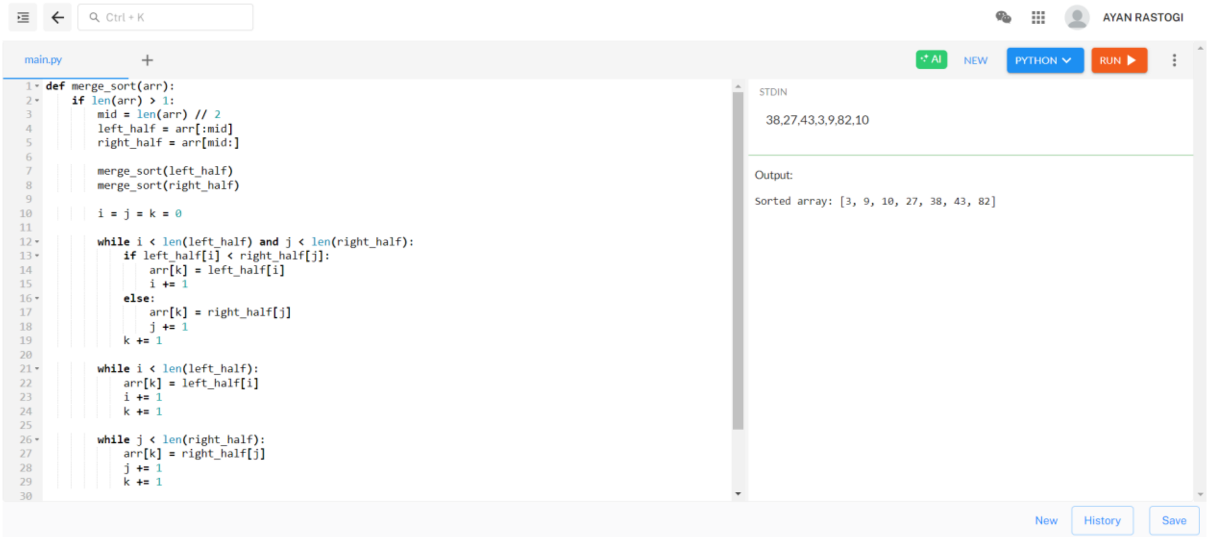
1.



```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

```

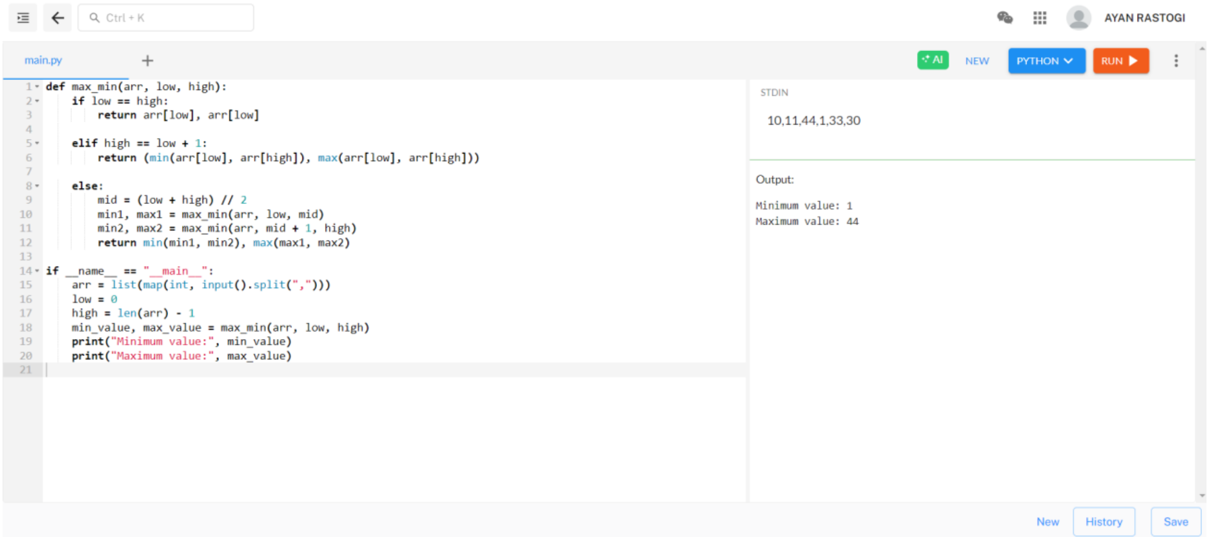
STDIN

```
38,27,43,3,9,82,10
```

Output:

```
Sorted array: [3, 9, 10, 27, 38, 43, 82]
```

2.



```python
def max_min(arr, low, high):
    if low == high:
        return arr[low], arr[low]

    elif high == low + 1:
        return (min(arr[low], arr[high]), max(arr[low], arr[high]))

    else:
        mid = (low + high) // 2
        min1, max1 = max_min(arr, low, mid)
        min2, max2 = max_min(arr, mid + 1, high)
        return min(min1, min2), max(max1, max2)

if __name__ == "__main__":
    arr = list(map(int, input().split(",")))
    low = 0
    high = len(arr) - 1
    min_value, max_value = max_min(arr, low, high)
    print("Minimum value:", min_value)
    print("Maximum value:", max_value)

```

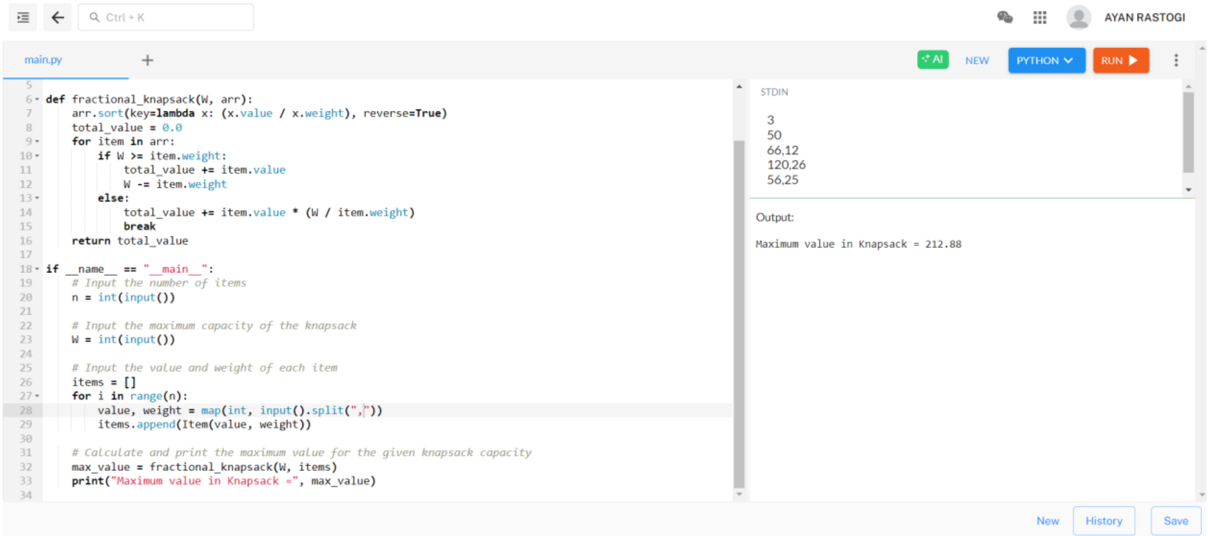
STDIN

```
10,11,44,1,33,30
```

Output:

```
Minimum value: 1
Maximum value: 44
```

3.



```python

def fractional_knapsack(W, arr):
    arr.sort(key=lambda x: (x.value / x.weight), reverse=True)
    total_value = 0.0
    for item in arr:
        if W >= item.weight:
            total_value += item.value
            W -= item.weight
        else:
            total_value += item.value * (W / item.weight)
            break
    return total_value

if __name__ == "__main__":
    # Input the number of items
    n = int(input())

    # Input the maximum capacity of the knapsack
    W = int(input())

    # Input the value and weight of each item
    items = []
    for i in range(n):
        value, weight = map(int, input().split(","))
        items.append(Item(value, weight))

    # Calculate and print the maximum value for the given knapsack capacity
    max_value = fractional_knapsack(W, items)
    print("Maximum value in Knapsack =", max_value)

```

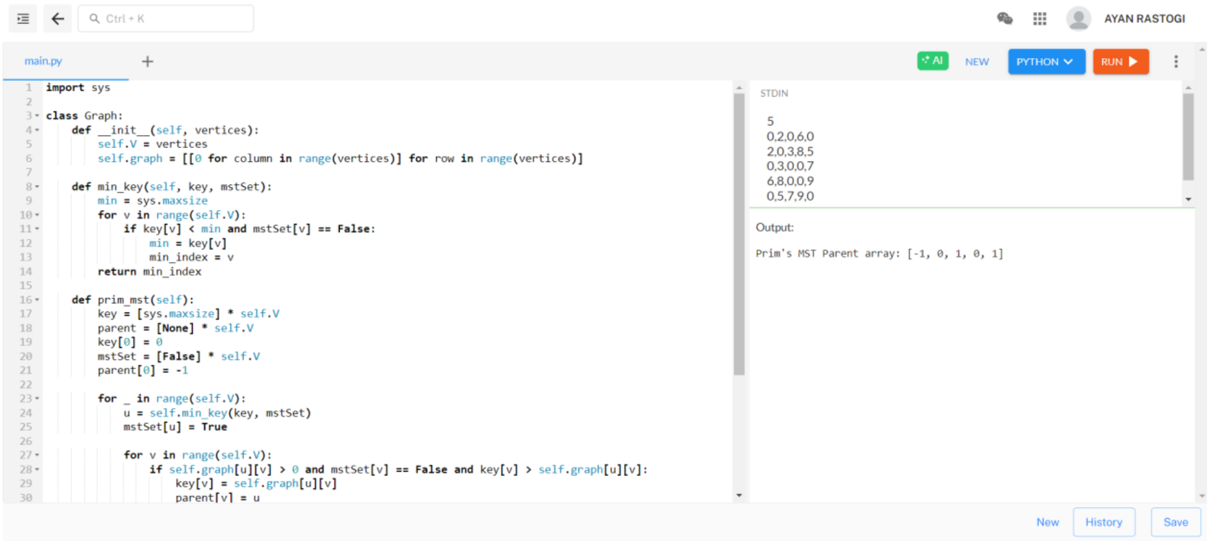
STDIN

```
3
50
66,12
120,26
56,25
```

Output:

```
Maximum value in Knapsack = 212.88
```

4.



```python
import sys

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

    def min_key(self, key, mstSet):
        min = sys.maxsize
        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v
        return min_index

    def prim_mst(self):
        key = [sys.maxsize] * self.V
        parent = [None] * self.V
        key[0] = 0
        mstSet = [False] * self.V
        parent[0] = -1

        for _ in range(self.V):
            u = self.min_key(key, mstSet)
            mstSet[u] = True

            for v in range(self.V):
                if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u
```

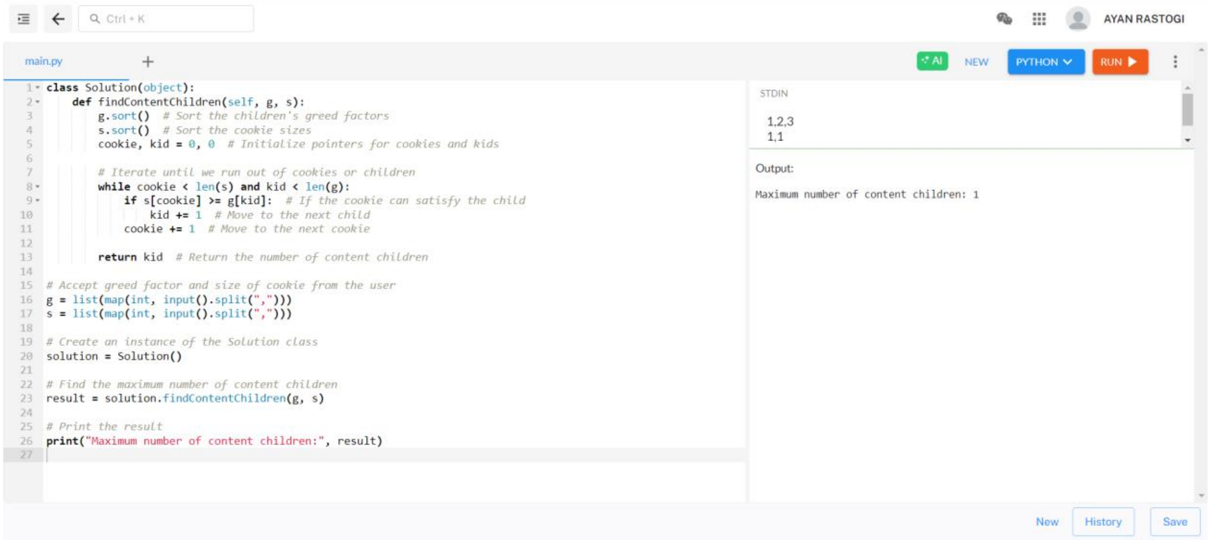
STDIN

```
5
0,2,0,6,0
2,0,3,8,5
0,3,0,0,7
6,8,0,0,9
0,5,7,9,0
```

Output:

```
Prim's MST Parent array: [-1, 0, 1, 0, 1]
```

5.



```python
class Solution(object):
    def findContentChildren(self, g, s):
        g.sort()  # Sort the children's greed factors
        s.sort()  # Sort the cookie sizes
        cookie, kid = 0, 0  # Initialize pointers for cookies and kids

        # Iterate until we run out of cookies or children
        while cookie < len(s) and kid < len(g):
            if s[cookie] >= g[kid]:  # If the cookie can satisfy the child
                kid += 1  # Move to the next child
            cookie += 1  # Move to the next cookie

        return kid  # Return the number of content children

# Accept greed factor and size of cookie from the user
g = list(map(int, input().split(",")))
s = list(map(int, input().split(",")))

# Create an instance of the Solution class
solution = Solution()

# Find the maximum number of content children
result = solution.findContentChildren(g, s)

# Print the result
print("Maximum number of content children:", result)
```

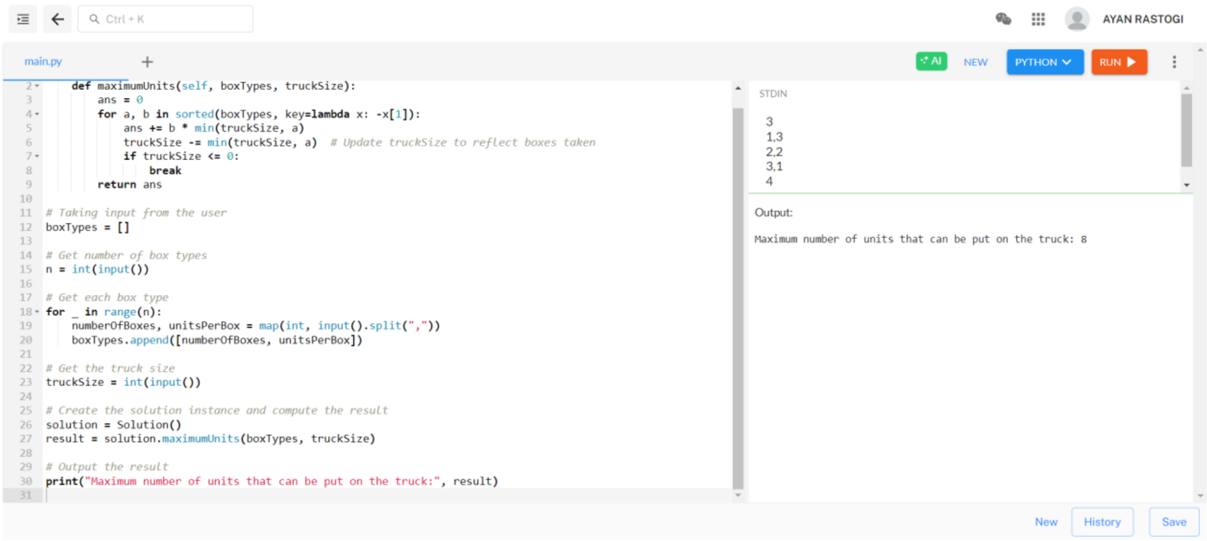
STDIN

```
1,2,3
1,1
```

Output:

```
Maximum number of content children: 1
```

6.



```python
    def maximumUnits(self, boxTypes, truckSize):
        ans = 0
        for a, b in sorted(boxTypes, key=lambda x: -x[1]):
            ans += b * min(truckSize, a)
            truckSize -= min(truckSize, a)  # Update truckSize to reflect boxes taken
            if truckSize <= 0:
                break
        return ans

# Taking input from the user
boxTypes = []

# Get number of box types
n = int(input())

# Get each box type
for _ in range(n):
    numberOfBoxes, unitsPerBox = map(int, input().split(","))
    boxTypes.append([numberOfBoxes, unitsPerBox])

# Get the truck size
truckSize = int(input())

# Create the solution instance and compute the result
solution = Solution()
result = solution.maximumUnits(boxTypes, truckSize)

# Output the result
print("Maximum number of units that can be put on the truck:", result)
```

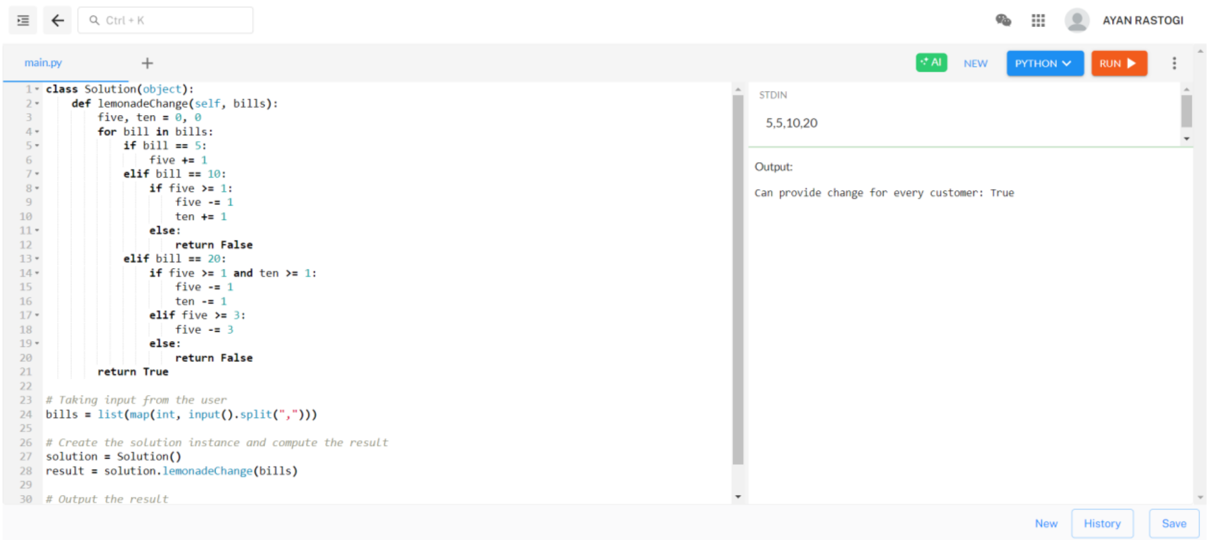
STDIN

```
3
1,3
2,2
3,1
4
```

Output:

```
Maximum number of units that can be put on the truck: 8
```

7.



```
class Solution(object):
    def lemonadeChange(self, bills):
        five, ten = 0, 0
        for bill in bills:
            if bill == 5:
                five += 1
            elif bill == 10:
                if five >= 1:
                    five -= 1
                    ten += 1
                else:
                    return False
            elif bill == 20:
                if five >= 1 and ten >= 1:
                    five -= 1
                    ten -= 1
                elif five >= 3:
                    five -= 3
                else:
                    return False
        return True

# Taking input from the user
bills = list(map(int, input().split(",")))

# Create the solution instance and compute the result
solution = Solution()
result = solution.lemonadeChange(bills)

# Output the result
```

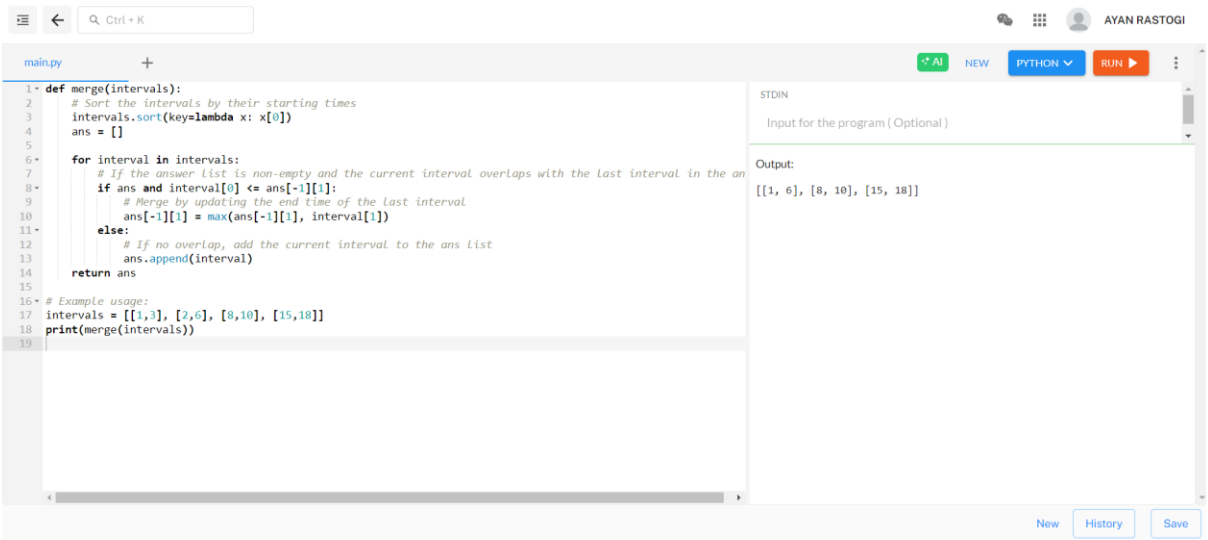
STDIN

```
5,5,10,20
```

Output:

```
Can provide change for every customer: True
```

8.



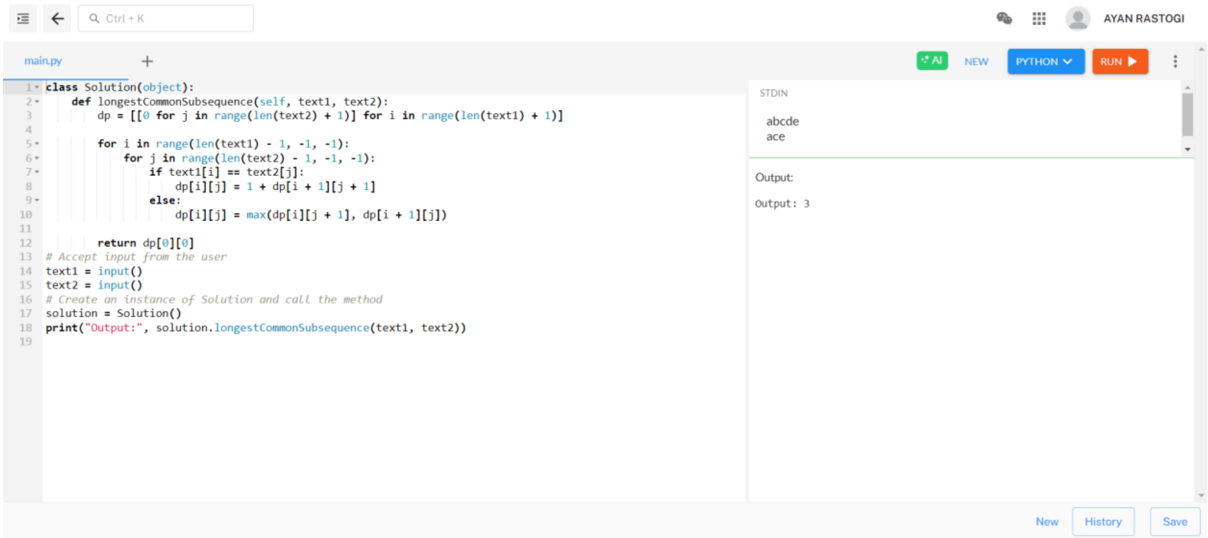
```
def merge(intervals):
    # Sort the intervals by their starting times
    intervals.sort(key=lambda x: x[0])
    ans = []

    for interval in intervals:
        # If the answer list is non-empty and the current interval overlaps with the last interval in the an
        if ans and interval[0] <= ans[-1][1]:
            # Merge by updating the end time of the last interval
            ans[-1][1] = max(ans[-1][1], interval[1])
        else:
            # If no overlap, add the current interval to the ans list
            ans.append(interval)
    return ans

# Example usage:
intervals = [[1,3], [2,6], [8,10], [15,18]]
print(merge(intervals))
```

STDIN

Output:

```
[[1, 6], [8, 10], [15, 18]]
```

9.



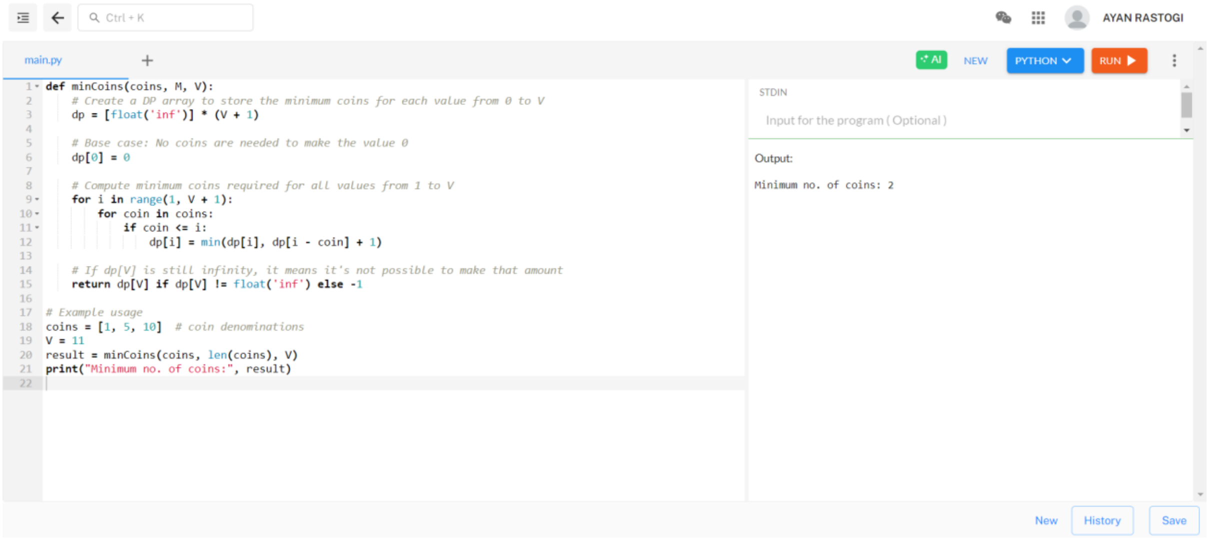
```python
class Solution(object):
    def longestCommonSubsequence(self, text1, text2):
        dp = [[0 for j in range(len(text2) + 1)] for i in range(len(text1) + 1)]

        for i in range(len(text1) - 1, -1, -1):
            for j in range(len(text2) - 1, -1, -1):
                if text1[i] == text2[j]:
                    dp[i][j] = 1 + dp[i + 1][j + 1]
                else:
                    dp[i][j] = max(dp[i][j + 1], dp[i + 1][j])

        return dp[0][0]
# Accept input from the user
text1 = input()
text2 = input()
# Create an instance of Solution and call the method
solution = Solution()
print("Output:", solution.longestCommonSubsequence(text1, text2))
```

```
abcde
ace
```

Output:

```
Output: 3
```

10.



```python
def minCoins(coins, M, V):
    # Create a DP array to store the minimum coins for each value from 0 to V
    dp = [float('inf')] * (V + 1)

    # Base case: No coins are needed to make the value 0
    dp[0] = 0

    # Compute minimum coins required for all values from 1 to V
    for i in range(1, V + 1):
        for coin in coins:
            if coin <= i:
                dp[i] = min(dp[i], dp[i - coin] + 1)

    # If dp[V] is still infinity, it means it's not possible to make that amount
    return dp[V] if dp[V] != float('inf') else -1

# Example usage
coins = [1, 5, 10]  # coin denominations
V = 11
result = minCoins(coins, len(coins), V)
print("Minimum no. of coins:", result)

```

Output:

```
Minimum no. of coins: 2
```