**Ex: 1.C** **CRYPT-ARITHMETIC PROBLEM**

**Aim:** To implement the Crypt-Arithmetic Problem using python

**PROCEDURE:**

A crypto-arithmetic problem is a mathematical puzzle where numbers are represented by letters.

The Crypt-Arithmetic problem in Artificial Intelligence is a type of encryption problem in which the written message in an alphabetical form which is easily readable and understandable is converted into a numeric form which is neither easily readable nor understandable. In simpler words, the crypt-arithmetic problem deals with the converting of the message from the readable plain text to the non-readable ciphertext. The constraints which this problem follows during the conversion is as follows:

1. A number 0-9 is assigned to a particular alphabet.
2. Each different alphabet has a unique number.
3. All the same, alphabets have the same numbers.
4. The numbers should satisfy all the operations that any normal number does.

Let us take an example of the message: SEND MORE MONEY.

Here, to convert it into numeric form, we first split each word separately and represent it as follows:

This means that:

- S corresponds to the digit 9
- E corresponds to the digit 5
- N corresponds to the digit 6
- D corresponds to the digit 7
- M corresponds to the digit 1
- O corresponds to the digit 0
- R corresponds to the digit 8
- Y corresponds to the digit 2

This program solves the classic "SEND+MORE=MONEY" crypto-arithmetic problem. You can replace the problem variable with any other crypto-arithmetic problem to solve it. The program uses a brute-force approach by trying all possible permutations of digits. If a solution is found, it prints the mapping of letters to digits. If no solution is found, it prints "No solution found."

**PROGRAM**

```
from itertools import permutations

def solve_crypto_arithmetic(problem):
    # Split the problem into three parts: two operands and one result
    operands, result = problem.split('=')
    operand1, operand2 = operands.split('+')

    # Generate all possible permutations of digits (0-9)
    for digits in permutations('0123456789', len(set(operand1 + operand2 + result))):
        # Create a dictionary to map letters to digits
        digit_map = {letter: digit for letter, digit in zip(set(operand1 + operand2 + result),
digits)}

        # Check if the mapping is valid (no leading zeros)
        if any(digit_map[letter] == '0' for letter in (operand1[0], operand2[0], result[0])):
            continue

        # Convert operands and result to integers using the digit map
        num1 = int(''.join(digit_map[letter] for letter in operand1))
        num2 = int(''.join(digit_map[letter] for letter in operand2))
        num_result = int(''.join(digit_map[letter] for letter in result))

        # Check if the equation holds true
        if num1 + num2 == num_result:
            return digit_map

    return None

# Example usage:
problem = "SEND+MORE=MONEY"
solution = solve_crypto_arithmetic(problem)
if solution:
    print("Solution found:")
    for letter, digit in solution.items():
        print(f"{letter} = {digit}")
else:
   print("No solution found.")
```

**OUTPUT:**

The output of the above program will be the mapping of letters to digits that satisfies the crypto-arithmetic problem. For the example problem "SEND+MORE=MONEY", the output will be:

```
Solution found:
S = 9
E = 5
N = 6
```

D = 7
M = 1
O = 0
R = 8
Y = 2

And indeed, when you replace the letters with these digits, the equation holds true:

9537 + 1082 = 10619

Note that there may be other solutions to the problem, but this program will only find one of them.

**Result :** Thus the Crypt-Arithmetic Problem is solved by Python code

**Ex: 1.b**

## 8-QUEENS PROBLEM

**PROGRAM**

```
def solve_n_queens(n):
    def is_safe(board, row, col):
        for i in range(row):
            if board[i] == col or \
                board[i] - i == col - row or \
                board[i] + i == col + row:
                return False
        return True

    def place_queens(n, row, board):
        if row == n:
            result.append(board[:])
            return
        for col in range(n):
            if is_safe(board, row, col):
                board[row] = col
                place_queens(n, row + 1, board)

    result = []
    place_queens(n, 0, [-1]*n)
    return [["."*i + "Q" + "."*(n-i-1) for i in sol] for sol in result]

# Test the function
n = 8
solutions = solve_n_queens(n)
for i, solution in enumerate(solutions):
    print(f"Solution {i+1}:")
    for row in solution:
        print(row)
```

```
print()
```

**Ex: 2** **A * ALGORITHM**

**Aim:** To implement the A* algorithm using python.

**Algorithm**

The A* (A-star) algorithm is a popular pathfinding algorithm used to find the shortest path between two points in a weighted graph or network.

1. Add the starting square (or node) to the open list.
2. Repeat the following:

   A) Look for the lowest F cost square on the open list. We refer to this as the currentsquare.

   B). Switch it to the closed list.

   C) For each of the 8 squares adjacent to this current square …

   - If it is not walk able or if it is on the closed list, ignore it. Otherwise do the following.
   - If it isn't on the open list, add it to the open list. Make the current square the parent of this square. Record the F, G, and H costs of the square.
   - If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your open list sorted by F score, you may need to resort the list to account for the change.

   D) Stop when you:

   - Add the target square to the closed list, in which case the path has been found, or
   - Fail to find the target square, and the open list is empty. In this case, there isno path.
3. Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is your path.

**PROGRAM**

```
import heapq
```

```python
def a_star(graph, start, goal):
    # Define the heuristic function (Manhattan distance)
    def heuristic(node):
        return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

    # Initialize the open list (priority queue)
    open_list = [(0, start)]
    heapq.heapify(open_list)

    # Initialize the came_from dictionary
    came_from = {start: None}

    # Initialize the cost_so_far dictionary
    cost_so_far = {start: 0}

    while open_list:
        # Get the node with the lowest f score (heuristic + cost)
        current = heapq.heappop(open_list)[1]

        # Check if we've reached the goal
        if current == goal:
            break

        # Explore neighbors
        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            x, y = current[0] + dx, current[1] + dy
            neighbor = (x, y)

            # Check if the neighbor is within bounds and not an obstacle
            if 0 <= x < len(graph) and 0 <= y < len(graph[0]) and graph[x][y] != 1:
                new_cost = cost_so_far[current] + 1
                if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                    cost_so_far[neighbor] = new_cost
                    priority = new_cost + heuristic(neighbor)
                    heapq.heappush(open_list, (priority, neighbor))
                    came_from[neighbor] = current

    # Reconstruct the path
    current = goal
    path = []
    while current != start:
        path.append(current)
        current = came_from[current]
    path.append(start)
    path.reverse()

    return path

# Example usage:
```

```
graph = [
    [0, 0, 0, 0, 0],
    [0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0]
]
start = (0, 0)
goal = (4, 4)
path = a_star(graph, start, goal)
print(path)
```

**OUTPUT**

(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]

This path represents the shortest route from the start position (0, 0) to the goal position (4, 4) in the graph, avoiding obstacles (represented by 1s in the graph).

Here's a visual representation of the path:

```
S -> (0, 0)
  |
  |
  v
(1, 0) -> (2, 0) -> (3, 0) -> (4, 0)
                   |
                   |
                   v
                   (4, 1) -> (4, 2) -> (4, 3) -> G (4, 4)
```

Note that the actual output may vary depending on the specific graph, start, and goal positions used.

**Result :** Thus the A* algorithm is implemented by Python code