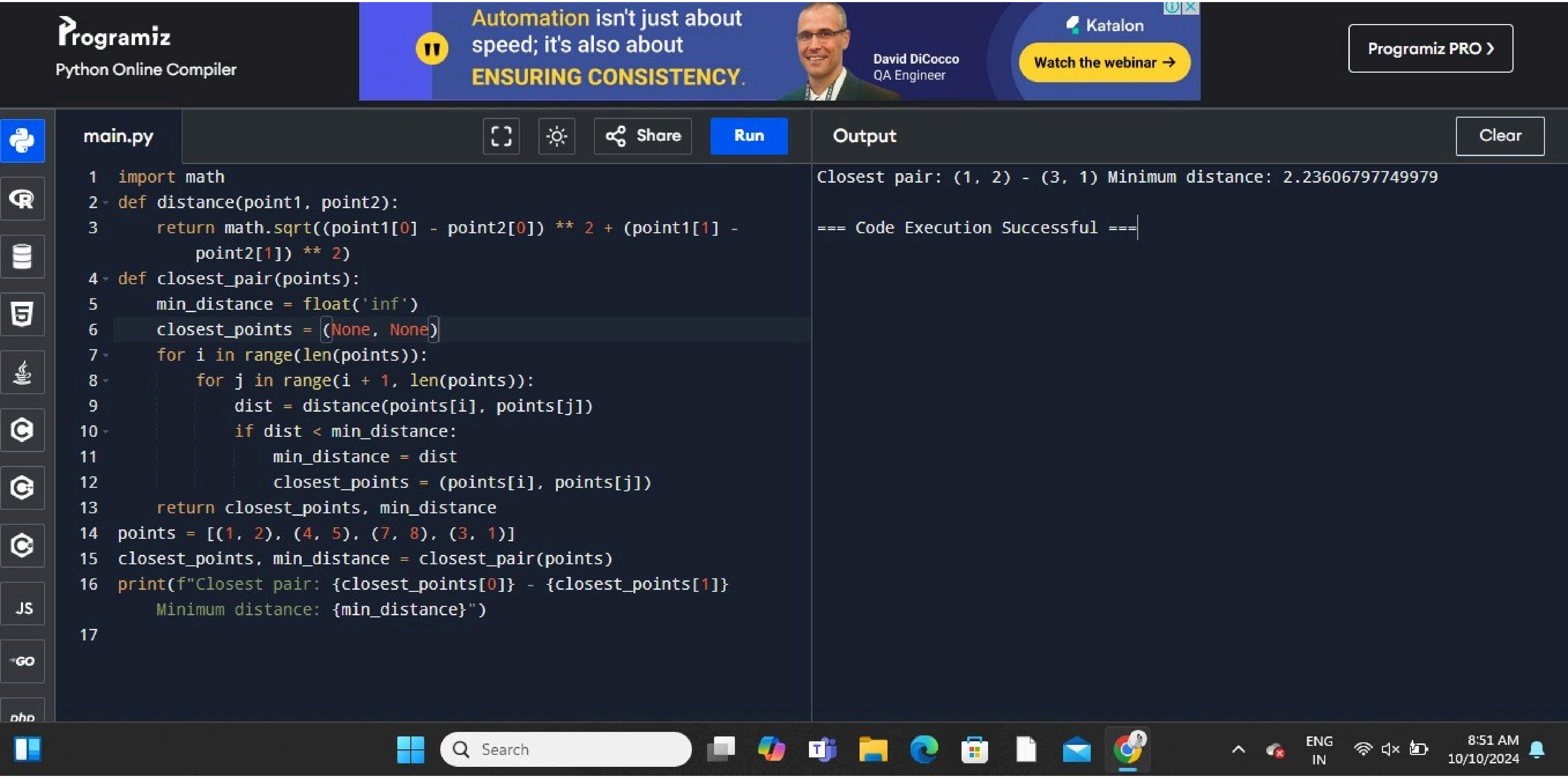
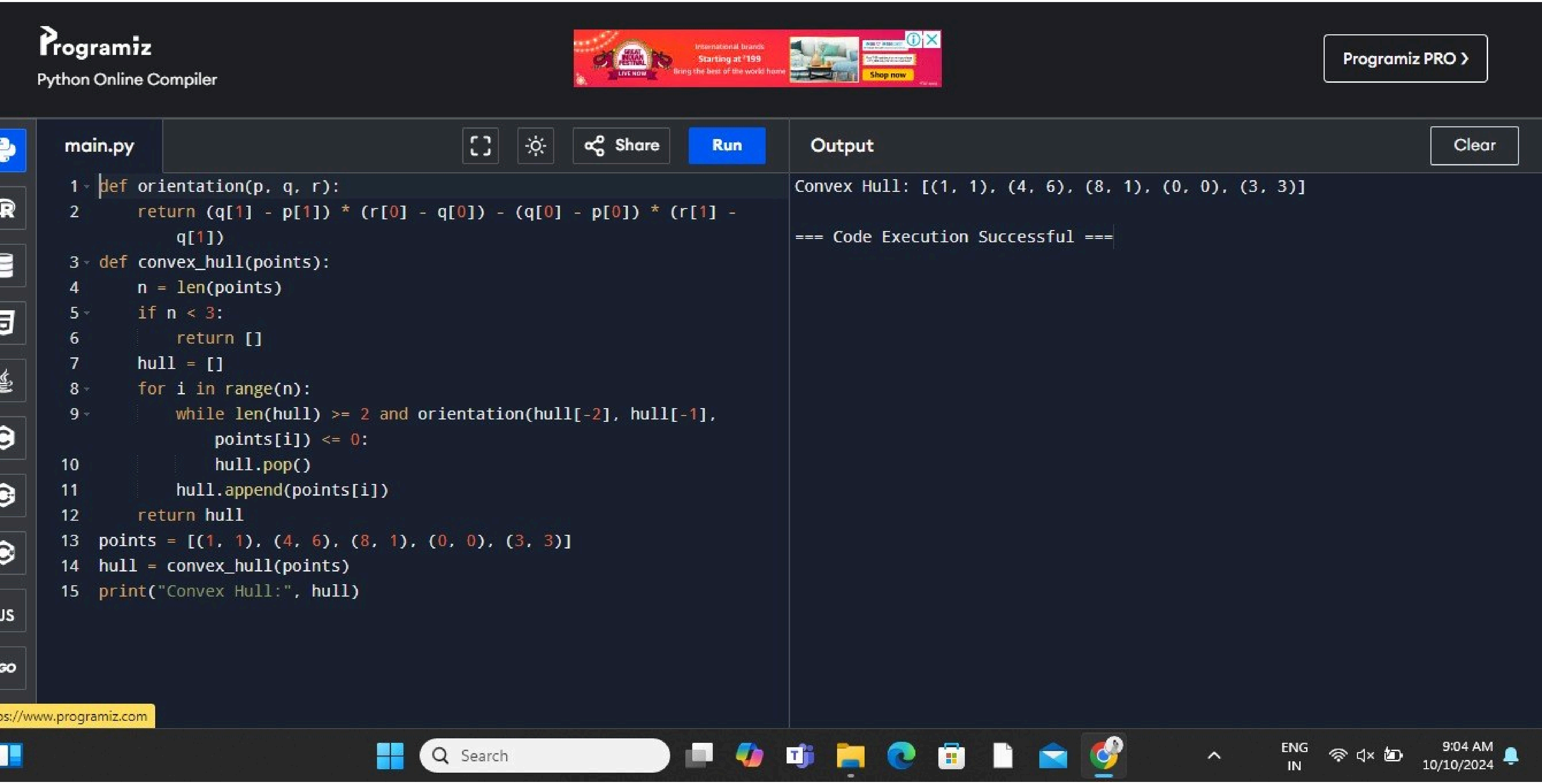


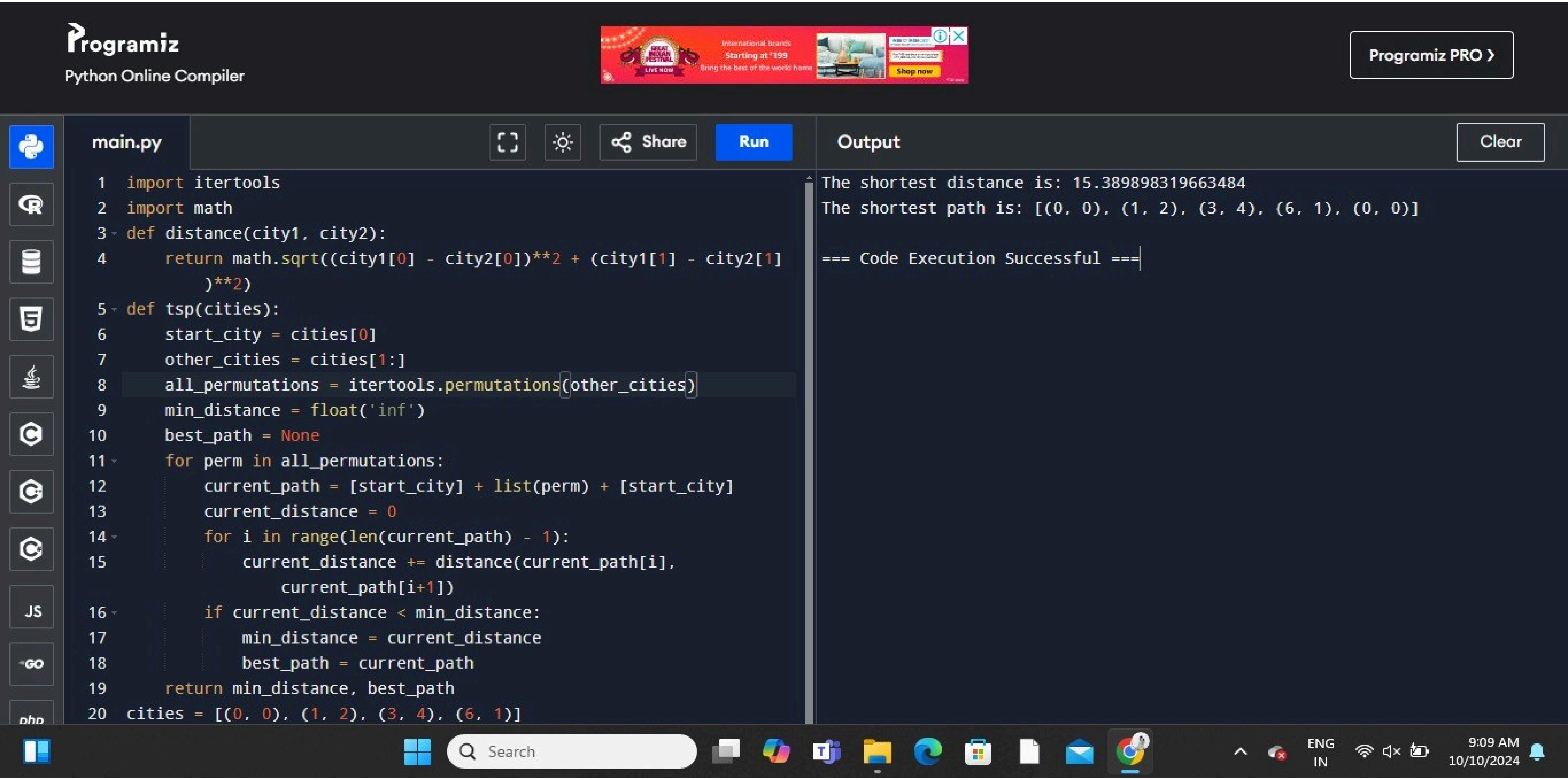
```python
import math
def distance(point1, point2):
    return math.sqrt((point1[0] - point2[0]) ** 2 + (point1[1] - point2[1]) ** 2)
def closest_pair(points):
    min_distance = float('inf')
    closest_points = (None, None)
    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            dist = distance(points[i], points[j])
            if dist < min_distance:
                min_distance = dist
                closest_points = (points[i], points[j])
    return closest_points, min_distance
points = [(1, 2), (4, 5), (7, 8), (3, 1)]
closest_points, min_distance = closest_pair(points)
print(f"Closest pair: {closest_points[0]} - {closest_points[1]} Minimum distance: {min_distance}")

```

```
Closest pair: (1, 2) - (3, 1) Minimum distance: 2.23606797749979

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



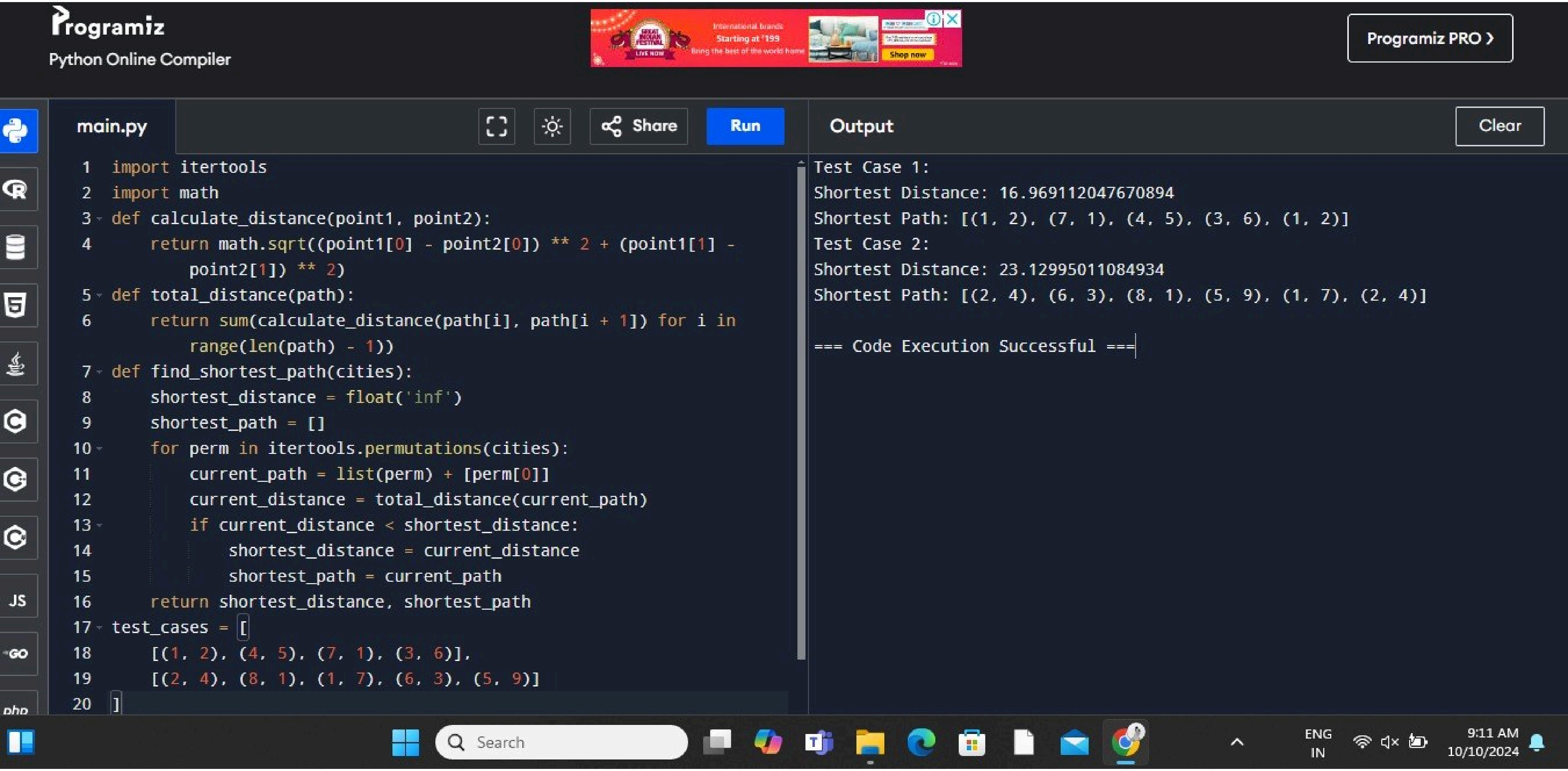
```python
def orientation(p, q, r):
    return (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] - q[1])
def convex_hull(points):
    n = len(points)
    if n < 3:
        return []
    hull = []
    for i in range(n):
        while len(hull) >= 2 and orientation(hull[-2], hull[-1], points[i]) <= 0:
            hull.pop()
        hull.append(points[i])
    return hull
points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
hull = convex_hull(points)
print("Convex Hull:", hull)
```

```
Convex Hull: [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]

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



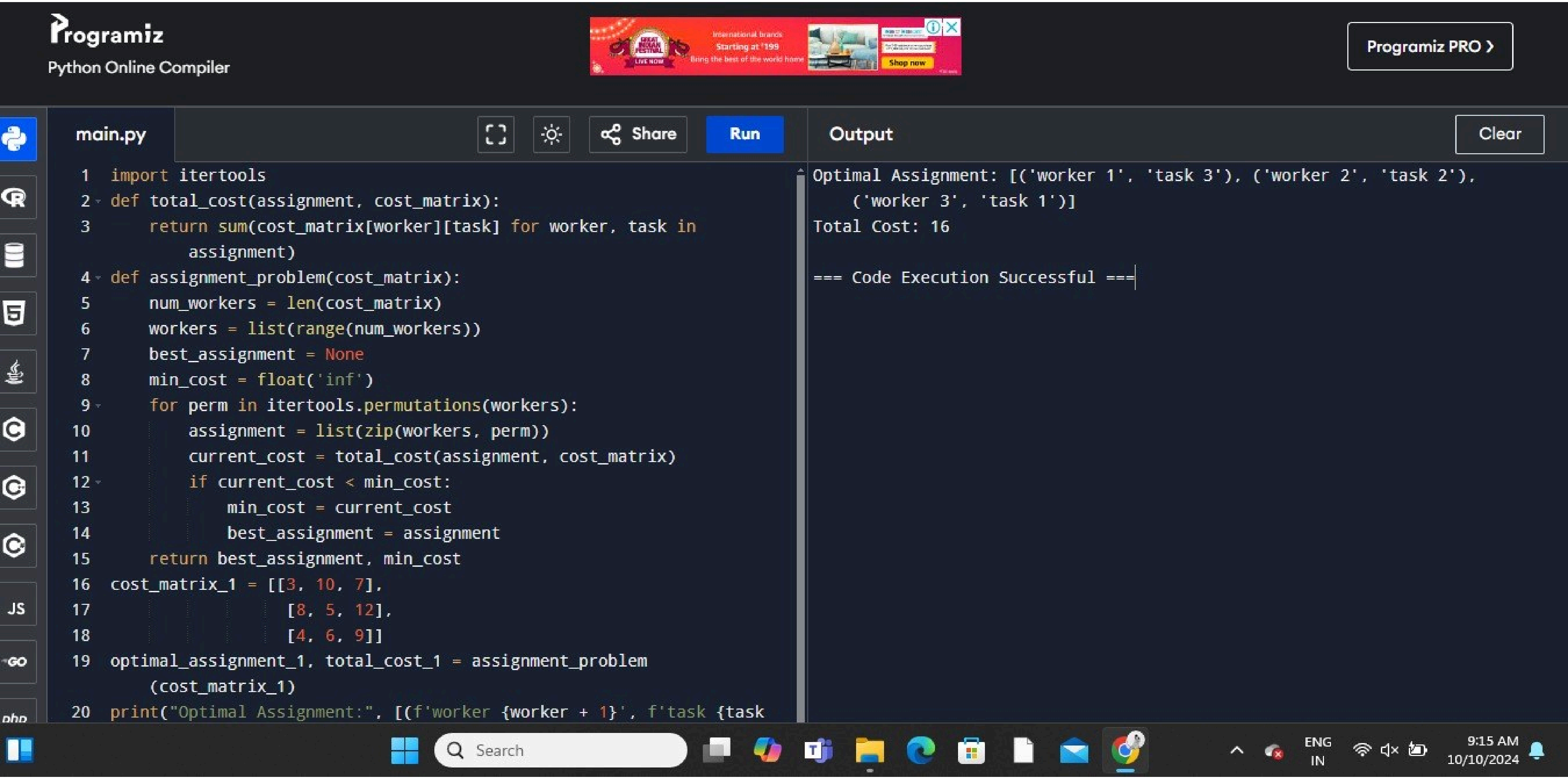
```python
import itertools
import math
def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)
def tsp(cities):
    start_city = cities[0]
    other_cities = cities[1:]
    all_permutations = itertools.permutations(other_cities)
    min_distance = float('inf')
    best_path = None
    for perm in all_permutations:
        current_path = [start_city] + list(perm) + [start_city]
        current_distance = 0
        for i in range(len(current_path) - 1):
            current_distance += distance(current_path[i], current_path[i+1])
        if current_distance < min_distance:
            min_distance = current_distance
            best_path = current_path
    return min_distance, best_path
cities = [(0, 0), (1, 2), (3, 4), (6, 1)]
```

```
The shortest distance is: 15.389898319663484
The shortest path is: [(0, 0), (1, 2), (3, 4), (6, 1), (0, 0)]

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



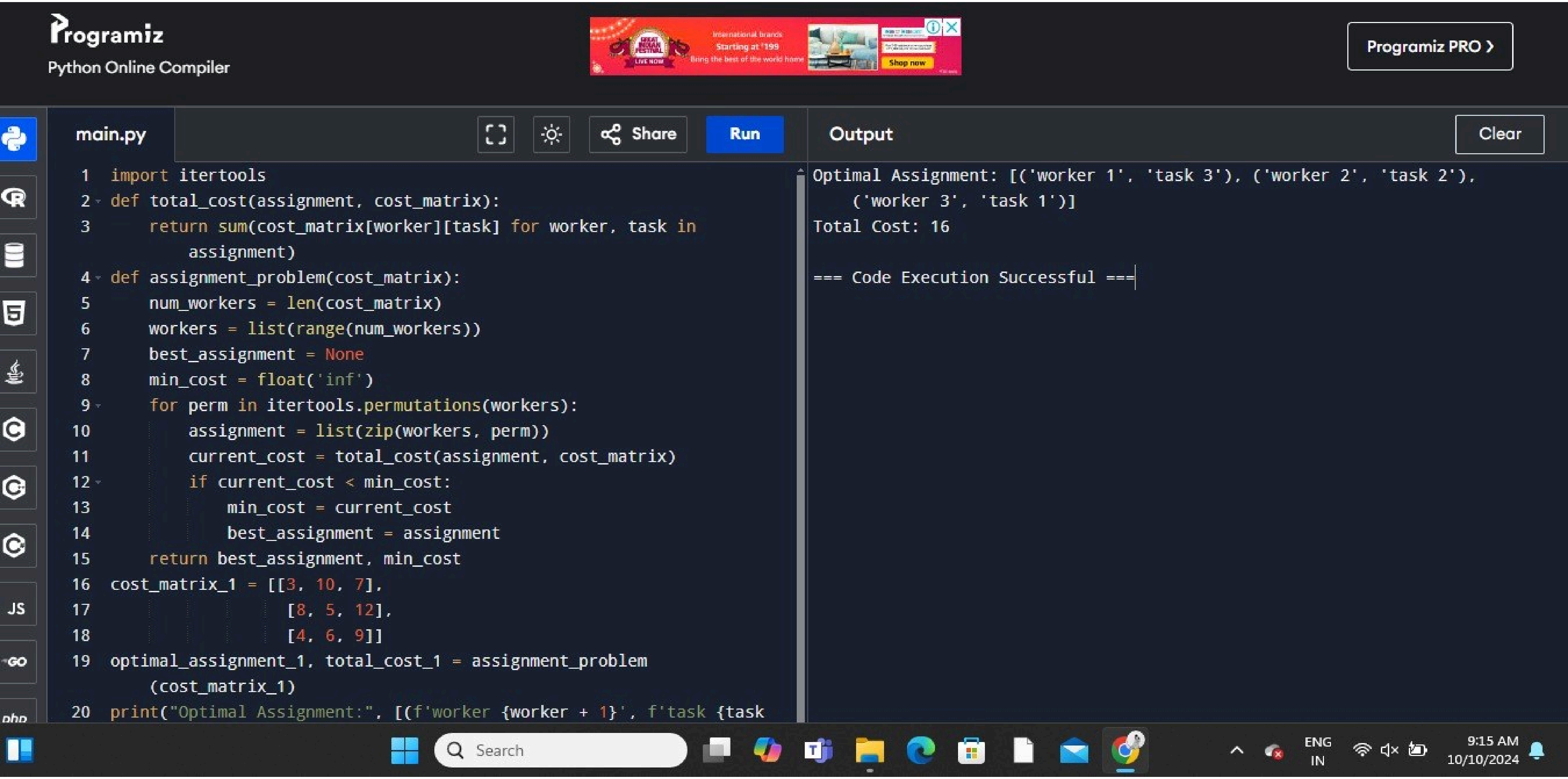
```python
import itertools
import math
def calculate_distance(point1, point2):
    return math.sqrt((point1[0] - point2[0]) ** 2 + (point1[1] - point2[1]) ** 2)
def total_distance(path):
    return sum(calculate_distance(path[i], path[i + 1]) for i in range(len(path) - 1))
def find_shortest_path(cities):
    shortest_distance = float('inf')
    shortest_path = []
    for perm in itertools.permutations(cities):
        current_path = list(perm) + [perm[0]]
        current_distance = total_distance(current_path)
        if current_distance < shortest_distance:
            shortest_distance = current_distance
            shortest_path = current_path
    return shortest_distance, shortest_path
test_cases = [
    [(1, 2), (4, 5), (7, 1), (3, 6)],
    [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]
]
```

```
Test Case 1:
Shortest Distance: 16.969112047670894
Shortest Path: [(1, 2), (7, 1), (4, 5), (3, 6), (1, 2)]
Test Case 2:
Shortest Distance: 23.12995011084934
Shortest Path: [(2, 4), (6, 3), (8, 1), (5, 9), (1, 7), (2, 4)]

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



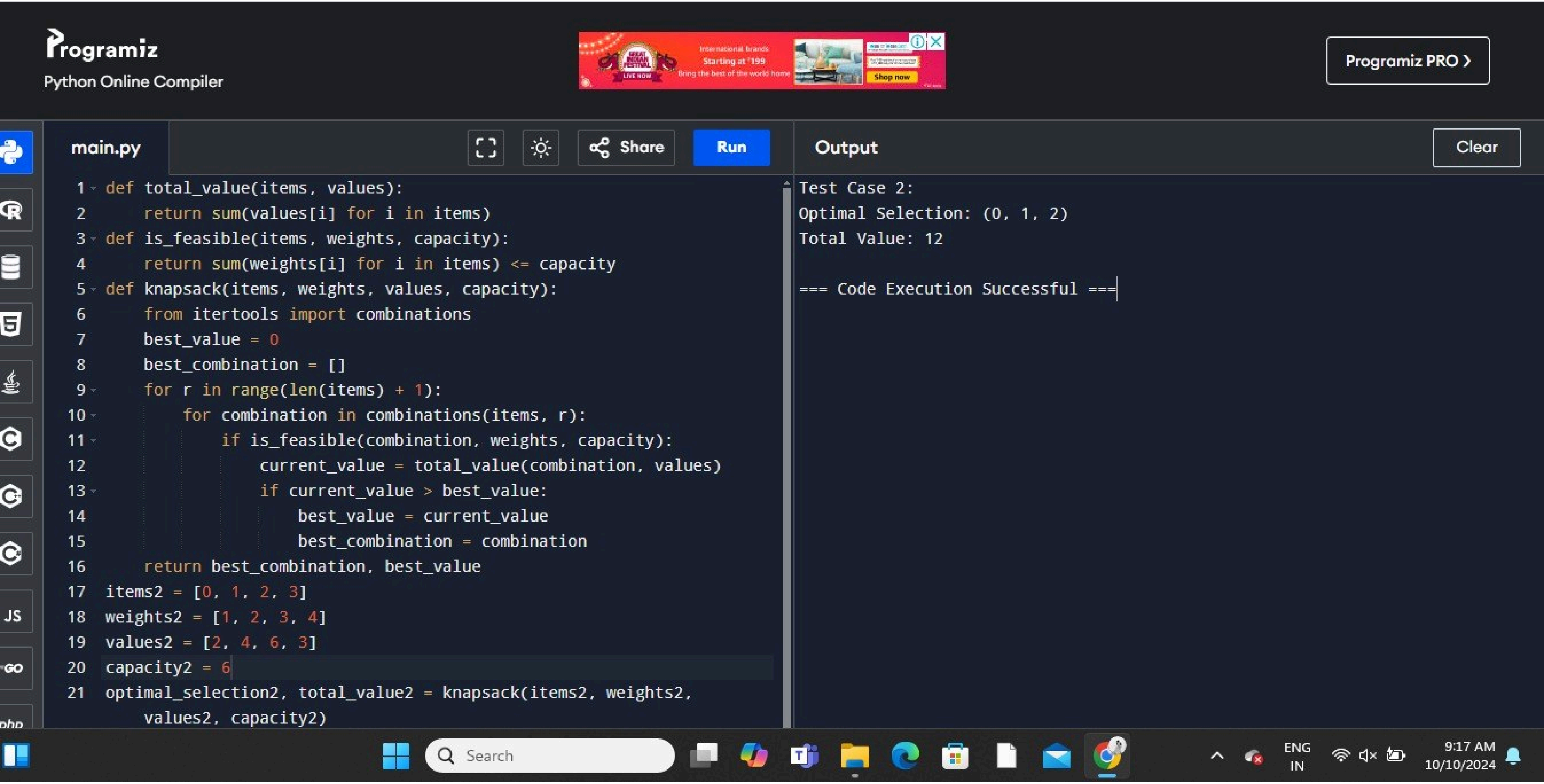
```python
import itertools
def total_cost(assignment, cost_matrix):
    return sum(cost_matrix[worker][task] for worker, task in assignment)
def assignment_problem(cost_matrix):
    num_workers = len(cost_matrix)
    workers = list(range(num_workers))
    best_assignment = None
    min_cost = float('inf')
    for perm in itertools.permutations(workers):
        assignment = list(zip(workers, perm))
        current_cost = total_cost(assignment, cost_matrix)
        if current_cost < min_cost:
            min_cost = current_cost
            best_assignment = assignment
    return best_assignment, min_cost
cost_matrix_1 = [[3, 10, 7],
                 [8, 5, 12],
                 [4, 6, 9]]
optimal_assignment_1, total_cost_1 = assignment_problem(cost_matrix_1)
print("Optimal Assignment:", [(f'worker {worker + 1}', f'task {task
```

```
Optimal Assignment: [('worker 1', 'task 3'), ('worker 2', 'task 2'), ('worker 3', 'task 1')]
Total Cost: 16

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



```python
import itertools
def total_cost(assignment, cost_matrix):
    return sum(cost_matrix[worker][task] for worker, task in assignment)
def assignment_problem(cost_matrix):
    num_workers = len(cost_matrix)
    workers = list(range(num_workers))
    best_assignment = None
    min_cost = float('inf')
    for perm in itertools.permutations(workers):
        assignment = list(zip(workers, perm))
        current_cost = total_cost(assignment, cost_matrix)
        if current_cost < min_cost:
            min_cost = current_cost
            best_assignment = assignment
    return best_assignment, min_cost
cost_matrix_1 = [[3, 10, 7],
                [8, 5, 12],
                [4, 6, 9]]
optimal_assignment_1, total_cost_1 = assignment_problem(cost_matrix_1)
print("Optimal Assignment:", [(f'worker {worker + 1}', f'task {task
```

```
Optimal Assignment: [('worker 1', 'task 3'), ('worker 2', 'task 2'),
    ('worker 3', 'task 1')]
Total Cost: 16

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



```python
def total_value(items, values):
    return sum(values[i] for i in items)
def is_feasible(items, weights, capacity):
    return sum(weights[i] for i in items) <= capacity
def knapsack(items, weights, values, capacity):
    from itertools import combinations
    best_value = 0
    best_combination = []
    for r in range(len(items) + 1):
        for combination in combinations(items, r):
            if is_feasible(combination, weights, capacity):
                current_value = total_value(combination, values)
                if current_value > best_value:
                    best_value = current_value
                    best_combination = combination
    return best_combination, best_value
items2 = [0, 1, 2, 3]
weights2 = [1, 2, 3, 4]
values2 = [2, 4, 6, 3]
capacity2 = 6
optimal_selection2, total_value2 = knapsack(items2, weights2, values2, capacity2)
```

```
Test Case 2:
Optimal Selection: (0, 1, 2)
Total Value: 12

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