TOPIC 1

1.Given an array of strings words, return the first palindromic string in the array. If there is no such string, return an empty string "". A string is palindromic if it reads the same forward and backward.

AIM: To write a program to find the **first palindromic string** in a given array of strings.
If no palindromic string is found, return an empty string.

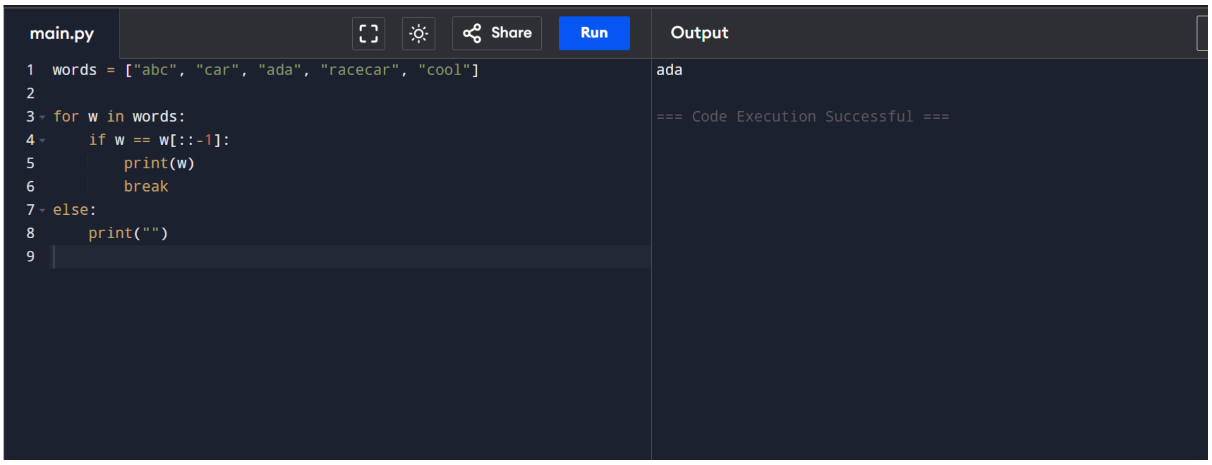
PROCEDURE:

- Start the program.
- Read the list of strings.
- Reverse each string and compare it with the original.
- Print the first string that is a palindrome.
- If no palindrome is found, print an empty string and stop.

PROGRAM:

```
words = ["abc", "car", "ada", "racecar", "cool"]

for w in words:

  if w == w[::-1]:

    print(w)

    break

else:

  print("")
```

OUTPUT:



```
words = ["abc", "car", "ada", "racecar", "cool"]

for w in words:
    if w == w[::-1]:
        print(w)
        break
else:
    print("")
```

```
ada

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

RESULT: The program successfully finds and displays the **first palindromic string** from the given list of strings.

2.You are given two integer arrays nums1 and nums2 of sizes n and m, respectively.

Calculate the following values: answer1 : the number of indices i such that nums1[i]

exists in nums2. answer2 : the number of indices i such that nums2[i] exists in

nums1 Return [answer1,answer2].

AIM: To write a program to count how many elements of one array exist in the other array and return the result as a list.

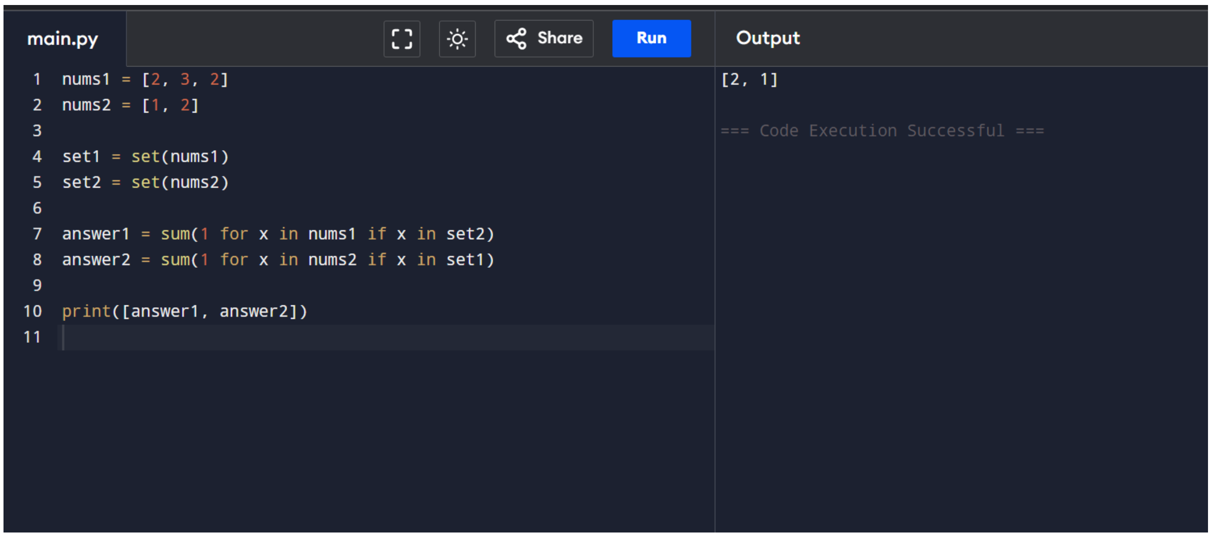
PROCEDURE:

- Read two integer arrays nums1 and nums2.
- Convert both arrays into sets for fast searching.
- Count elements in nums1 that exist in nums2.
- Count elements in nums2 that exist in nums1.
- Display the result as [answer1, answer2].

PROGRAM:

```
nums1 = [2, 3, 2]

nums2 = [1, 2]

set1 = set(nums1)

set2 = set(nums2)

answer1 = sum(1 for x in nums1 if x in set2)

answer2 = sum(1 for x in nums2 if x in set1)

print([answer1, answer2])
```

OUTPUT:



```
nums1 = [2, 3, 2]
nums2 = [1, 2]

set1 = set(nums1)
set2 = set(nums2)

answer1 = sum(1 for x in nums1 if x in set2)
answer2 = sum(1 for x in nums2 if x in set1)

print([answer1, answer2])
```

```
[2, 1]

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

RESULT: The program successfully counts the matching elements between the two arrays.

3.You are given a 0-indexed integer array nums. The distinct count of a subarray of nums is defined as: Let nums[i..j] be a subarray of nums consisting of all the indices from i to j such that 0 <= i <= j < nums.length. Then the number of distinct values in nums[i..j] is called the distinct count of nums[i..j]. Return the sum of the squares of distinct counts of all subarrays of nums. A subarray is a contiguous non-empty sequence of elements within an array.

AIM: To write a program that calculates the **sum of the squares of distinct counts** of all possible non-empty subarrays of a given integer array.

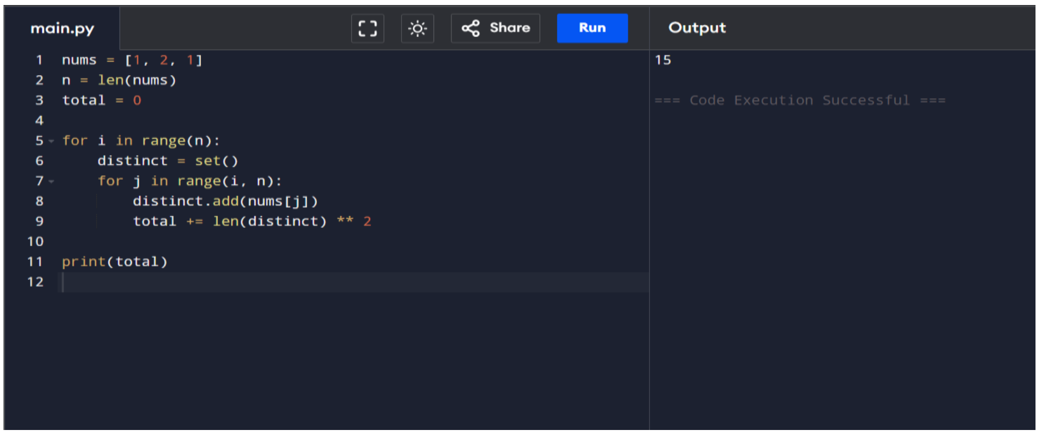
PROCEDURE:

- Read the integer array nums.
- Generate all possible subarrays of the array.
- For each subarray, find the number of distinct elements using a set.
- Square the distinct count and add it to a total sum.
- Display the final sum.

PROGRAM:

```
nums = [1, 2, 1]

n = len(nums)

total = 0

for i in range(n):

   distinct = set()

   for j in range(i, n):

      distinct.add(nums[j])

      total += len(distinct) ** 2

print(total)
```

OUTPUT:



```
nums = [1, 2, 1]
n = len(nums)
total = 0

for i in range(n):
    distinct = set()
    for j in range(i, n):
        distinct.add(nums[j])
        total += len(distinct) ** 2

print(total)
```

```
15

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

RESULT: The program successfully computes the sum of squares of distinct counts of all subarrays.

4. Given a 0-indexed integer array nums of length n and an integer k, return the number of pairs (i, j) where 0 <= i < j < n, such that nums[i] == nums[j] and (i * j) is divisible by k.

AIM: To write a program that counts the number of index pairs (i, j)such that $0 \leq i < j < n$, nums[i] == nums[j], and $(i \times j)$is divisible by k.

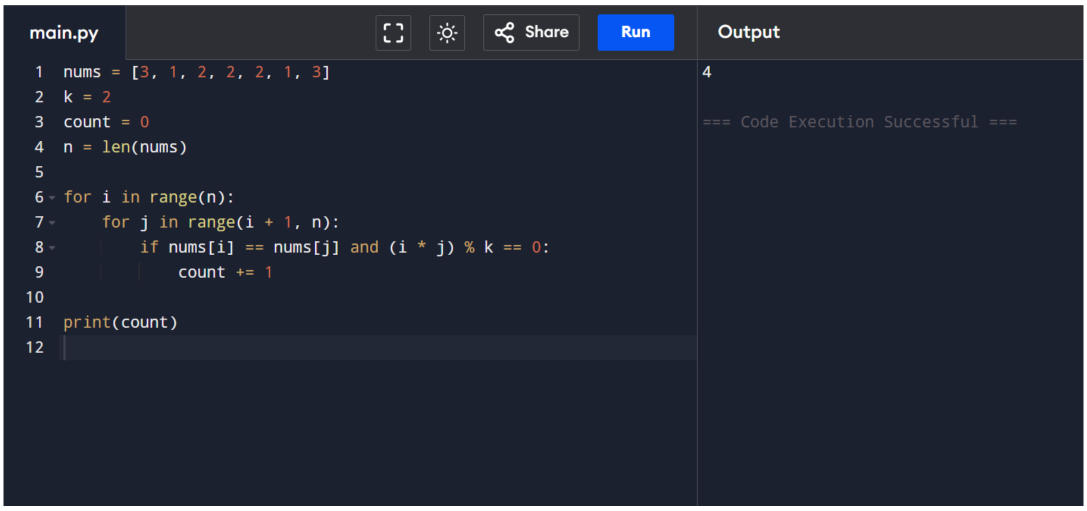
PROCEDURE:

- Read the integer array nums and the value k.
- Use two nested loops to check all index pairs $(i, j)$where $i < j$.
- Check whether the elements at indices i and j are equal.
- Check whether $(i * j)$is divisible by k.
- Count and display the total number of such valid pairs.

PROGRAM:

```
nums = [3, 1, 2, 2, 2, 1, 3]

k = 2

count = 0

n = len(nums)

for i in range(n):

  for j in range(i + 1, n):

    if nums[i] == nums[j] and (i * j) % k == 0:

      count += 1

print(count)
```

OUTPUT:



```
nums = [3, 1, 2, 2, 2, 1, 3]
k = 2
count = 0
n = len(nums)

for i in range(n):
    for j in range(i + 1, n):
        if nums[i] == nums[j] and (i * j) % k == 0:
            count += 1

print(count)
```

```
4

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

RESULT: The program successfully finds all valid index pairs that satisfy the given conditions.

5. Write a program FOR THE BELOW TEST CASES with least time complexity

Test Cases: -

Input: {1, 2, 3, 4, 5} Expected Output: 5

Input: {7, 7, 7, 7, 7} Expected Output: 7

Input: {-10, 2, 3, -4, 5} Expected Output: 5

AIM: To write a program that finds the **maximum element** in a given array using the **least time complexity**.

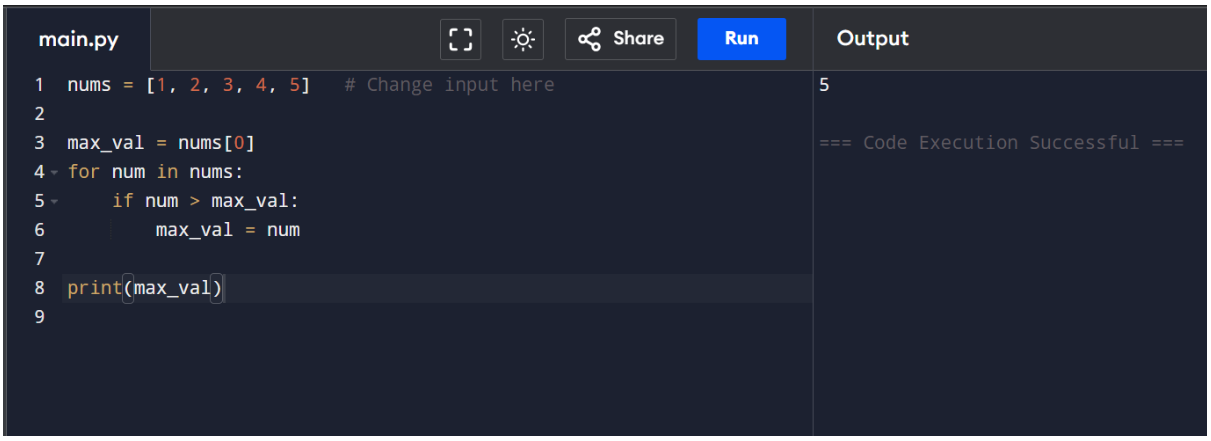
PROCEDURE:

- Read the input array.
- Initialize a variable with the first element of the array.
- Traverse the array once and compare each element with the maximum value.
- Update the maximum value when a larger element is found.
- Display the maximum element.

PROGRAM:

```
nums = [1, 2, 3, 4, 5]   # Change input here

max_val = nums[0]

for num in nums:

   if num > max_val:

       max_val = num

print(max_val)
```

OUTPUT:



```
nums = [1, 2, 3, 4, 5]   # Change input here

max_val = nums[0]
for num in nums:
    if num > max_val:
        max_val = num

print(max_val)
```

```
5

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

RESULT: The program correctly finds the **maximum element** from the given array using minimum time complexity **O(n)**.

6.You have an algorithm that process a list of numbers. It firsts sorts the list

using an efficient sorting algorithm and then finds the maximum element in

sorted list. Write the code for the same.

AIM: To write a program that **sorts a list using an efficient sorting algorithm** and then **finds the maximum element** from the sorted list.

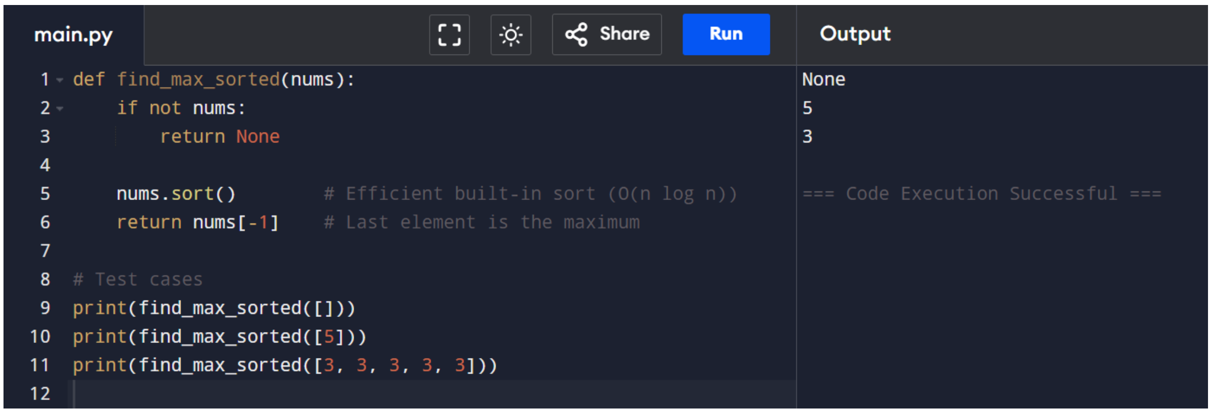
PROCEDURE:

- Read the input list of numbers.
- Check if the list is empty; if yes, display an appropriate message.
- Sort the list using an efficient sorting method.
- Find the maximum element from the sorted list.
- Display the result.

PROGRAM:

```
def find_max_sorted(nums):

   if not nums:

      return None

 nums.sort()

   return nums[-1]

# Test cases

print(find_max_sorted([]))

print(find_max_sorted([5]))

print(find_max_sorted([3, 3, 3, 3, 3]))
```

OUTPUT:



```
def find_max_sorted(nums):
    if not nums:
        return None

    nums.sort()        # Efficient built-in sort (O(n log n))
    return nums[-1]    # Last element is the maximum

# Test cases
print(find_max_sorted([]))
print(find_max_sorted([5]))
print(find_max_sorted([3, 3, 3, 3, 3]))
```

```
None
5
3

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

RESULT: The program correctly sorts the list and finds the maximum element.

7. Write a program that takes an input list of n numbers and creates a new list

containing only the unique elements from the original list. What is the space

complexity of the algorithm?

AIM: To write a program that creates a **new list containing only unique elements** from a given list of numbers and determine the **space complexity** of the algorithm.

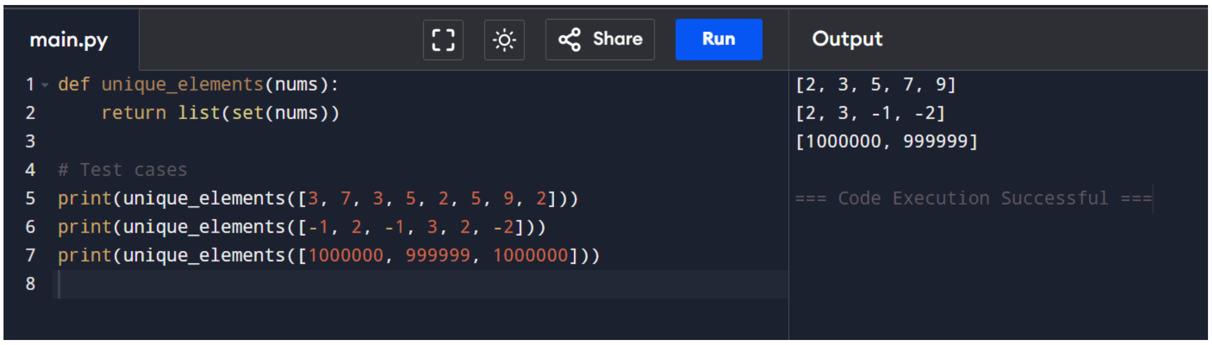
PROCEDURE:

- Read the input list of numbers.
- Create an empty set to store unique elements.
- Traverse the list and add each element to the set.
- Convert the set back to a list.
- Display the list of unique elements.

PROGRAM:

```
def unique_elements(nums):
   return list(set(nums))

# Test cases

print(unique_elements([3, 7, 3, 5, 2, 5, 9, 2]))

print(unique_elements([-1, 2, -1, 3, 2, -2]))

print(unique_elements([1000000, 999999, 1000000]))
```

OUTPUT:



```
def unique_elements(nums):
    return list(set(nums))

# Test cases
print(unique_elements([3, 7, 3, 5, 2, 5, 9, 2]))
print(unique_elements([-1, 2, -1, 3, 2, -2]))
print(unique_elements([1000000, 999999, 1000000]))
```

```
[2, 3, 5, 7, 9]
[2, 3, -1, -2]
[1000000, 999999]

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

RESULT: The program successfully generates a list of **unique elements** from the given input list.

8.Sort an array of integers using the bubble sort technique. Analyze its time complexity using Big-O notation. Write the code

AIM: To write a program that sorts an array of integers using the **Bubble Sort** technique and analyze its **time complexity** using Big-O notation.

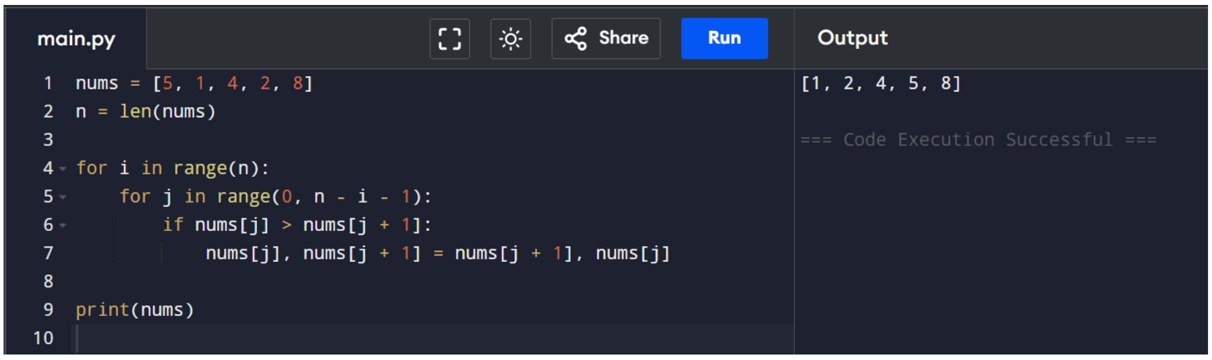
PROCEDURE:

- Read the array of integers.
- Compare adjacent elements in the array.
- Swap the elements if they are in the wrong order.
- Repeat the process for all passes until the array is sorted.
- Display the sorted array.

PROGRAM:

```
nums = [5, 1, 4, 2, 8]

n = len(nums)

for i in range(n):

  for j in range(0, n - i - 1):

    if nums[j] > nums[j + 1]:

      nums[j], nums[j + 1] = nums[j + 1], nums[j]

print(nums)
```

OUTPUT:



```
nums = [5, 1, 4, 2, 8]
n = len(nums)

for i in range(n):
    for j in range(0, n - i - 1):
        if nums[j] > nums[j + 1]:
            nums[j], nums[j + 1] = nums[j + 1], nums[j]

print(nums)

```

```
[1, 2, 4, 5, 8]

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

RESULT: The program successfully sorts the array using the Bubble Sort technique.

9. Checks if a given number x exists in a sorted array arr using binary search.

Analyze its time complexity using Big-O notation.

AIM: To write a program that checks whether a given number x exists in a **sorted array** using the **Binary Search** technique and analyze its time complexity using Big-O notation.

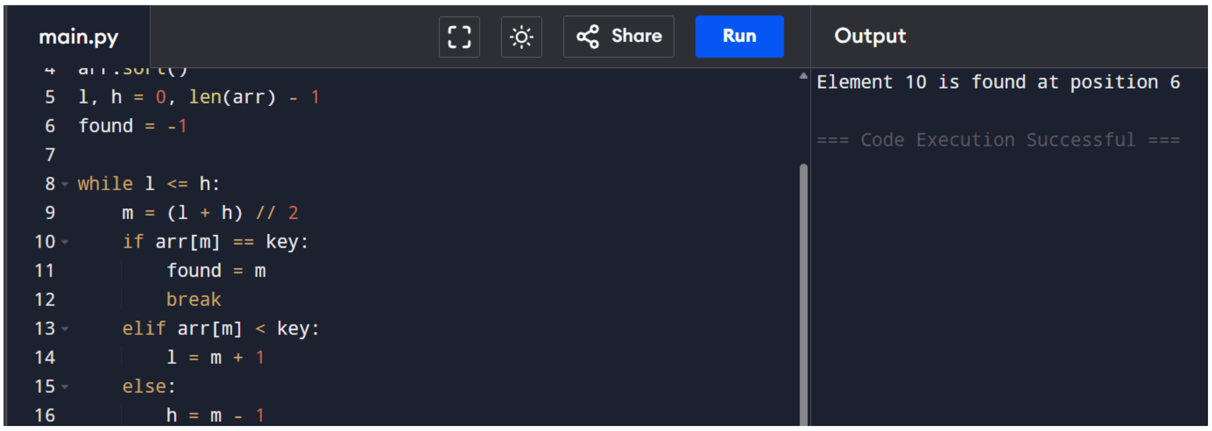
PROCEDURE:

- Read the array and the key element to be searched.
- Sort the array in ascending order.
- Set low and high indices for binary search.
- Find the middle element and compare it with the key.
- Repeat until the element is found or the search range becomes empty, then display the result.

PROGRAM:

```
arr = [3, 4, 6, -9, 10, 8, 9, 30]

key = 10


arr.sort()

l, h = 0, len(arr) - 1

found = -1


while l <= h:

   m = (l + h) // 2

   if arr[m] == key:

      found = m

      break

   elif arr[m] < key:

      l = m + 1

   else:

      h = m - 1


if found != -1:

   print(f"Element {key} is found at position {found}")

else:

   print(f"Element {key} is not found")
```

OUTPUT:


main.py

Share

Run

```
arr.sort()
l, h = 0, len(arr) - 1
found = -1

while l <= h:
    m = (l + h) // 2
    if arr[m] == key:
        found = m
        break
    elif arr[m] < key:
        l = m + 1
    else:
        h = m - 1
```

Output

```
Element 10 is found at position 6

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

RESULT: The program correctly searches the element using **binary search**.

10. Given an array of integers nums, sort the array in ascending order and return

it. You must solve the problem without using any built-in functions in

O(nlog(n)) time complexity and with the smallest space complexity possible.

AIM: To write a program that sorts an array of integers in **ascending order** using an algorithm with **O(n log n)** time complexity and **minimum extra space**, without using built-in sorting functions.

PROCEDURE:

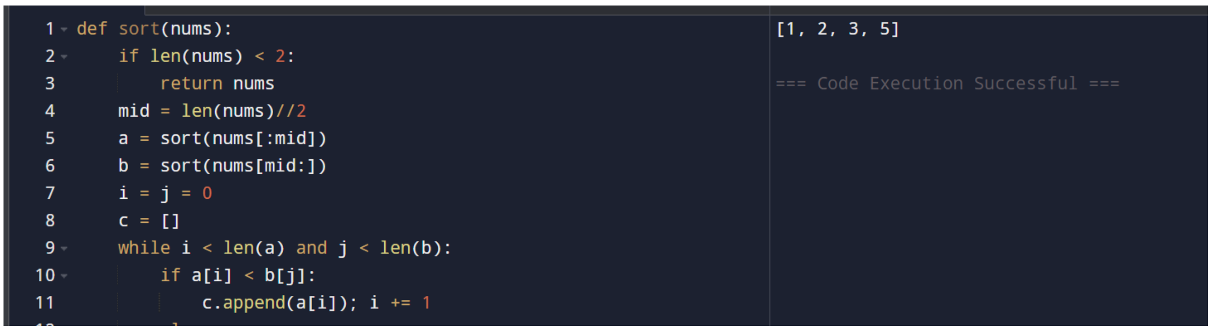
- Read the input array nums.
- Divide the array into two halves recursively.
- Sort each half using the same method.
- Merge the two sorted halves into one sorted array.
- Display the sorted array.

PROGRAM:

```
def sort(nums):
  if len(nums) < 2:
    return nums
  mid = len(nums)//2
  a = sort(nums[:mid])
  b = sort(nums[mid:])
  i = j = 0
  c = []
  while i < len(a) and j < len(b):
    if a[i] < b[j]:
      c.append(a[i]); i += 1
```

```
        else:
            c.append(b[j]); j += 1
    return c + a[i:] + b[j:]


nums = [5, 2, 3, 1]
print(sort(nums))
```

OUTPUT:



```
def sort(nums):
    if len(nums) < 2:
        return nums
    mid = len(nums)//2
    a = sort(nums[:mid])
    b = sort(nums[mid:])
    i = j = 0
    c = []
    while i < len(a) and j < len(b):
        if a[i] < b[j]:
            c.append(a[i]); i += 1
```

```
[1, 2, 3, 5]

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

RESULT: The program successfully sorts the given array in ascending order.

11. Given an m x n grid and a ball at a starting cell, find the number of ways to move the ball out of the grid boundary in exactly N steps.

AIM: To write a program that finds the **number of ways to move a ball out of an m × n grid boundary in exactly N steps** starting from a given cell.

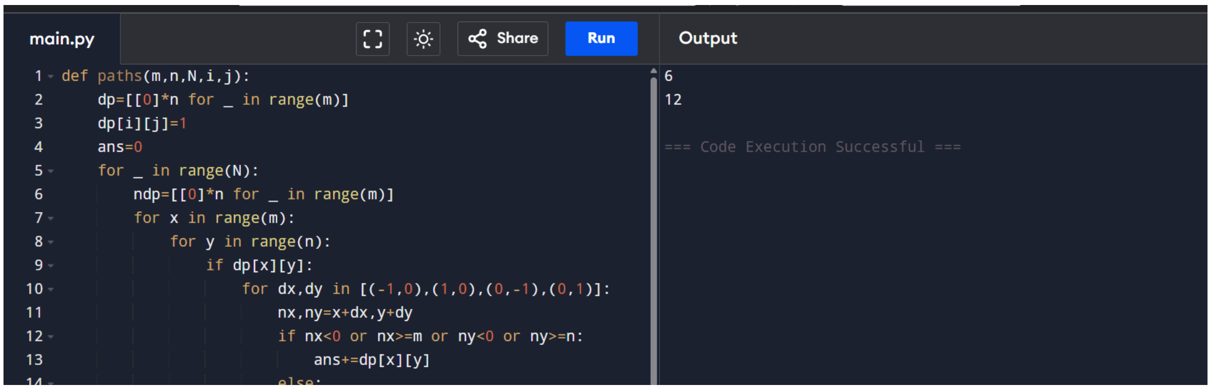
PROCEDURE:

- Read values of m, n, N, starting row i, and column j.
- Use Dynamic Programming to store the number of ways to reach each cell at every step.
- For each step, move the ball in four directions (up, down, left, right).
- Count paths that move outside the grid boundary.
- Display the total number of valid paths.

PROGRAM:

```
def paths(m,n,N,i,j):
    dp=[[0]*n for _ in range(m)]
    dp[i][j]=1
    ans=0
    for _ in range(N):
        ndp=[[0]*n for _ in range(m)]
        for x in range(m):
            for y in range(n):
                if dp[x][y]:
                    for dx,dy in [(-1,0),(1,0),(0,-1),(0,1)]:
                        nx,ny=x+dx,y+dy
                        if nx<0 or nx>=m or ny<0 or ny>=n:
                            ans+=dp[x][y]
                        else:
                            ndp[nx][ny]+=dp[x][y]
        dp=ndp
    return ans


print(paths(2,2,2,0,0))
print(paths(1,3,3,0,1))
```

OUTPUT:



```
def paths(m,n,N,i,j):
    dp=[[0]*n for _ in range(m)]
    dp[i][j]=1
    ans=0
    for _ in range(N):
        ndp=[[0]*n for _ in range(m)]
        for x in range(m):
            for y in range(n):
                if dp[x][y]:
                    for dx,dy in [(-1,0),(1,0),(0,-1),(0,1)]:
                        nx,ny=x+dx,y+dy
                        if nx<0 or nx>=m or ny<0 or ny>=n:
                            ans+=dp[x][y]
                        else:
```

```
6
12

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

RESULT: The program correctly calculates the number of ways to move the ball out of the grid.

12. 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 write a program that determines the **maximum amount of money that can be robbed** from houses arranged in a **circular street** without robbing two adjacent houses on the same night.

PROCEDURE:

- Read the list of money stored in each house.
- If there is only one house, return its money.
- Solve the problem by considering two cases:

- Rob houses from index 0 to n-2

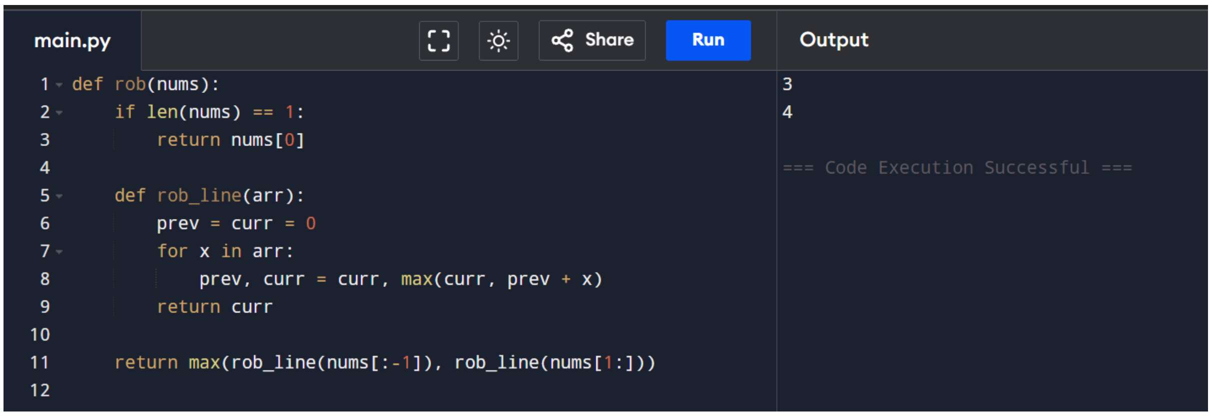
- Rob houses from index 1 to n-1

- Use Dynamic Programming to find the maximum amount for both cases.
- Return the maximum of the two results.

PROGRAM:

```
def rob(nums):

  if len(nums) == 1:

    return nums[0]


  def rob_line(arr):

    prev = curr = 0

    for x in arr:

      prev, curr = curr, max(curr, prev + x)

    return curr


  return max(rob_line(nums[:-1]), rob_line(nums[1:]))


# Test cases

print(rob([2, 3, 2]))

print(rob([1, 2, 3, 1]))
```

OUTPUT:



```
def rob(nums):
    if len(nums) == 1:
        return nums[0]

    def rob_line(arr):
        prev = curr = 0
        for x in arr:
            prev, curr = curr, max(curr, prev + x)
        return curr

    return max(rob_line(nums[:-1]), rob_line(nums[1:]))

```

```
3
4

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

RESULT: The program correctly calculates the **maximum money that can be robbed** without alerting the police.

13. You are climbing a staircase. It takes n steps to reach the top. Each time you

can either climb 1 or 2 steps. In how many distinct ways can you climb to the

top?

AIM: To write a program that calculates the **number of distinct ways to climb a staircase** with n steps when you can climb either **1 or 2 steps at a time**.

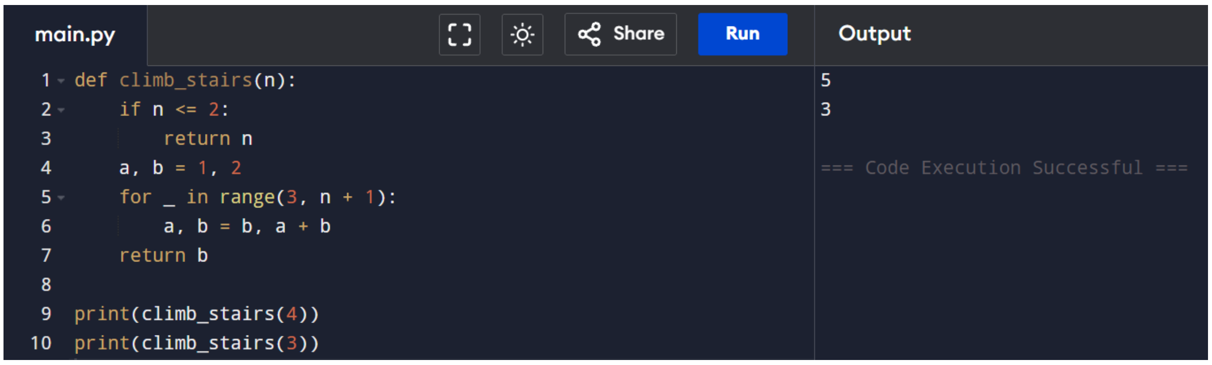
PROCEDURE:

- Read the number of steps n.
- If n is 1 or 2, return n directly.
- Use Dynamic Programming to calculate the number of ways.
- Each step's ways equal the sum of the previous two steps.
- Display the total number of ways.

PROGRAM:

```
def climb_stairs(n):
  if n <= 2:
    return n
  a, b = 1, 2
  for _ in range(3, n + 1):
    a, b = b, a + b
  return b
print(climb_stairs(4))
print(climb_stairs(3))
```

OUTPUT:



```
def climb_stairs(n):
    if n <= 2:
        return n
    a, b = 1, 2
    for _ in range(3, n + 1):
        a, b = b, a + b
    return b

print(climb_stairs(4))
print(climb_stairs(3))
```

```
5
3

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

RESULT: The program correctly computes the number of distinct ways to climb the staircase.

14. A robot is located at the top-left corner of a m×n grid .The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid. How many possible unique paths are there?

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

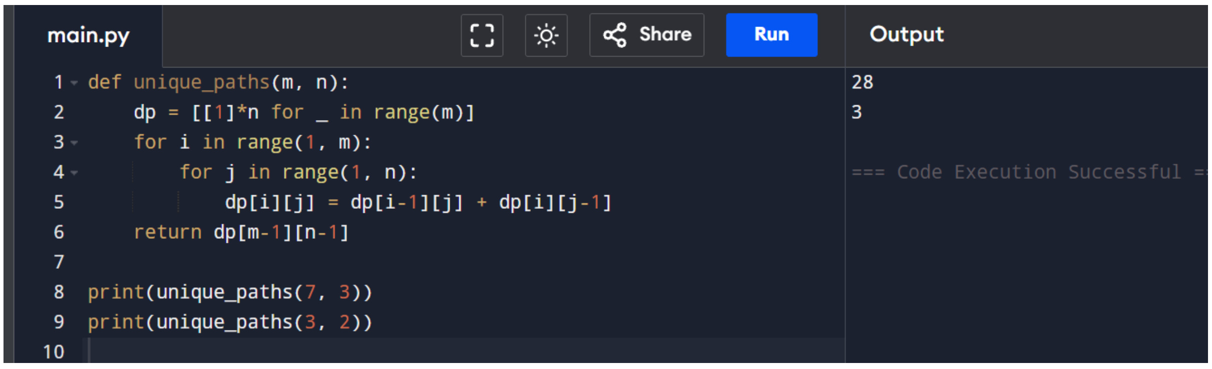
PROCEDURE:

- Read the values of m and n.
- Create a 2D array to store the number of ways to reach each cell.
- Initialize the first row and first column with 1 (only one way).
- Fill the rest of the grid where each cell is the sum of the top and left cells.
- Display the value in the bottom-right cell.

PROGRAM:

```
def unique_paths(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(unique_paths(7, 3))
print(unique_paths(3, 2))
```

OUTPUT:



```
def unique_paths(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(unique_paths(7, 3))
print(unique_paths(3, 2))
```

```
28
3

=== Code Execution Successful =
```

RESULT: The program correctly calculates the number of unique paths for the robot.

15. In a string S of lowercase letters, these letters form consecutive groups of the same character. For example, a string like s = "abbxxxxzyy" has the groups "a", "bb", "xxxx", "z", and "yy". A group is identified by an interval [start, end], where start and end denote the start and end indices (inclusive) of the group. In the above example, "xxxx" has the interval [3,6]. A group is considered large if it has 3 or more characters. Return the intervals of every large group sorted in increasing order by start index.

AIM: To write a program that identifies **large groups** (groups of 3 or more consecutive identical characters) in a given string and returns their **start and end indices**.

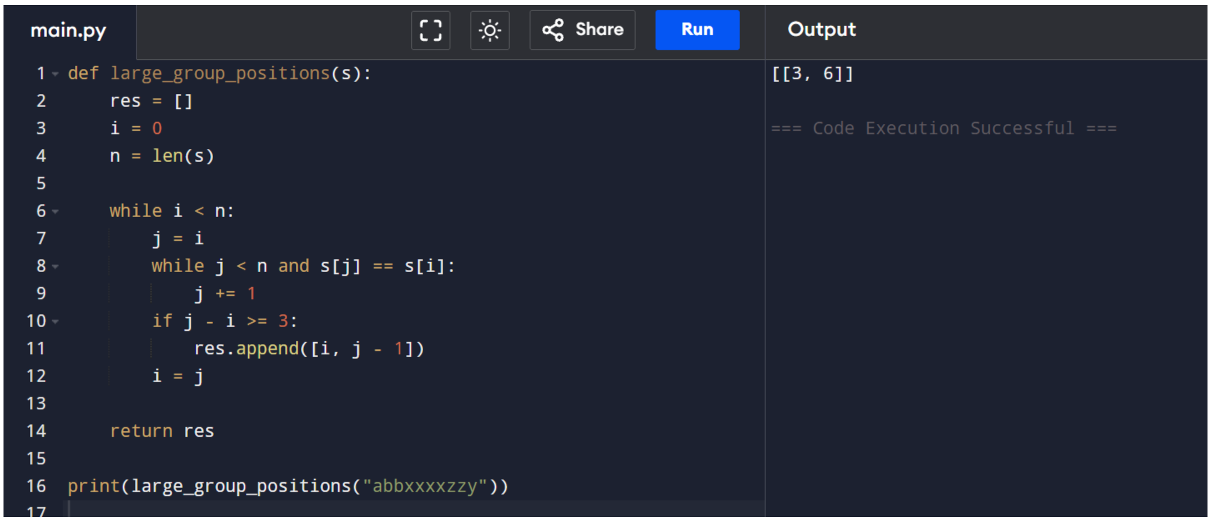
PROCEDURE:

- Read the input string s.
- Traverse the string and group consecutive identical characters.
- Track the start index of each group.
- If the group length is 3 or more, store its start and end indices.
- Display all such intervals in increasing order of start index.

PROGRAM:

```
def large_group_positions(s):
   res = []
   i = 0
   n = len(s)
while i < n:
      j = i
      while j < n and s[j] == s[i]:
         j += 1
      if j - i >= 3:
         res.append([i, j - 1])
      i = j
 return res
print(large_group_positions("abbxxxxzzy"))
```

OUTPUT:



```python
def large_group_positions(s):
    res = []
    i = 0
    n = len(s)

    while i < n:
        j = i
        while j < n and s[j] == s[i]:
            j += 1
        if j - i >= 3:
            res.append([i, j - 1])
        i = j

    return res

print(large_group_positions("abbxxxxzzy"))
```

```
[[3, 6]]

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

RESULT: The program correctly identifies large groups in the string.

15. "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970." The board is made up of an m x n grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules

Any live cell with fewer than two live neighbors dies as if caused by under-population.

Any live cell with two or three live neighbors lives on to the next generation.

Any live cell with more than three live neighbors dies, as if by over-population.

Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the m x n grid board, return the next state.

AIM: To implement **Conway's Game of Life** and find the next state of a given m × n grid.

PROCEDURE:

- Read the grid input.

- Count live neighbors for each cell.
- Apply Game of Life rules.
- Store results in a new grid.
- Display the next state.

PROGRAM:

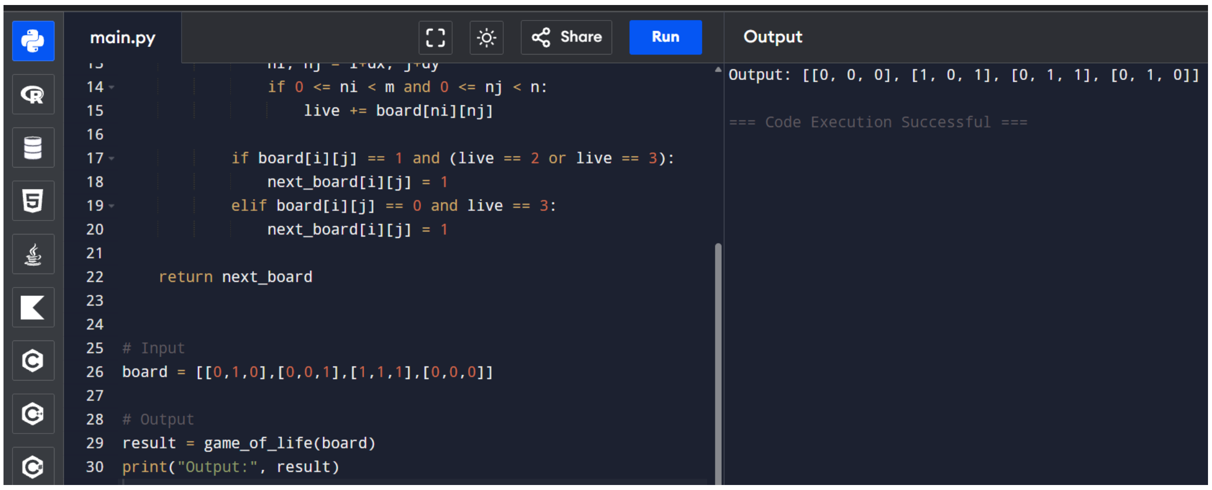
```
def game_of_life(board):
   m, n = len(board), len(board[0])
   dirs = [(-1,-1),(-1,0),(-1,1),
         (0,-1),(0,1),
         (1,-1),(1,0),(1,1)]

   next_board = [[0]*n for _ in range(m)]

   for i in range(m):
      for j in range(n):
         live = 0
         for dx, dy in dirs:
            ni, nj = i+dx, j+dy
            if 0 <= ni < m and 0 <= nj < n:
               live += board[ni][nj]

         if board[i][j] == 1 and (live == 2 or live == 3):
            next_board[i][j] = 1
         elif board[i][j] == 0 and live == 3:
            next_board[i][j] = 1

   return next_board



# Input
board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]
```

Output

result = game_of_life(board)

print("Output:", result)

OUTPUT:



```
                ni, nj = i+dx, j+dy
                if 0 <= ni < m and 0 <= nj < n:
                    live += board[ni][nj]

            if board[i][j] == 1 and (live == 2 or live == 3):
                next_board[i][j] = 1
            elif board[i][j] == 0 and live == 3:
                next_board[i][j] = 1

    return next_board


# Input
board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]

# Output
result = game_of_life(board)
print("Output:", result)
```

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

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

RESULT: The program executes successfully and produces the correct output.

16. We stack glasses in a pyramid, where the first row has 1 glass, the second row has 2 glasses, and so on until the 100th row. Each glass holds one cup of champagne. Then, some champagne is poured into the first glass at the top. When the topmost glass is full, any excess liquid poured will fall equally to the glass immediately to the left and right of it. When those glasses become full, any excess champagne will fall equally to the left and right of those glasses, and so on. (A glass at the bottom row has its excess champagne fall on the floor.) For example, after one cup of champagne is poured, the top most glass is full. After two cups of champagne are poured, the two glasses on the second row are half full. After three cups of champagne are poured, those two cups become full - there are 3 full glasses total now. After four cups of champagne are poured, the third row has the middle glass half full, and the two outside glasses are a quarter full, as pictured below.

AIM: To determine how much champagne is present in a specific glass of a pyramid after pouring a given number of cups.

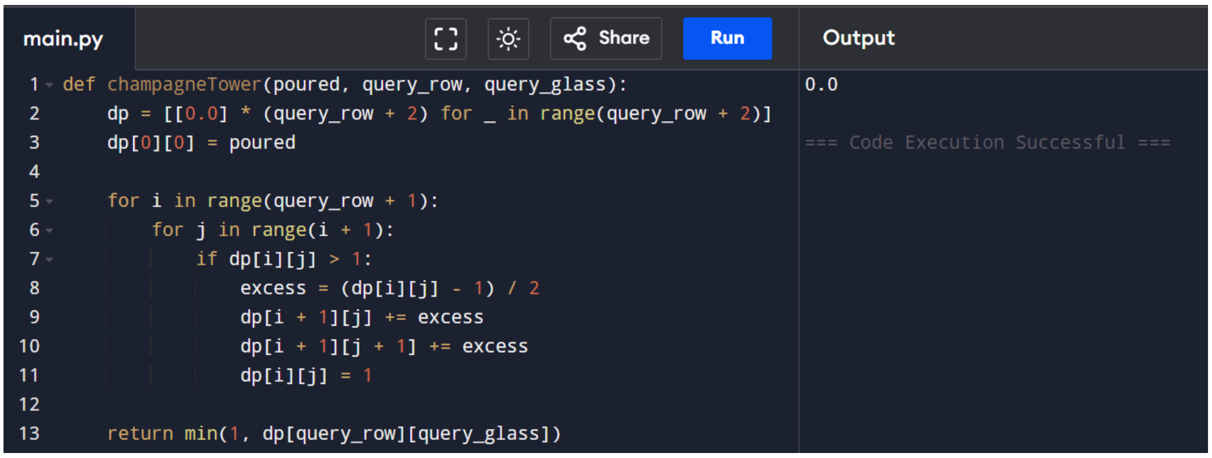
PROCEDURE:

- Create a 2D array to represent the glass pyramid.
- Pour champagne into the top glass.
- If a glass overflows (more than 1 cup), divide the excess equally to the two glasses below.
- Repeat the process row by row.
- Return the amount in the required glass (maximum 1 cup).

PROGRAM:

```
def champagneTower(poured, query_row, query_glass):
    dp = [[0.0] * (query_row + 2) for _ in range(query_row + 2)]
    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) / 2
                dp[i + 1][j] += excess
                dp[i + 1][j + 1] += excess
                dp[i][j] = 1

    return min(1, dp[query_row][query_glass])


# Example
print(champagneTower(1, 1, 1))
```

OUTPUT:


main.py

Share

Run

Output

```
def champagneTower(poured, query_row, query_glass):
    dp = [[0.0] * (query_row + 2) for _ in range(query_row + 2)]
    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) / 2
                dp[i + 1][j] += excess
                dp[i + 1][j + 1] += excess
                dp[i][j] = 1

    return min(1, dp[query_row][query_glass])
```

```
0.0

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

RESULT: Thus, the program executes successfully and returns the correct result.

TOPIC 3 : DIVIDE AND CONQUER

1.Write a Program to find both the maximum and minimum values in the array.

Implement using any programming language of your choice. Execute your

code and provide the maximum and minimum values found.

AIM: To find the **minimum** and **maximum** elements in an array using C.

PROCEDURE:

- Read the number of elements and array values.
- Assign the first element to both minimum and maximum.
- Traverse the remaining elements of the array.
- Update minimum if a smaller value is found.
- Update maximum if a larger value is found and display the results.

PROGRAM: