1.12 MAXIMUM AMOUNT OF MONEY A ROBBER CAN ROB

Question:

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have security systems connected, and it will automatically contact the police if two adjacent houses were broken into on the same night.

AIM:

To find the maximum amount of money a robber can rob from houses arranged in a circle, such that no two adjacent houses are robbed.

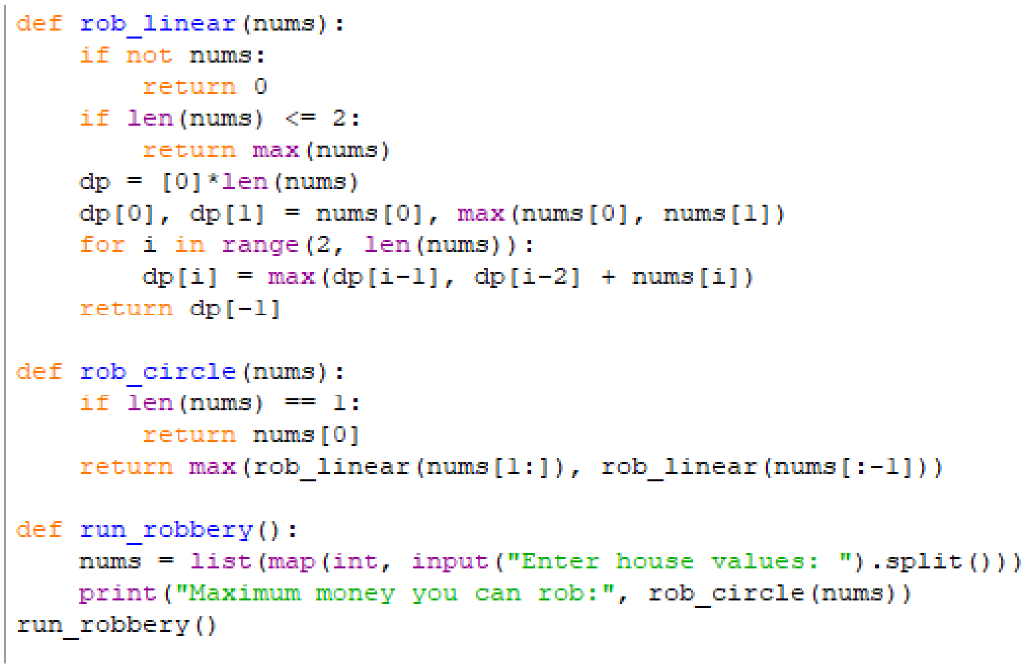
ALGORITHM:

1. Observation:
    - Because houses are in a circle, the first and last house cannot both be robbed.
    - So, break the problem into two cases:
    - Case 1: Rob houses from 0 to n-2 (exclude last house).
    - Case 2: Rob houses from 1 to n-1 (exclude first house).
    - The answer = max(Case 1, Case 2).
2. Subproblem (House Robber I):
    - For a linear arrangement of houses, use dynamic programming:
    - dp[i] = max(dp[i-1], dp[i-2] + nums[i])
    - Base cases:
    - dp[0] = nums[0]
    - dp[1] = max(nums[0], nums[1])

3. Handle edge cases:

   - If nums is empty → return 0.
   - If nums has only 1 element → return nums[0].

PROGRAM:

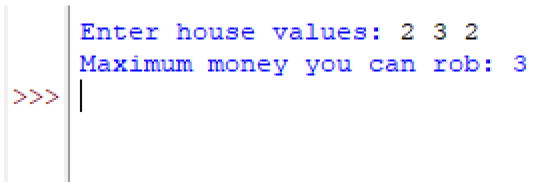


```python
def rob_linear(nums):
    if not nums:
        return 0
    if len(nums) <= 2:
        return max(nums)
    dp = [0]*len(nums)
    dp[0], dp[1] = nums[0], max(nums[0], nums[1])
    for i in range(2, len(nums)):
        dp[i] = max(dp[i-1], dp[i-2] + nums[i])
    return dp[-1]

def rob_circle(nums):
    if len(nums) == 1:
        return nums[0]
    return max(rob_linear(nums[1:]), rob_linear(nums[:-1]))

def run_robbery():
    nums = list(map(int, input("Enter house values: ").split()))
    print("Maximum money you can rob:", rob_circle(nums))
run_robbery()
```

Input:

nums = [1, 2, 3, 1]

Output:



```
Enter house values: 2 3 2
Maximum money you can rob: 3
>>> |
```

RESULT:

Thus the program is successfully executed, and the output is verified.

PERFORMANCE ANALYSIS:

- Time Complexity: O(n) (one DP pass for each case).
- Space Complexity: O(1) (if optimized to use just two variables).