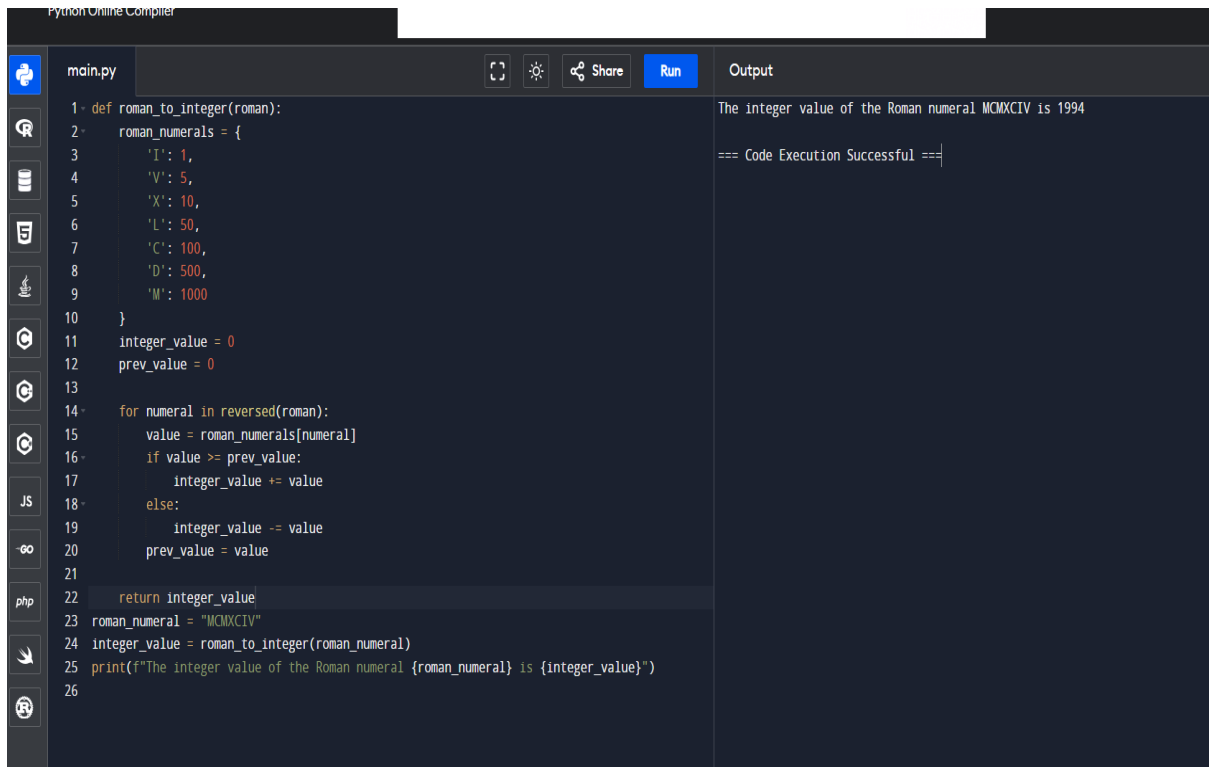


# 1.converting roman numbers to integers.



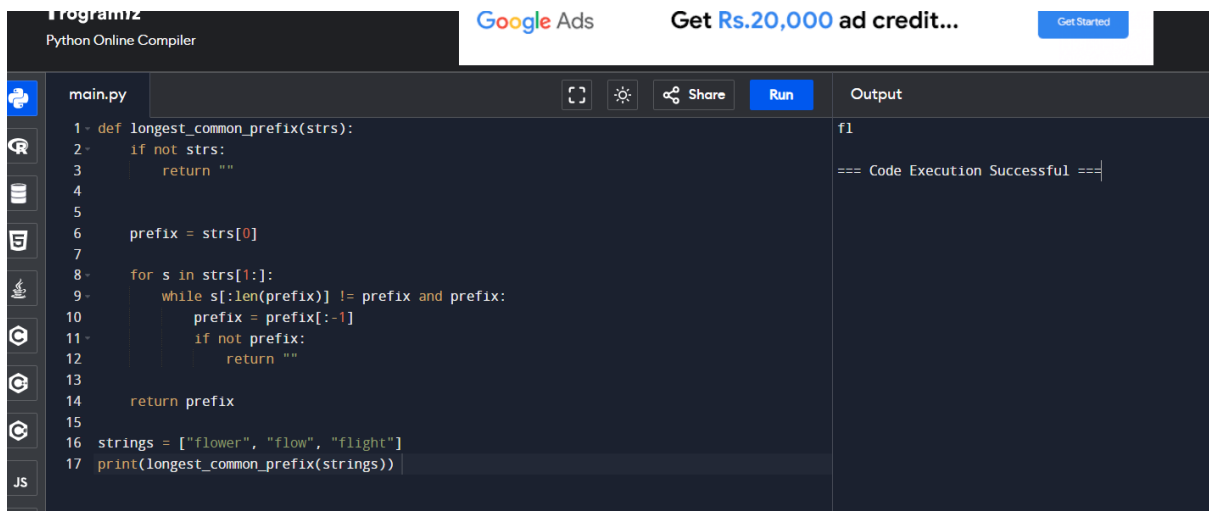
The screenshot shows a Python Online Compiler interface. The main editor displays a Python script for converting Roman numerals to integers. The script defines a function `roman_to_integer` that takes a Roman numeral string and returns its integer value. It uses a dictionary to map Roman numerals to their values and iterates through the string in reverse order to handle subtraction cases. The script then calls this function with the input "MCMXCIV" and prints the result.

```
1 def roman_to_integer(roman):
2     roman_numerals = {
3         'I': 1,
4         'V': 5,
5         'X': 10,
6         'L': 50,
7         'C': 100,
8         'D': 500,
9         'M': 1000
10    }
11    integer_value = 0
12    prev_value = 0
13
14    for numeral in reversed(roman):
15        value = roman_numerals[numeral]
16        if value >= prev_value:
17            integer_value += value
18        else:
19            integer_value -= value
20        prev_value = value
21
22    return integer_value
23 roman_numeral = "MCMXCIV"
24 integer_value = roman_to_integer(roman_numeral)
25 print(f"The integer value of the Roman numeral {roman_numeral} is {integer_value}")
26
```

The Output panel on the right shows the result of the execution:

```
The integer value of the Roman numeral MCMXCIV is 1994
=== Code Execution Successful ===
```

**2.write a function to find the longest common prefix string amongst an array of strings. “if there is no common prefix, return an empty string”.**



The screenshot shows a web-based Python compiler interface. At the top, there is a header with the 'Programiz' logo, 'Python Online Compiler' text, a 'Google Ads' banner, and a 'Get Rs.20,000 ad credit...' offer with a 'Get Started' button. The main area is divided into a code editor and an output panel. The code editor contains a Python function `longest_common_prefix` that takes a list of strings `strs` and returns the longest common prefix. The function uses a loop to compare the prefix of the first string with the prefix of each subsequent string, shortening the prefix until it matches or becomes empty. Below the function, a test case is provided: `strings = ["flower", "flow", "flight"]` and `print(longest_common_prefix(strings))`. The output panel shows the result `fl` and a message `=== Code Execution Successful ===`.

```
1- def longest_common_prefix(strs):
2-     if not strs:
3-         return ""
4-
5-
6-     prefix = strs[0]
7-
8-     for s in strs[1:]:
9-         while s[:len(prefix)] != prefix and prefix:
10-            prefix = prefix[:-1]
11-         if not prefix:
12-             return ""
13-
14-     return prefix
15-
16 strings = ["flower", "flow", "flight"]
17 print(longest_common_prefix(strings))
```

Output

fl

=== Code Execution Successful ===

**3. Given the root of a binary tree and an integer sum return true if the tree has a root to leaf such that adding up all the values.**

main.py	Output
<pre>1 class TreeNode: 2     def __init__(self, val=0, left=None, right=None): 3         self.val = val 4         self.left = left 5         self.right = right 6 7 def hasPathSum(root, sum): 8     if not root: 9         return False 10 11     if not root.left and not root.right: 12         return sum == root.val 13 14     sum -= root.val 15     return hasPathSum(root.left, sum) or hasPathSum(root.right, sum) 16 17 root = TreeNode(5) 18 root.left = TreeNode(4) 19 root.right = TreeNode(8) 20 root.left.left = TreeNode(11) 21 root.left.left.left = TreeNode(7) 22 root.left.left.right = TreeNode(2) 23 root.right.left = TreeNode(13) 24 root.right.right = TreeNode(4) 25 root.right.right.right = TreeNode(1) 26 27 print(hasPathSum(root, 22))</pre>	<pre>True === Code Execution Successful ===</pre>

## 4.Binary tree traversal.

```
main.py
1 class TreeNode:
2     def __init__(self, key):
3         self.left = None
4         self.right = None
5         self.val = key
6
7     def inorder_traversal(root):
8         result = []
9
10        def traverse(node):
11            if node:
12                traverse(node.left)
13                result.append(node.val)
14                traverse(node.right)
15
16        traverse(root)
17        return result
18
19    def preorder_traversal(root):
20        result = []
21
22        def traverse(node):
23            if node:
24                result.append(node.val)
25                traverse(node.left)
26                traverse(node.right)
27
28        traverse(root)
29        return result
30
31    def postorder_traversal(root):
32        result = []
33
34        def traverse(node):
35            if node:
36                traverse(node.left)
37                traverse(node.right)
38                result.append(node.val)
39
40        traverse(root)
41        return result
42
43    from collections import deque
44
45    def levelorder_traversal(root):
46        result = []
47        if not root:
48            return result
49
50        queue = deque([root])
51
52        while queue:
53            node = queue.popleft()
54            result.append(node.val)
55            if node.left:
56                queue.append(node.left)
57            if node.right:
58                queue.append(node.right)
59
60        return result
61
62
63 root = TreeNode()
```

Output

```
In-order Traversal: [4, 2, 5, 1, 3]
Pre-order Traversal: [1, 2, 4, 5, 3]
Post-order Traversal: [4, 5, 2, 3, 1]
Level-order Traversal: [1, 2, 3, 4, 5]

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

## 5.Bit Reserving.

main.py		Output
<pre>1 def reserve_bits(num, mask): 2     """ 3     Reserves (sets) bits in 'num' according to 'mask'. 4 5     :param num: Integer, the number whose bits are to be reserved. 6     :param mask: Integer, the mask indicating which bits to reserve. 7     :return: Integer, the result after reserving the bits. 8     """ 9     return num   mask 10 11 num = 0b00101100 12 mask = 0b00010010 13 14 15 result = reserve_bits(num, mask) 16 17 print(f"Original number: {bin(num)} ({num})") 18 print(f"Mask: {bin(mask)} ({mask})") 19 print(f"Result after reserving bits: {bin(result)} ({result})") 20</pre>	<pre>Original number: 0b101100 (44) Mask: 0b10010 (18) Result after reserving bits: 0b111110 (62)  === Code Execution Successful ===</pre>	

**6.convert sorted array to binary search tree give an integer array num where the elements are sorted in ascending order, convert it to a height-balanced.**

main.py

Share

Run

Output

```
1 class TreeNode:  
2     def __init__(self, val=0, left=None, right=None):  
3         self.val = val  
4         self.left = left  
5         self.right = right  
6  
7 def isBalanced(root: TreeNode) -> bool:  
8     def checkHeight(node: TreeNode) -> int:  
9         if not node:  
10             return 0  
11  
12         left_height = checkHeight(node.left)  
13         if left_height == -1:  
14             return -1  
15  
16         right_height = checkHeight(node.right)  
17         if right_height == -1:  
18             return -1  
19  
20         if abs(left_height - right_height) > 1:  
21             return -1  
22  
23         return max(left_height, right_height) + 1  
24  
25     return checkHeight(root) != -1  
26  
27 root = TreeNode
```

=== Code Execution Successful ===

**7. Given a binary tree, determine if it is height balanced.**

```
main.py
1- class TreeNode:
2-     def __init__(self, value=0, left=None, right=None):
3-         self.value = value
4-         self.left = left
5-         self.right = right
6-
7- def is_balanced(root):
8-     """
9-     Determine if a binary tree is height-balanced.
10-
11-     :param root: TreeNode, the root of the binary tree.
12-     :return: True if the tree is height-balanced, False otherwise.
13-     """
14-     def check_height(node):
15-         if node is None:
16-             return 0
17-
18-         left_height = check_height(node.left)
19-         if left_height == -1:
20-             return -1
21-
22-         right_height = check_height(node.right)
23-         if right_height == -1:
24-             return -1
25-
26-         if abs(left_height - right_height) > 1:
27-             return -1
28-
29-         return max(left_height, right_height) + 1
30-
31-     return check_height(root) != -1
32-
33-
34- root = TreeNode(1)
35- root.left = TreeNode(2)
36- root.right = TreeNode(3)
37- root.left.left = TreeNode(4)
38- root.left.right = TreeNode(5)
39-
40- print(is_balanced(root))
41-
```

Output

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

**8.Climbing stairs, your 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?**



```
main.py
1 def climb_stairs(n):
2     """
3     Calculate the number of distinct ways to climb to the top of a staircase with
4     'n' steps.
5
6     :param n: Integer, the number of steps in the staircase.
7     :return: Integer, the number of distinct ways to reach the top.
8     """
9     if n == 1:
10        return 1
11    if n == 2:
12        return 2
13
14    dp = [0] * (n + 1)
15
16    dp[1] = 1
17    dp[2] = 2
18
19    for i in range(3, n + 1):
20        dp[i] = dp[i-1] + dp[i-2]
21
22    return dp[n]
23
24 n = 5
25 print(climb_stairs(n))
```

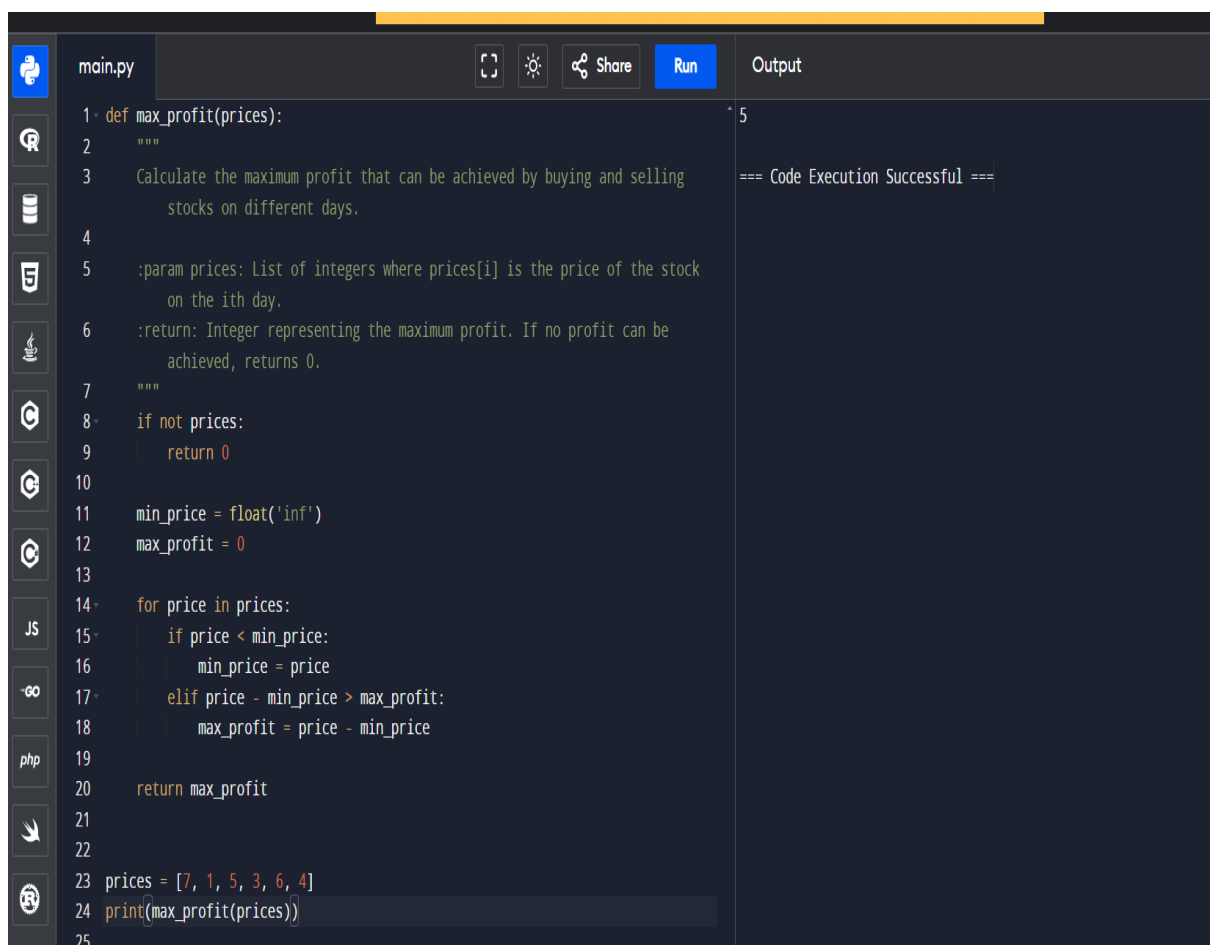
Output

8

=== Code Execution Successful ===

**9. Best time to buy and sell stock, you are given an array prices where prices[i] is the price of the stock on the i-th day. You want to maximize your profit by choosing a single day to buy one stock and a single day to sell one stock. Return the maximum profit you can achieve from this transaction. If there is no profit, return 0.**

**profit by choosing a single day to buy one stock and choosing a different day in the future to sell stocks. if you cannot achieve any profit, return 0. give python programming.**



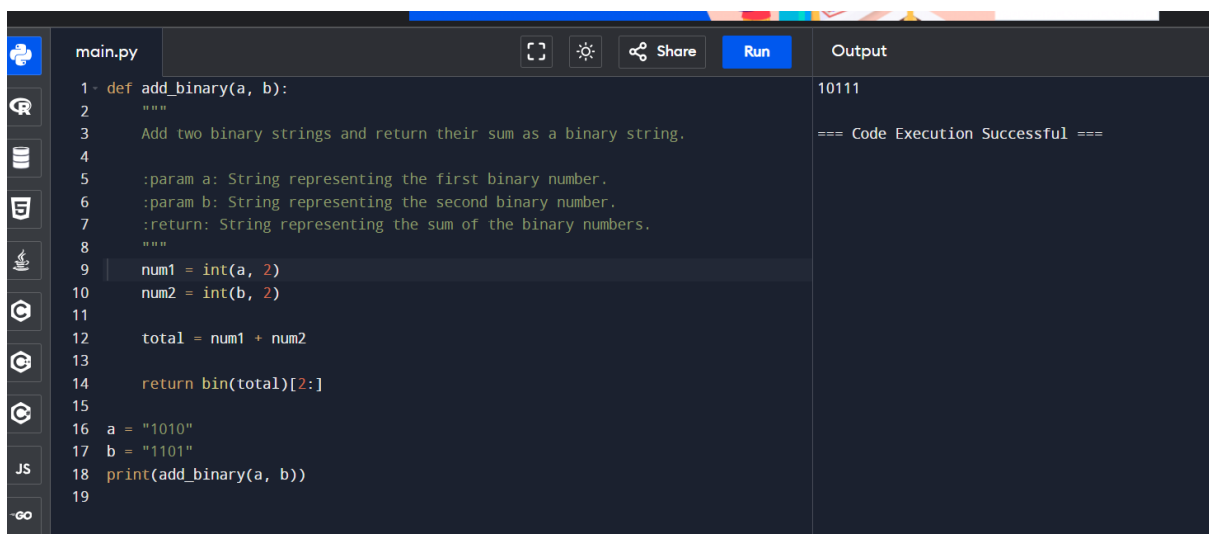
The image shows a Python IDE interface with a dark theme. On the left is a sidebar with icons for Python, R, Java, C++, JavaScript, Go, PHP, and others. The main editor window displays a Python script named 'main.py'. The script defines a function 'max\_profit(prices)' that calculates the maximum profit from a list of stock prices. It uses a single pass algorithm to find the minimum price and update the maximum profit as it iterates through the list. At the bottom, it initializes a 'prices' list with values [7, 1, 5, 3, 6, 4] and prints the result of 'max\_profit(prices)'. Above the editor, there are buttons for 'Run', 'Share', and 'Output'. The 'Run' button is highlighted in blue. To the right of the editor, the 'Output' pane shows the result '5' and a message '=== Code Execution Successful ==='.

```
1 def max_profit(prices):
2     """
3     Calculate the maximum profit that can be achieved by buying and selling
4     stocks on different days.
5
6     :param prices: List of integers where prices[i] is the price of the stock
7     on the ith day.
8     :return: Integer representing the maximum profit. If no profit can be
9     achieved, returns 0.
10    """
11    if not prices:
12        return 0
13
14    min_price = float('inf')
15    max_profit = 0
16
17    for price in prices:
18        if price < min_price:
19            min_price = price
20        elif price - min_price > max_profit:
21            max_profit = price - min_price
22
23    return max_profit
24
25 prices = [7, 1, 5, 3, 6, 4]
26 print(max_profit(prices))
```

5

=== Code Execution Successful ===

**10. Given two binary strings A and B ,return their sum as a binary string give python code.**



The screenshot shows a Python IDE with a file named 'main.py'. The code defines a function 'add\_binary(a, b)' that takes two binary strings as input and returns their sum as a binary string. The function uses the built-in 'int' function to convert the binary strings to integers, adds them, and then uses the 'bin' function to convert the result back to a binary string, removing the '0b' prefix. The function is tested with 'a = "1010"' and 'b = "1101"', and the result is printed. The output of the code execution is '10111'.

```
1- def add_binary(a, b):
2-     """
3-     Add two binary strings and return their sum as a binary string.
4-
5-     :param a: String representing the first binary number.
6-     :param b: String representing the second binary number.
7-     :return: String representing the sum of the binary numbers.
8-     """
9-     num1 = int(a, 2)
10-    num2 = int(b, 2)
11-
12-    total = num1 + num2
13-
14-    return bin(total)[2:]
15-
16- a = "1010"
17- b = "1101"
18- print(add_binary(a, b))
19-
```

Output

```
10111
=== Code Execution Successful ===
```