr1, expr2):

    # Step 1:
    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False
        else:
            tmp = str(expr1) + '/' + str(expr2)
            return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)

        # Step 2
        if predicate_symbol_1 != predicate_symbol_2:
            return False
        # Step 3
        elif len(arg_list_1) != len(arg_list_2):
            return False
        else:
            # Step 4: Create substitution list
            sub_list = list()

            # Step 5:
            for i in range(len(arg_list_1)):
                tmp = unify(arg_list_1[i], arg_list_2[i])

                if not tmp:
                    return False
                elif tmp == 'Null':
```

```
                pass
            else:
                if type(tmp) == list:
                    for j in tmp:
                        sub_list.append(j)
                else:
                    sub_list.append(tmp)

        # Step 6
        return sub_list


if __name__ == '__main__':

    f1 = 'Q(a, g(x, a), f(y))'
    f2 = 'Q(a, g(f(b), a), x)'

    result = unify(f1, f2)
    if not result:
        print('Unification failed!')
    else:
        print('Unification successful!')
        print(result)
```

Resolution

```
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_  = ['Friends(Alice,Bob,Charlie,Diana)',
'Friends(Diana,Charlie,Bob,Alice)']
```

```
inputSentences_ = ['Friends(a,b,c,d)', 'NotFriends(a,b,c,d)']
knowledgeBase = KB(inputSentences_)
knowledgeBase.prepareKB()
results_ = knowledgeBase.askQueries(inputQueries_)
printOutput("output.txt", results_)
```

INPUT :

3

Avoid(Alice,Bob)

AtRisk(Che)

Avoid(Bob,Alice)

8

Take(x,Zn) => AtRisk(x)

Take(y,Ox) => ~AtRisk(y)

AtRisk(y) => Avoid(y,x)

Avoid(x,y) => Avoid(y,x)

Take(Alice,x) => ~Take(Bob,x)

Take(Che,Ox) => Take(Alice,Ox)

~Take(Che,Ox)

Take(Alice,Zn)

OUTPUT :

Unification:

```
Vaishnavimoorthy:~/environment/RA1811028010040 $ python3 unification.py
Unification successful!
['f(b)/x', 'f(y)/x']
```

Resolution:

```
Vaishnavimoorthy:~/environment/RA1811028010040 $ python3 resolution.py
['TRUE', 'FALSE', 'TRUE']
Vaishnavimoorthy:~/environment/RA1811028010040 $
```

RESULT : Developed Unification and Resolution Algorithm in Python for solving logical problems.

Experiment No: 10

DATE : 11/03/21

LANGUAGE USED : PYTHON

EXPERIMENT TITLE : Implementation of block world problem

PROBLEM STATEMENT : Developing an optimized technique using an appropriate artificial intelligence algorithm to solve the block world problem.

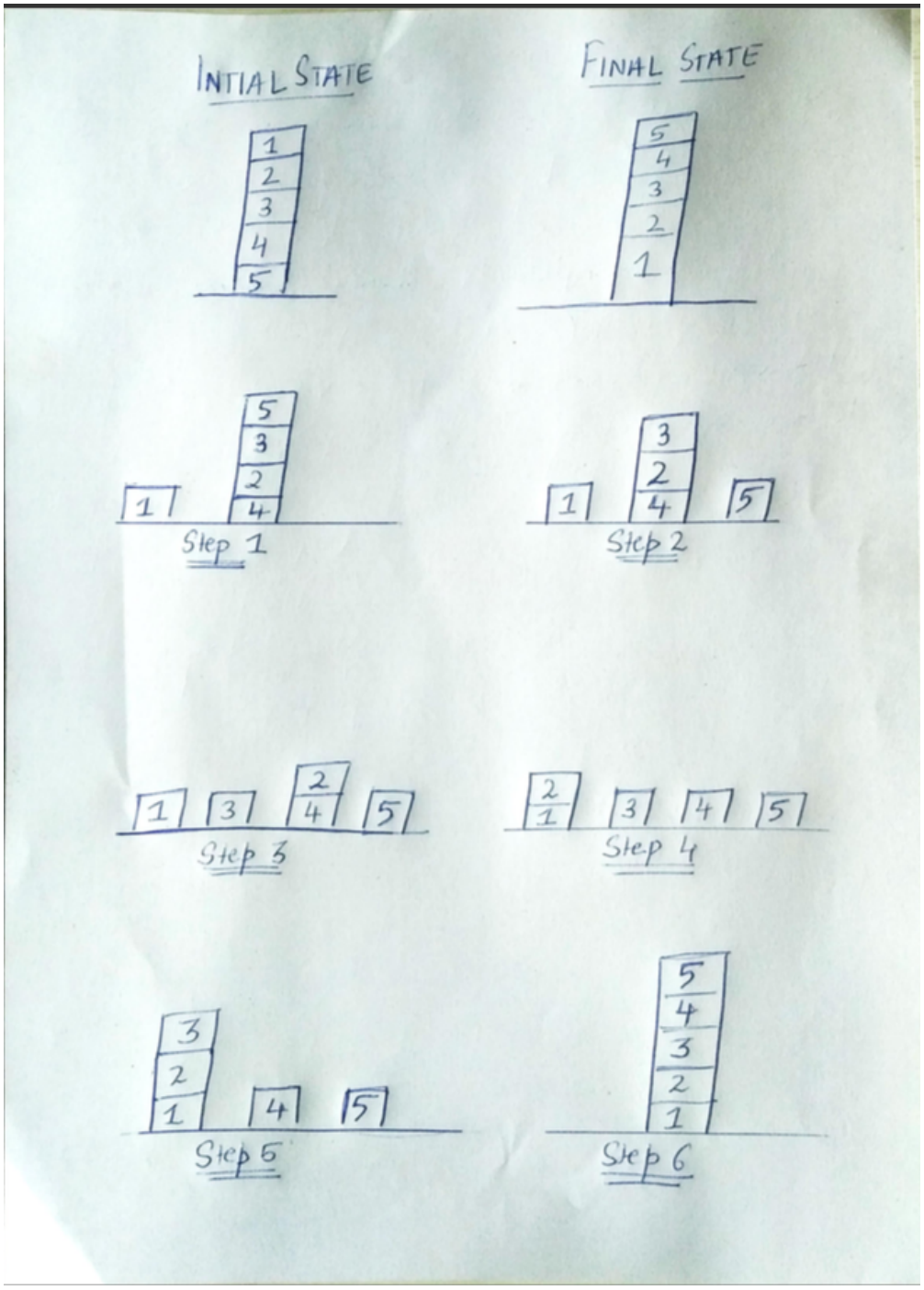
ALGORITHM :

1. Initialise a stack to store the blocks.
2. Make sure the stack is empty when HEAD NODE.NEXT = NULL
3. Read the pattern of blocks given label it START STATE
4. Compare the given pattern to the given final pattern label it GOAL STATE
5. Now start the movement of the blocks one by on either one on top or to the floor according to the need.
6. Keep recording these movements in the empty stack created by push and pop methods.
7. Stop the block manipulation when goal state is reached.

OPTIMIZATION TECHNIQUE :

- Here keeping track of movement of the block is the main problem.
- We keep traversing the floor again and again after each move, our time complexity will be O(n2) which is exponentially higher than what is needed and should be avoided.
- To solve this problem a stack data structure can be used, so whenever a movement is made the movement can be conveniently stored in the stack which will be initialized as empty by assigning it the value NULL.
- When the block is to be added to the sequence of blocks simply use the push method to make the movement.
- When a block is supposed to be removed from the pattern of blocks, the pop method can be used to make that movement.

- Implementing this will bring down the time complexity from O(n) and worst case of O(n2) to O(1) that is unit time which is a major optimization from exponential time.



CODE :

```
class Node:
   def __init__(self, val=None):
       self.data = val
       self.next = None

class Stack:

   def __init__(self):
       print("Stack created")
       self.stack_pointer = None

   def push(self, x):
       if not isinstance(x, Node):
           x = Node(x)
       print(f"Adding {x.data} to the top of stack")
       if self.is_empty():
           self.stack_pointer = x
       else:
           x.next = self.stack_pointer
           self.stack_pointer = x

   def pop(self):

       if not self.is_empty():
           print(f"Removing node on top of stack")
           curr = self.stack_pointer
           self.stack_pointer = self.stack_pointer.next
           curr.next = None
           return curr.data
       else:
           return "Stack is empty"

   def is_empty(self):
       return self.stack_pointer == None

   def peek(self):

       if not self.is_empty():
           return self.stack_pointer.data
```

```
   def __str__(self):
       print("Printing Stack state...")
       to_print = ""
       curr = self.stack_pointer
       while curr is not None:
           to_print += str(curr.data) + "->"
           curr = curr.next
       if to_print:
           print("Stack Pointer")
           print(" |")
           print(" V")
           return "[" + to_print[:-2] + "]"
       return "[]"

print ("INITIAL STATE : {[1], [2], [3], [4], [5]}")
print("-"*70)
print ("FINAL STATE :[4->3->2->1]")
my_stack = Stack()
print("Checking if stack is empty:", my_stack.is_empty())
my_stack.push(1)

my_stack.push(2)
print(my_stack)

my_stack.push(3)

my_stack.push(4)

print("Checking item on top of stack:", my_stack.peek())

my_stack.push(5)
print(my_stack)

print(my_stack.pop())

print(my_stack.pop())
print(my_stack)

my_stack.push(4)
print(my_stack)
```

OUTPUT

```
Vaishnavimoorthy:~/environment/RA1811028010040 $ python3 blockworld.py
INITIAL STATE : {[1], [2], [3], [4], [5]}
----------------------------------------------------------------
FINAL STATE :[4->3->2->1]
Stack created
Checking if stack is empty: True
Adding 1 to the top of stack
Adding 2 to the top of stack
Printing Stack state...
Stack Pointer
 |
 V
[2->1]
Adding 3 to the top of stack
Adding 4 to the top of stack
Checking item on top of stack: 4
Adding 5 to the top of stack
Printing Stack state...
Stack Pointer
 |
 V
[5->4->3->2->1]
Removing node on top of stack
5
Removing node on top of stack
4
Printing Stack state...
Stack Pointer
 |
 V
[3->2->1]
Adding 4 to the top of stack
Printing Stack state...
Stack Pointer
 |
 V
[4->3->2->1]
Vaishnavimoorthy:~/environment/RA1811028010040 $
```

RESULT : Block world problem successfully implemented using optimal artificial intelligence techniques under time complexity of O(n2).

Experiment No: 8

DATE : 21/04/21

LANGUAGE USED : PYTHON

EXPERIMENT TITLE : Implementation of Rule Based Inference problem

PROBLEM STATEMENT : Developing an optimized technique using an appropriate artificial intelligence algorithm to detect the animal.

ALGORITHM :

In artificial intelligence, we need intelligent computers which can create new logic from old logic or by evidence, so generating the conclusions from evidence and facts is termed as Inference.

Inference rules are the templates for generating valid arguments. Inference rules are applied to derive proofs in artificial intelligence, and the proof is a sequence of the conclusion that leads to the desired goal.

In inference rules, the implication among all the connectives plays an important role. Following are some terminologies related to inference rules:

- Implication: It is one of the logical connectives which can be represented as $P \rightarrow Q$. It is a Boolean expression.
- Converse: The converse of implication, which means the right-hand side proposition goes to the left-hand side and vice-versa. It can be written as $Q \rightarrow P$.
- Contrapositive: The negation of converse is termed as contrapositive, and it can be represented as $\neg Q \rightarrow \neg P$.
- Inverse: The negation of implication is called inverse. It can be represented as $\neg P \rightarrow \neg Q$.

Identification of animal:

```
cheetah :- mammal,
           carnivore,
           verify(has_tawny_color),
           verify(has_dark_spots).
tiger :- mammal,
         carnivore,
         verify(has_tawny_color),
         verify(has_black_stripes).
giraffe :- ungulate,
           verify(has_long_neck),
           verify(has_long_legs).
zebra :- ungulate,
         verify(has_black_stripes).
```

Classification rules:

```
mammal    :- verify(has_hair), !.
mammal    :- verify(gives_milk).
bird      :- verify(has_feathers), !.
bird      :- verify(flys),
             verify(lays_eggs).
carnivore :- verify(eats_meat), !.
carnivore :- verify(has_pointed_teeth),
             verify(has_claws),
             verify(has_forward_eyes).
ungulate :- mammal,
            verify(has_hooves), !.
ungulate :- mammal,
            verify(chews_cud).
```

OPTIMIZATION TECHNIQUE :

Rule-based systems provide the computational mechanisms found in most expert systems. Knowledge is specified via facts and IF–THEN rules, and modus ponens is used as the underlying inference method to derive new conclusions from existing knowledge. These production rules in many cases allow a straightforward encoding of expertise about a particular domain, often as a situation–action pairs where the IF part of the rule specifies aspects of a situation leading to one or more actions as described in the THEN part. In principle, the rules and

facts in a rule-based system can be translated into equivalent logical sentences. A combination of restrictions and additional constructs in their language, together with the tight integration between language and evaluation mechanism, offers substantial practical advantages for rule-based systems

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and **logical connectives**.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called **tautology**, and it is also called a valid sentence.
- A proposition formula which is always false is called **Contradiction**.
- A proposition formula which has both true and false values is called
- Statements which are questions, commands, or opinions are not propositions such as "**Where is Rohini**", "**How are you**", "**What is your name**", are not propositions.

CODE :

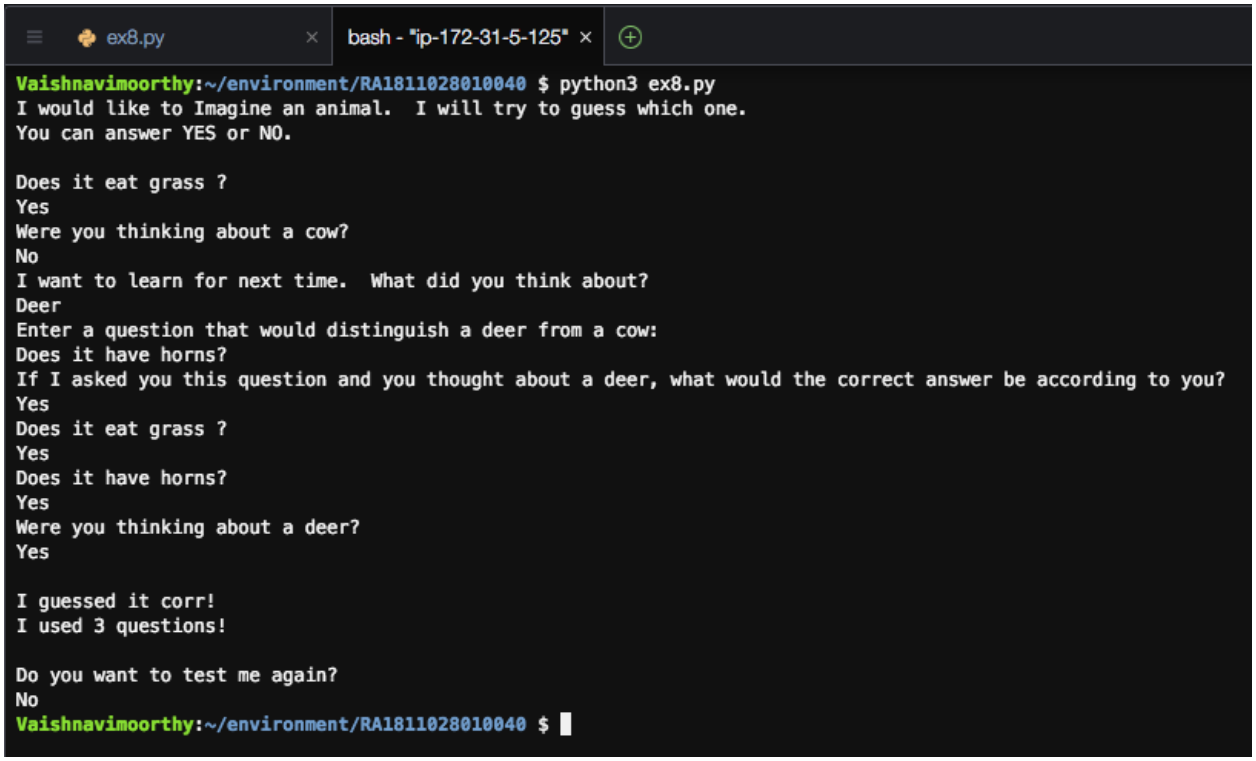
```
import sys

def definiteNoun(s):
 s = s.lower().strip()
 if s in ['a', 'e', 'i', 'o', 'u', 'y']:
   return "an " + s
 else:
   return "a " + s

def removeArticle(s):
 "Remove the definite article 'a' or 'an' from a noun."
 s = s.lower().strip()
 if s[0:3] == "an ": return s[3:]
```

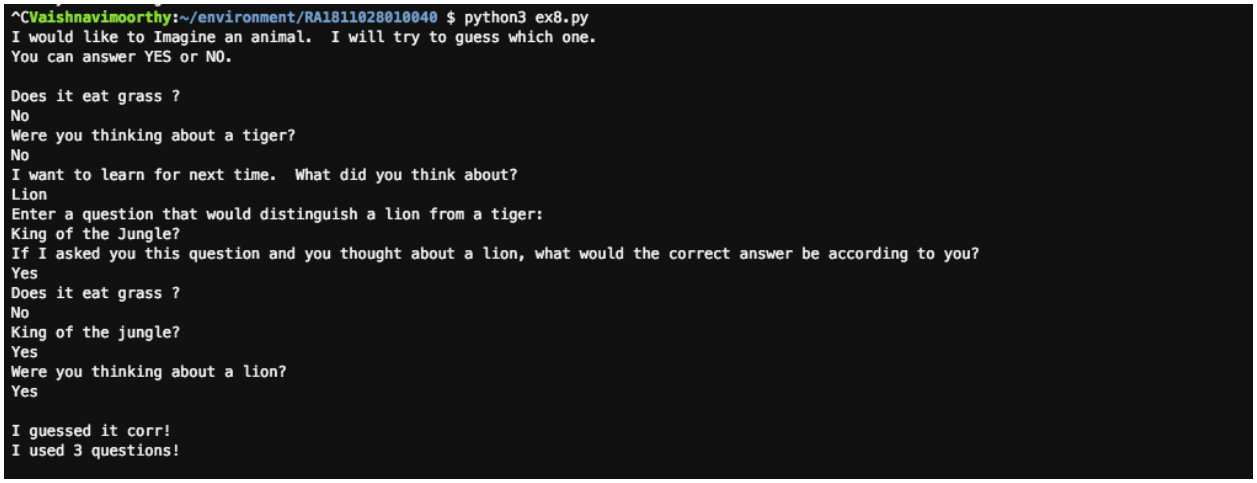
```
 if s[0:2] == "a ": return s[2:]
 return s

def makeQuestion(question, yes, no):
 return [question, yes, no]

def isQuestion(p):
 "Check if node is a question (with answers), or a plain answer."
 return type(p).__name__ == "list"

def askQuestion(question):
 print ("\r%s " % question,)
 return sys.stdin.readline().strip().lower()

def getAnswer(question):
 if isQuestion(question):
   return askQuestion(question[0])
 else:
   return askQuestion("Were you thinking about %s?" % definiteNoun(question))

def answeredYes(answer):
 if len(answer) > 0:
   return answer.lower()[0] == "y"
 return False

def gameOver(message):
 global tries
 print ("")
 print ("\r%s" % message)
 print ("")

def playAgain():
 return answeredYes(askQuestion("Do you want to play again?"))

def correctGuess(message):
 global tries
 gameOver(message)

 if playAgain():
   print ("")
   tries = 0
   return Q
```

```
 else:
   sys.exit(0)

def nextQuestion(question, answer):
 global tries
 tries += 1

 if isQuestion(question):
   if answer:
     return question[1]
   else:
     return question[2]
 else:
   if answer:
     return correctGuess("I knew it!")
   else:
     return makeNewQuestion(question)

def replaceAnswer(tree, find, replace):
 if not isQuestion(tree):
   if tree == find:
     return replace
   else:
     return tree
 else:
   return makeQuestion(tree[0],
     replaceAnswer(tree[1], find, replace),
     replaceAnswer(tree[2], find, replace))

def makeNewQuestion(wrongAnimal):
 global Q, tries

 correctAnimal = removeArticle(askQuestion("I give up.  What did you think about?"))

 newQuestion = askQuestion("Enter a question that would distinguish %s from %s:"
     % (definiteNoun(correctAnimal), definiteNoun(wrongAnimal))).capitalize()

 yesAnswer = answeredYes(askQuestion("If I asked you this question " +
       "and you thought about %s, what would the correct answer be?" %
definiteNoun(correctAnimal)))

 # Create new question node
```

```
 if yesAnswer:
   q = makeQuestion(newQuestion, correctAnimal, wrongAnimal)
 else:
   q = makeQuestion(newQuestion, wrongAnimal, correctAnimal)
 Q = replaceAnswer(Q, wrongAnimal, q)
 tries = 0
 return Q

def addNewQuestion(wrongAnimal, newques, correct):
   global Q
   q = makeQuestion(newques, correct, wrongAnimal)
   Q = replaceAnswer(Q, wrongAnimal, q)
   return Q

tries = 0
Q = (makeQuestion('Does it eat Grass?', 'Cow', 'Tiger'))
q = addNewQuestion('Tiger', 'Does it have dark spots?', 'Leopard')
q = addNewQuestion('Leopard', 'Is it the fastest animal?', 'Cheetah')
q = addNewQuestion('Penguin', 'Can it fly?', 'Parrot')
q = Q

print ("Imagine an animal.  I will try to guess which one.")
print ("You are only allowed to answer YES or NO.")
print ("")

try:
 while True:
   ans = answeredYes(getAnswer(q))
   q = nextQuestion(q, ans)
except KeyboardInterrupt:
 sys.exit(0)
except Exception:
 sys.exit(1)
```

OUTPUT :





RESULT : Animal Detection problem successfully implemented using optimal artificial intelligence techniques under time complexity of O(n2).