11. Finding the number of ways a ball can move out of a grid in exactly

Aim

To write a Python program that finds the number of ways a ball can move out of a grid in exactly n steps

Algorithm

1. Input values: grid size m × n, number of steps N, starting cell (i, j).
2. The ball can move in 4 directions: up, down, left, right.
3. If the ball goes outside the grid, count as 1 way.
4. If steps are finished but ball is still inside, count as 0 ways.
5. Otherwise, try moving in all 4 directions and add the results.
6. Use memoization to store already computed results.
7. Print the total number of ways

OUTPUT:

main.py

Share Run

Output

```python
def findPaths(m, n, N, i, j):
    directions = [(1,0), (-1,0), (0,1), (0,-1)]
    memo = {}
    def dfs(x, y, steps):

        if x < 0 or x >= m or y < 0 or y >= n:
            return 1
        if steps == 0:
            return 0
        if (x, y, steps) in memo:
            return memo[(x, y, steps)]
        ways = 0
        for dx, dy in directions:
            ways += dfs(x+dx, y+dy, steps-1)
        memo[(x, y, steps)] = ways
        return ways
    return dfs(i, j, N)
print(findPaths(2, 2, 2, 0, 0))
print(findPaths(1, 3, 3, 0, 1))

```

```
6
12

=== Code Execution Successful =
```

Result:

Hence, the program correctly finds the number of ways the ball can move out of the grid in exactly N steps

12.House Robber II problem

Aim

To write a Python program that finds the maximum money a robber can rob from houses arranged in a **circle**, without robbing two adjacent houses (which would alert the police).

Algorithm

1. Input: An array nums[] where each element represents money in a house.
2. Since houses are in a circle, the first house and last house are adjacent.
3. Therefore:
    - Case 1: Rob houses from index 0 to n-2 (ignore last house).
    - Case 2: Rob houses from index 1 to n-1 (ignore first house).
4. Use the House Robber I (linear**)** approach:
    - For each case, maintain two variables:
        - prev1 = max money till previous house.
        - prev2 = max money till house before previous.
    - At each house: curr = max(prev1, prev2 + nums[i]).
5. Answer = maximum of Case 1 and Case 2.
6. Print result

OUTPUT :

```python
def rob_linear(nums):
    prev1, prev2 = 0, 0
    for num in nums:
        curr = max(prev1, prev2 + num)
        prev2, prev1 = prev1, curr
    return prev1
def rob(nums):
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]
    return max(rob_linear(nums[:-1]), rob_linear(nums[1:]))
nums1 = [2, 3, 2]
nums2 = [1, 2, 3, 1]
print("Output 1: The maximum money you can rob without alerting the police is", rob(nums1))
print("Output 2: The maximum money you can rob without alerting the police is", rob(nums2))
```

```
Output 1: The maximum money you can rob without alerting the police is 3
Output 2: The maximum money you can rob without alerting the police is 4

=== Code Execution Successful ===
```

Result :

Hence, the program correctly calculates the maximum amount that can be robbed without alerting the police

13 . Climbing Stairs problem

Aim

To write a Python program that finds the number of distinct ways to climb to the top of a staircase of n steps, where you can climb either 1 step or 2 steps at a time.

Algorithm

1. Input: integer n (number of steps).
2. If n = 1, only 1 way.
3. If n = 2, only 2 ways (1+1, or 2).
4. For n > 2:
   - The number of ways to reach step n = ways(n-1) + ways(n-2)
   - Because the last step can be reached either from n-1 (taking 1 step) or from n-2 (taking 2 steps).
5. Implement using iteration (efficient)

Output :

main.py | Share | Run | Output

```
def climbStairs(n):
    if n == 1:
        return 1
    if n == 2:
        return 2
    prev1, prev2 = 2, 1
    for i in range(3, n+1):
        curr = prev1 + prev2
        prev2, prev1 = prev1, curr
    return prev1
print("Output 1:", climbStairs(4))
print("Output 2:", climbStairs(3))
```

```
Output 1: 5
Output 2: 3

=== Code Execution Successful ===
```

Result :

Hence, the program correctly finds the number of distinct ways to climb the staircase

14. Unique Paths problem

Aim

To write a Python program that finds the number of unique paths a robot can take from the top-left corner to the bottom-right corner of an m × n grid, moving only right or down.

Algorithm

1. Input: integers m (rows), n (columns).
2. The robot starts at (0,0) and needs to reach (m-1,n-1).
3. At each cell (i,j), the robot can move either:
    - Down → (i+1, j)
    - Right → (i, j+1)
4. The number of paths to reach (i,j) is:
5. paths(i,j) = paths(i-1,j) + paths(i,j-1)

(sum of ways from top and left).

6. Base condition:
    - First row & first column have only 1 way.
7. Fill a DP table of size m × n using the above rule.
8. Return dp[m-1][n-1] as the result

Output :

```
def uniquePaths(m, n):
    dp = [[1]*n for _ in range(m)]
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    return dp[m-1][n-1]
print("Output 1:", uniquePaths(7, 3))
print("Output 2:", uniquePaths(3, 2))
```

```
Output 1: 28
Output 2: 3

=== Code Execution Successful ===
```

Result :

Hence, the program correctly calculates the total number of unique paths

15 . Large Group Positions problem

Aim

To write a Python program that finds all large groups (length ≥ 3) in a string and returns their start and end indices.

Algorithm

1. Input: a string s.
2. Initialize start = 0.
3. Traverse the string with index i.
4. If s[i] != s[i-1], that means the group ended at i-1.
    - Check if (i-1 - start + 1) >= 3.
    - If yes, append [start, i-1] to result.
    - Update start = i.
5. After the loop ends, check the last group.
6. Return result list.

Output :

main.py

Share Run

```
def largeGroupPositions(s):
    result = []
    start = 0
    for i in range(1, len(s)+1):
        if i == len(s) or s[i] != s[i-1]:
            if i - start >= 3:
                result.append([start, i-1])
            start = i
    return result
print("Output 1:", largeGroupPositions("abbxxxxzzy"))
print("Output 2:", largeGroupPositions("abc"))
print("Output 3:", largeGroupPositions("abbxxxxyyyzz"))
```

Output

```
Output 1: [[3, 6]]
Output 2: []
Output 3: [[3, 6], [7, 9]]

=== Code Execution Successful ===
```

Result :

Program correctly finds intervals of large groups .

16. Conway's Game of Life problem

Aim

To write a Python program that computes the next state of Conway's Game of Life given the current board configuration.

Algorithm

1. Input: 2D grid board (size m × n).
2. For each cell (i, j):
   - Count live neighbors (check all 8 directions).
   - Apply rules to decide its next state.
3. Since updates must happen simultaneously:
   - Either create a copy board and fill next state.
   - Or update in place with special markers (2 = was alive, now dead, -1 = was dead, now alive).
4. Return the updated board.

Output :

main.py

Share Run

```python
def gameOfLife(board):
    m, n = len(board), len(board[0])
    directions = [(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
    new_board = [[0]*n for _ in range(m)]
    for i in range(m):
        for j in range(n):
            live_neighbors = 0
            for dx, dy in directions:
                ni, nj = i+dx, j+dy
                if 0 <= ni < m and 0 <= nj < n and board[ni][nj] == 1:
                    live_neighbors += 1
            if board[i][j] == 1:
                if live_neighbors == 2 or live_neighbors == 3:
                    new_board[i][j] = 1
            else:
                if live_neighbors == 3:
                    new_board[i][j] = 1

    return new_board
print("Output 1:", gameOfLife([[0,1,0],[0,0,1],[1,1,1],[0,0,0]]))
print("Output 2:", gameOfLife([[1,1],[1,0]]))
```

Output

```
Output 1: [[0, 0, 0], [1, 0, 1], [0, 1, 1], [0, 1, 0]]
Output 2: [[1, 1], [1, 1]]

=== Code Execution Successful ===
```

Result :

The program correctly simulates the next generation in Conway's Game of Life .

17. Champagne Tower problem

Aim

To write a Python program that calculates how full a specific glass is in a champagne tower after pouring a given amount of champagne.

Algorithm

1. Input:
   - poured = total cups of champagne poured at the top.
   - query_row, query_glass = indices of the glass to check.
2. Create a 2D array dp where dp[i][j] = amount of champagne in row i and glass j.
3. Pour all champagne into the top glass: dp[0][0] = poured.
4. For each glass dp[i][j]:
   - If it has more than 1 cup, the excess (dp[i][j] - 1) is split equally to the two glasses below:
     - dp[i+1][j] += excess / 2
     - dp[i+1][j+1] += excess / 2
5. After simulating rows up to query_row, the amount in the target glass is:
   - min(1, dp[query_row][query_glass]) (since each glass can hold at most 1 cup)

Output :

```python
def champagneTower(poured, query_row, query_glass):
    dp = [[0.0]*101 for _ in range(101)]
    dp[0][0] = poured

    for i in range(query_row + 1):
        for j in range(i + 1):
            if dp[i][j] > 1:
                excess = dp[i][j] - 1
                dp[i][j] = 1
                dp[i+1][j] += excess / 2
                dp[i+1][j+1] += excess / 2

    return dp[query_row][query_glass]
print("Output 1:", champagneTower(1, 1, 1))
print("Output 2:", champagneTower(2, 1, 1))
print("Output 3:", champagneTower(4, 2, 1))
```

```
Output 1: 0.0
Output 2: 0.5
Output 3: 0.5

=== Code Execution Successful ===
```

Result :

The program correctly simulates the flow of champagne and computes how full any glass is .

18. List Examples in Python

Aim:

To write a Python program to demonstrate and manipulate lists including empty lists, lists with one element, lists with identical elements, and lists containing negative numbers, and display their contents.

Algorithm:

1. Start
2. Define an empty list and print it.
3. Define a list with one element and print it.
4. Define a list with all identical elements and print it.
5. Define a list with negative numbers. Sort the list in ascending order using sort() method.
6. Print the sorted list of negative numbers.
7. End

Output :

```
empty_list = []
print("Empty list:", empty_list)
single_element_list = [1]
print("Single element list:", single_element_list)
identical_list = [7, 7, 7, 7]
print("List with identical elements:", identical_list)
negative_list = [-5, -1, -3, -2, -4]
negative_list.sort()
print("List with negative numbers (sorted):", negative_list)
```

STDIN

Input for the program (Optional)

Output:

```
Empty list: []
Single element list: [1]
List with identical elements: [7, 7, 7, 7]
List with negative numbers (sorted): [-5, -4, -3,
```

Result :

Thus ,the program of List example of python was successfully implemented.

19. Selection Sort algorithm

Aim :

To write a program to sort an array using the Selection Sort algorithm and test it with different cases (random array, reverse-sorted array, already sorted array).

Algorithm:

1. Start.
2. Input an array of n elements.
3. Repeat for each position i = 0 to n-1:
   - Assume the smallest element is at index i (set min_index = i).
   - For each element from index i+1 to n-1:
     - If a smaller element is found, update min_index.
   - Swap the element at index i with the element at min_index.
4. Continue until the array is fully sorted.
5. Display the sorted array.
6. Stop

Output :

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
arr1 = [5, 2, 9, 1, 5, 6]
sorted_arr1 = selection_sort(arr1.copy())
print("Sorted Random Array:", sorted_arr1)
arr2 = [10, 8, 6, 4, 2]
sorted_arr2 = selection_sort(arr2.copy())
print("Sorted Reverse Array:", sorted_arr2)
arr3 = [1, 2, 3, 4, 5]
sorted_arr3 = selection_sort(arr3.copy())
print("Sorted Already Sorted Array:", sorted_arr3)
```

STDIN

Input for the program (Optional)

Output:

```
Sorted Random Array: [1, 2, 5, 5, 6, 9]
Sorted Reverse Array: [2, 4, 6, 8, 10]
Sorted Already Sorted Array: [1, 2, 3, 4, 5]
```

Result :

The program was successfully executed.Selection Sort was implemented, and the arrays were sorted correctly

20. Bubble Sort with early stopping

Aim :

To write a program for Bubble Sort with early stopping (optimized version) that terminates if the list becomes sorted before completing all passes.

Algorithm

1. Start.
2. Input the list/array to be sorted.
3. Set n = length of the array.
4. Repeat for i = 0 to n-1:
   - Set swapped = False.
   - For j = 0 to n-i-2:
     - If arr[j] > arr[j+1], swap them.
     - Set swapped = True.
   - If swapped == False, break the loop (array already sorted).
5. Print the sorted array.
6. Stop

Output :

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr
nums = [5, 1, 4, 2, 8]
print("Before Sorting:", nums)
sorted_nums = bubble_sort(nums)
print("After Sorting:", sorted_nums)
```

STDIN

Input for the program (Optional)

Output:

```
Before Sorting: [5, 1, 4, 2, 8]
After Sorting: [1, 2, 4, 5, 8]
```

Result :

The given list was sorted using the optimized Bubble Sort algorithm, which stops early if the list becomes sorted before all passes .