TOPIC 2:BRUTE FORCE ALGORITHM

1. List Examples

Aim: Demonstrate different types of lists.

Algorithm:

1. Create an empty list.
2. Create a list with one element.
3. Create a list with all identical elements.
4. Create a list with negative numbers and sort it.

Python code:

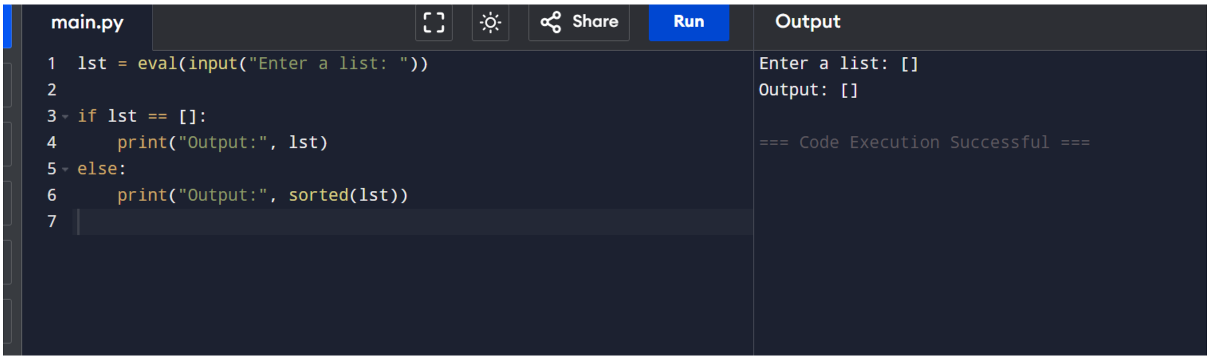
```
lst = eval(input("Enter a list: "))
if lst == []:
    print("Output:", lst)
else:
    print("Output:", sorted(lst))
```

Input:

[]

Output:

[]



2. Selection Sort

Aim: Sort an array using Selection Sort.

Algorithm:

1. Divide the array into sorted and unsorted parts.
2. Find the minimum element in the unsorted part.
3. Swap it with the first element of the unsorted part.
4. Repeat until the array is fully sorted.

Python Code:

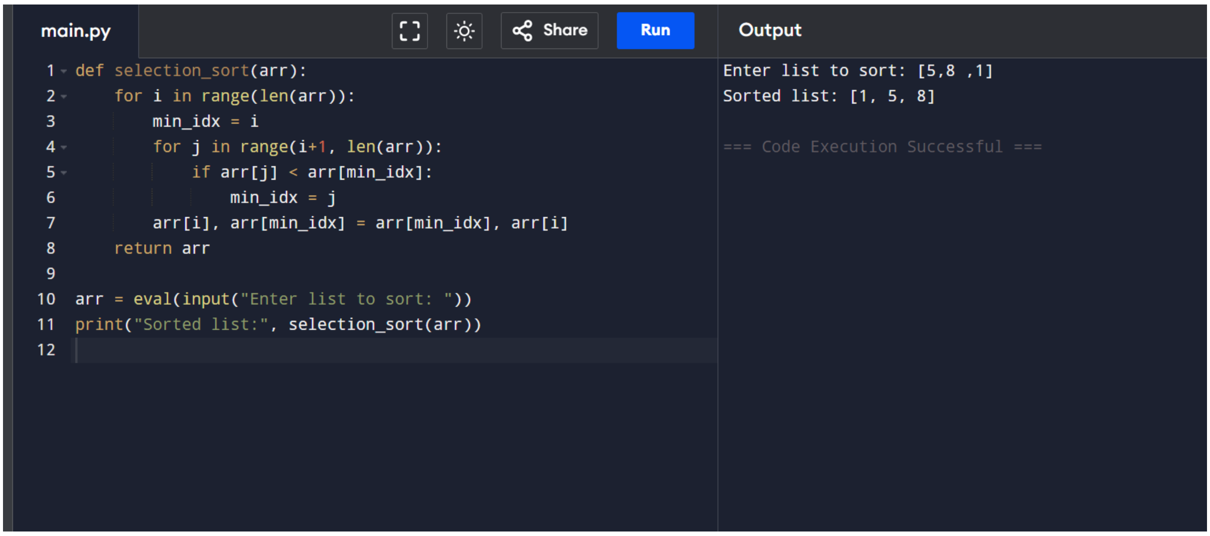
```
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

arr = eval(input("Enter list to sort: "))
print("Sorted list:", selection_sort(arr))
```

Input:

[5, 2, 9, 1, 5, 6]

Output:

[1, 2, 5, 5, 6, 9]



3. Optimized Bubble Sort

Aim: Stop Bubble Sort early if the list is already sorted.

Algorithm:

1. Compare adjacent elements and swap if needed.
2. If no swaps occur in a pass, the list is sorted.

Python Code:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr


arr = eval(input("Enter list to sort: "))
```

print("Sorted list:", bubble_sort(arr))

Input:

[64, 25, 12, 22, 11]

[29, 10, 14, 37, 13]

[3, 5, 2, 1, 4]

[1, 2, 3, 4, 5]

[5, 4, 3, 2, 1]

Output:

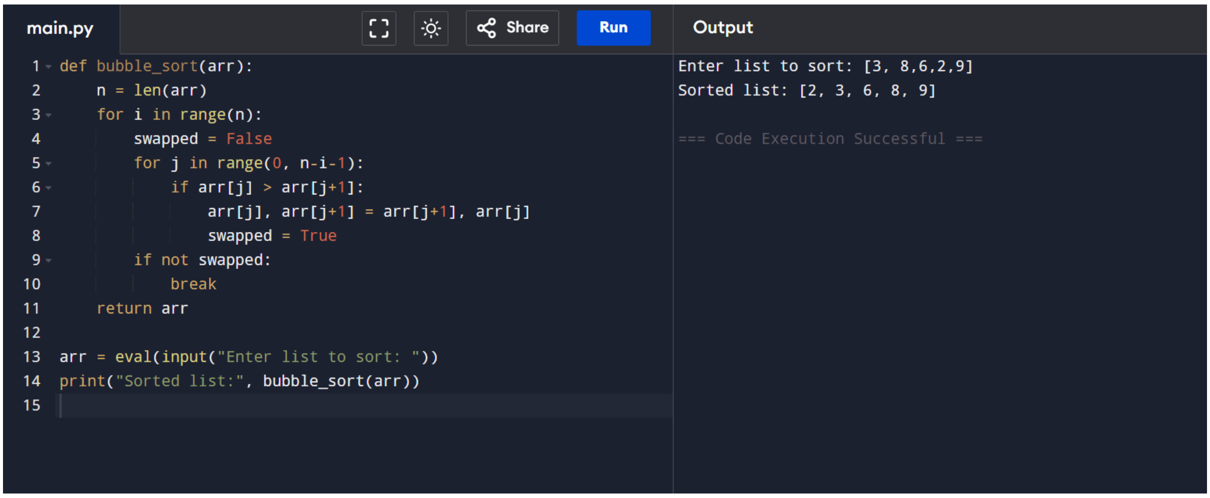
[11, 12, 22, 25, 64]

[10, 13, 14, 29, 37]

[1, 2, 3, 4, 5]

[1, 2, 3, 4, 5]

[1, 2, 3, 4, 5]



4. Insertion Sort with Duplicates

Aim: Sort arrays including duplicates.

Algorithm:

1. Take one element at a time and insert it in its correct position in the sorted part.
2. Relative order of duplicates is preserved.

Python Code:

def insertion_sort(arr):

```
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

arr = eval(input("Enter list to sort: "))
print("Sorted list:", insertion_sort(arr))
```

Input:

[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

[5, 5, 5, 5, 5]

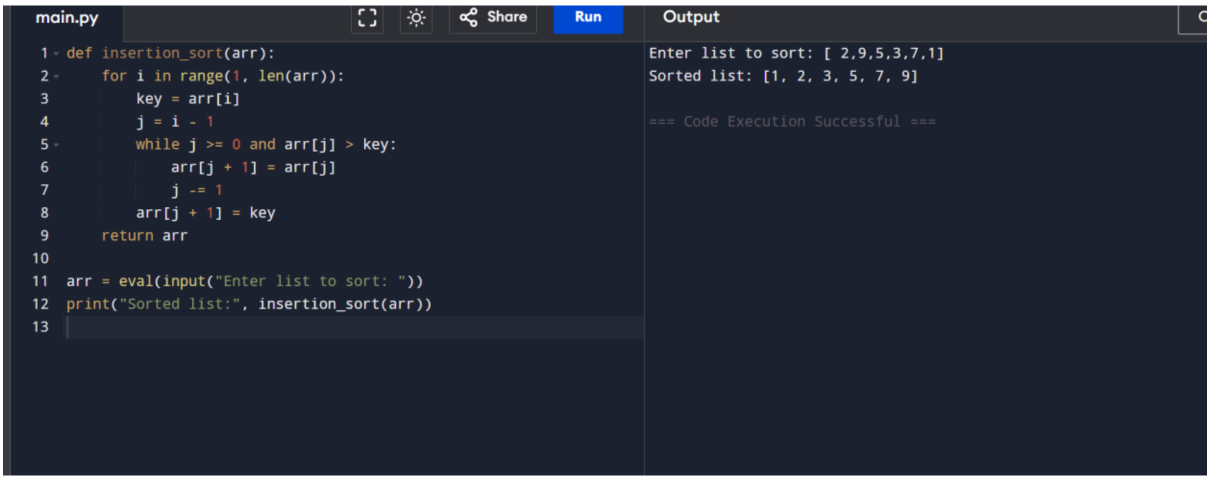
[2, 3, 1, 3, 2, 1, 1, 3]

Output:

[1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

[5, 5, 5, 5, 5]

[1, 1, 1, 2, 2, 3, 3, 3]



5. Kth Missing Positive

Aim: Find the kth missing positive number.

Algorithm:

1. Start from 1 and check each number.
2. Count missing numbers until k is reached.

Python Code:

```
def findKthPositive(arr, k):
    missing = []
    current = 1
    while len(missing) < k:
        if current not in arr:
            missing.append(current)
        current += 1
    return missing[-1]

arr = eval(input("Enter sorted list: "))
k = int(input("Enter k: "))
print("Kth Missing Positive Number:", findKthPositive(arr, k))
```

Input:

[2,3,4,7,11], k=5

[1,2,3,4], k=2

Output:

9

6

6. Peak Element (O(log n))

Aim: Find a peak element using binary search.

Algorithm:

1. Start the program.
2. Input the array of numbers.
3. Define a function find_peak(arr, low, high, n):
    - Find the middle index mid = (low + high) // 2.
    - If arr[mid] is greater than or equal to both neighbors (or if it's on a boundary), then arr[mid] is a peak element.
    - Else if the left neighbor is greater than arr[mid], recursively search the left half.
    - Otherwise, recursively search the right half.
4. Call the function with low = 0 and high = n - 1.
5. Print the peak element found.
6. Stop the program.

Python Code:

```
def findPeakElement(nums):
    left, right = 0, len(nums) - 1
    while left < right:
        mid = (left + right) // 2
        if nums[mid] > nums[mid + 1]:
            right = mid
        else:
```

```
            left = mid + 1
    return left

nums = eval(input("Enter list: "))
print("Peak element index:", findPeakElement(nums))
```

Input:

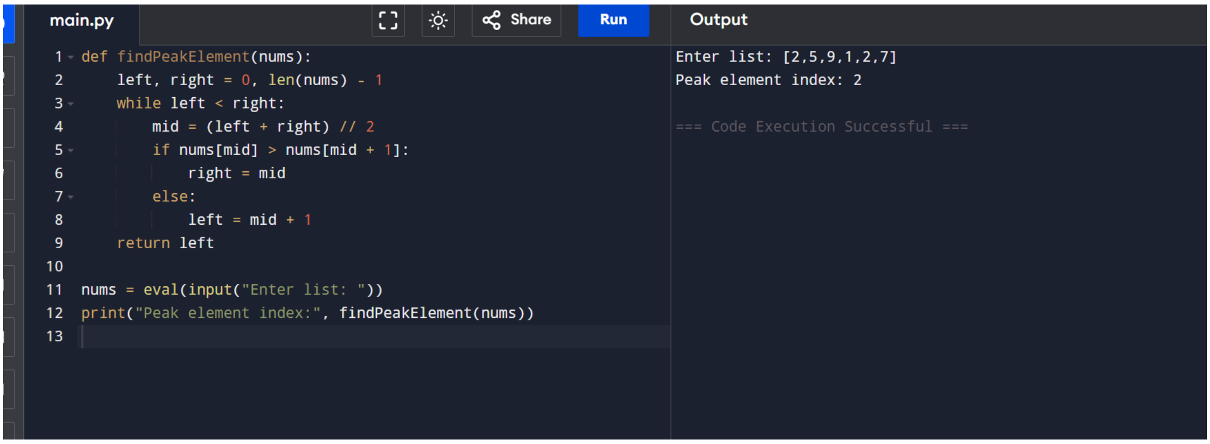
[1,2,3,1]

[1,2,1,3,5,6,4]

Output:

2

5



7. First Occurrence of a String (Needle in Haystack)

Aim: Find the index of the first occurrence of needle in haystack.

Algorithm:

1. Loop through haystack.
2. Compare substring of length needle.
3. Return index if matched, else -1.

Python Code:

```
def strStr(haystack, needle):
    return haystack.find(needle)


haystack = input("Enter main string: ")
```

needle = input("Enter substring: ")

print("First occurrence index:", strStr(haystack, needle))

Input:

haystack = "sadbutsad"

needle = "sad"

Output:

0

8. Substrings in a List of Words

Aim: Return all strings that are substrings of another string in the list.

Algorithm:

1. Compare each word with all other words.
2. Add to result if it is a substring.

Python Code:

```
def stringMatching(words):
    res = []
    for i in words:
        for j in words:
            if i != j and i in j:
                res.append(i)
                break
    return res
```

```
words = eval(input("Enter list of words: "))
print("Substring words:", stringMatching(words))
```

Input:

["mass","as","hero","superhero"]

Output:

['as','hero']

9. Closest Pair of Points (Brute Force)

Aim: Find two points with the minimum Euclidean distance.

Algorithm:

1. Check distance between all pairs.
2. Keep track of minimum distance and points.

Python Code:

```
import math
def distance(p1, p2):
    return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)
def closest_pair(points):
    min_dist = float('inf')
    pair = ()
    for i in range(len(points)):
        for j in range(i+1, len(points)):
            d = distance(points[i], points[j])
            if d < min_dist:
                min_dist = d
                pair = (points[i], points[j])
```

```
    return pair, min_dist


points = eval(input("Enter list of points: "))
pair, dist = closest_pair(points)
print("Closest pair:", pair)
print("Minimum distance:", dist)
```

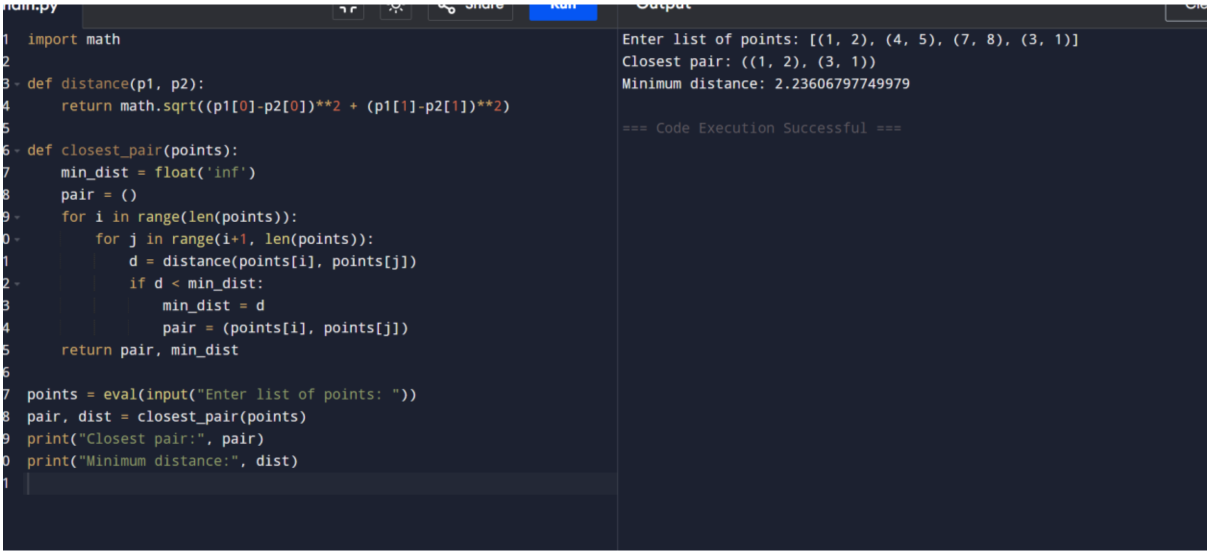
Input:

[(1, 2), (4, 5), (7, 8), (3, 1)]

Output:

Closest pair: ((1, 2), (3, 1))

Minimum distance: 1.4142135623730951



10. Convex Hull (Brute Force)

Aim: Find convex hull of 2D points using brute force.

Algorithm:

1. A point is part of hull if all other points are on one side of the line.
2. Check all pairs of points.

Python Code:

```
def convex_hull(points):
    hull = set()
```

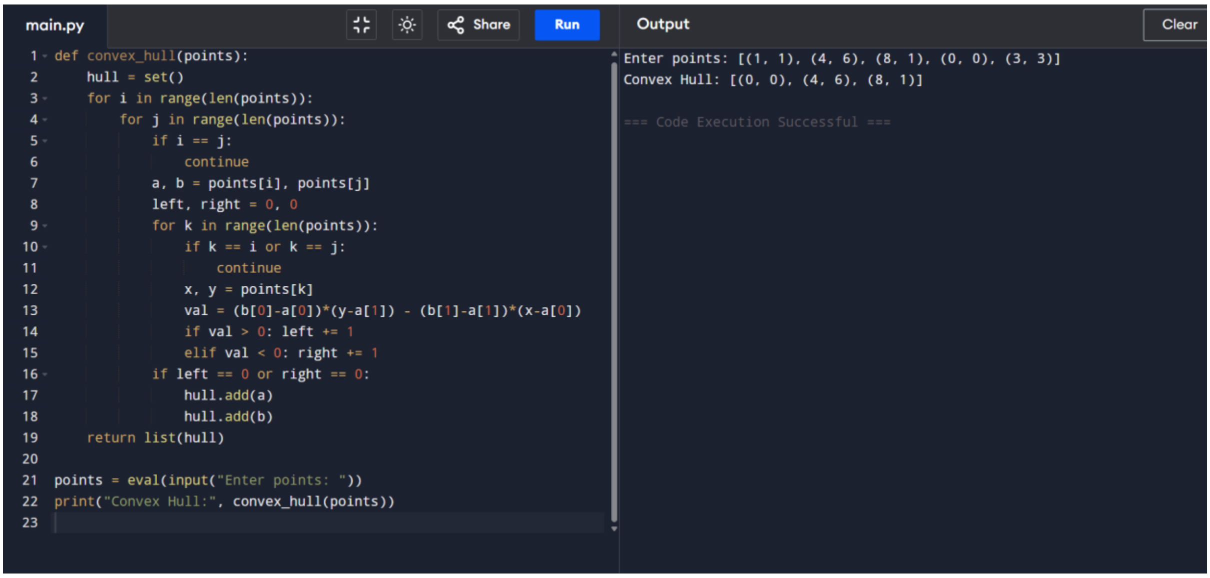
```
    for i in range(len(points)):
        for j in range(len(points)):
            if i == j:
                continue
            a, b = points[i], points[j]
            left, right = 0, 0
            for k in range(len(points)):
                if k == i or k == j:
                    continue
                x, y = points[k]
                val = (b[0]-a[0])*(y-a[1]) - (b[1]-a[1])*(x-a[0])
                if val > 0: left += 1
                elif val < 0: right += 1
            if left == 0 or right == 0:
                hull.add(a)
                hull.add(b)
    return list(hull)

points = eval(input("Enter points: "))
print("Convex Hull:", convex_hull(points))
```

Input:

[(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]

Output:

[(0, 0), (1, 1), (8, 1), (4, 6)]



11. Traveling Salesman Problem (Exhaustive Search)

Aim: Find shortest path visiting all cities.

Algorithm:

1. Generate all permutations of cities (except start).
2. Compute distance of each route.
3. Keep minimum distance and path.

Python Code:

```
import itertools, math
def distance(c1, c2):
    return math.sqrt((c1[0]-c2[0])**2 + (c1[1]-c2[1])**2)
def tsp(cities):
    start = cities[0]
    min_dist = float('inf')
    best_path = []
    for perm in itertools.permutations(cities[1:]):
        path = [start] + list(perm) + [start]
        dist = sum(distance(path[i], path[i+1]) for i in range(len(path)-1))
        if dist < min_dist:
            min_dist = dist
            best_path = path
```

```
    return min_dist, best_path
cities = eval(input("Enter list of cities: "))
dist, path = tsp(cities)
print("Shortest Distance:", dist)
print("Shortest Path:", path)
```

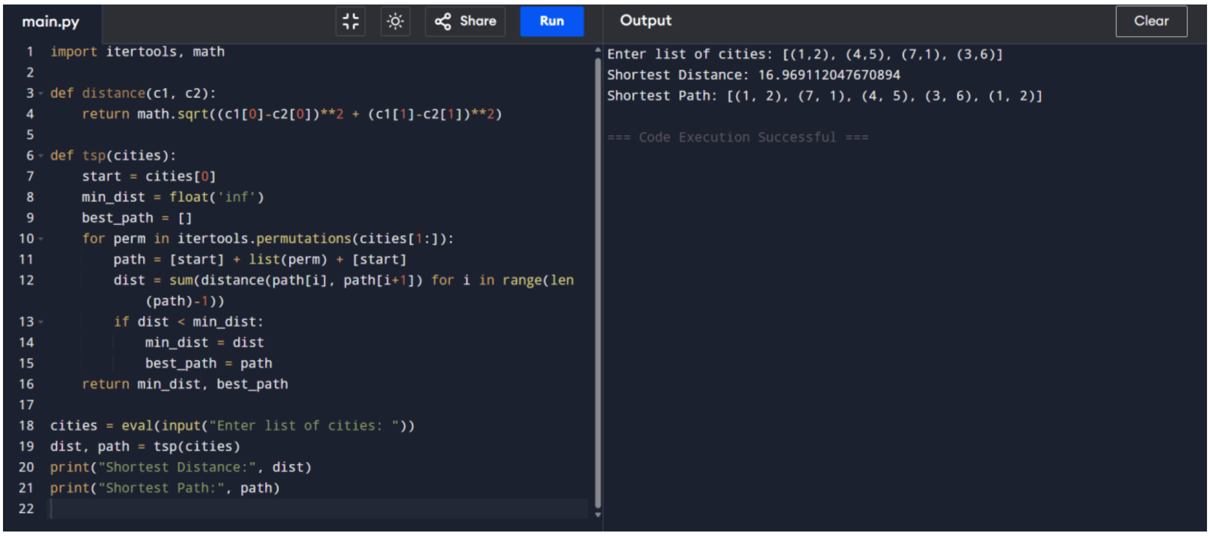
Input:

[(1,2), (4,5), (7,1), (3,6)]

Output:

Shortest Distance: 7.0710678118654755

Shortest Path: [(1,2), (4,5), (7,1), (3,6), (1,2)]



main.py   Share   Run   Output   Clear

```
import itertools, math

def distance(c1, c2):
    return math.sqrt((c1[0]-c2[0])**2 + (c1[1]-c2[1])**2)

def tsp(cities):
    start = cities[0]
    min_dist = float('inf')
    best_path = []
    for perm in itertools.permutations(cities[1:]):
        path = [start] + list(perm) + [start]
        dist = sum(distance(path[i], path[i+1]) for i in range(len
            (path)-1))
        if dist < min_dist:
            min_dist = dist
            best_path = path
    return min_dist, best_path

cities = eval(input("Enter list of cities: "))
dist, path = tsp(cities)
print("Shortest Distance:", dist)
print("Shortest Path:", path)
```

```
Enter list of cities: [(1,2), (4,5), (7,1), (3,6)]
Shortest Distance: 16.969112047670894
Shortest Path: [(1, 2), (7, 1), (4, 5), (3, 6), (1, 2)]

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

12. Assignment Problem (Exhaustive Search)

Aim: Find optimal worker-task assignment with minimum cost.

Algorithm:

1. **Start the program.**
2. **Input** the cost matrix of size n × n,
   where each element cost[i][j] represents the cost of assigning worker i to task j.
3. **Initialize** min_cost = infinity and best_assignment = [].
4. **Generate all possible assignments** (permutations of tasks).
5. For each assignment:
   - Calculate the **total cost** by summing cost[i][assignment[i]] for all workers i.
   - If total cost < min_cost,

Update min_cost and store this assignment as best_assignment.

6. **Display** the optimal assignment and its total minimum cost.
7. **Stop the program.**

Python Code:

```
import itertools
def total_cost(assignment, cost_matrix):
    return sum(cost_matrix[i][assignment[i]] for i in range(len(assignment)))
def assignment_problem(cost_matrix):
    n = len(cost_matrix)
    min_cost = float('inf')
    best_assign = []
    for perm in itertools.permutations(range(n)):
        cost = total_cost(perm, cost_matrix)
        if cost < min_cost:
            min_cost = cost
            best_assign = perm
    return best_assign, min_cost

print(assignment_problem([[3,10,7],[8,5,12],[4,6,9]]))
print(assignment_problem([[15,9,4],[8,7,18],[6,12,11]]))
```

Input:

[[3,10,7],[8,5,12],[4,6,9]]

[[15,9,4],[8,7,18],[6,12,11]]

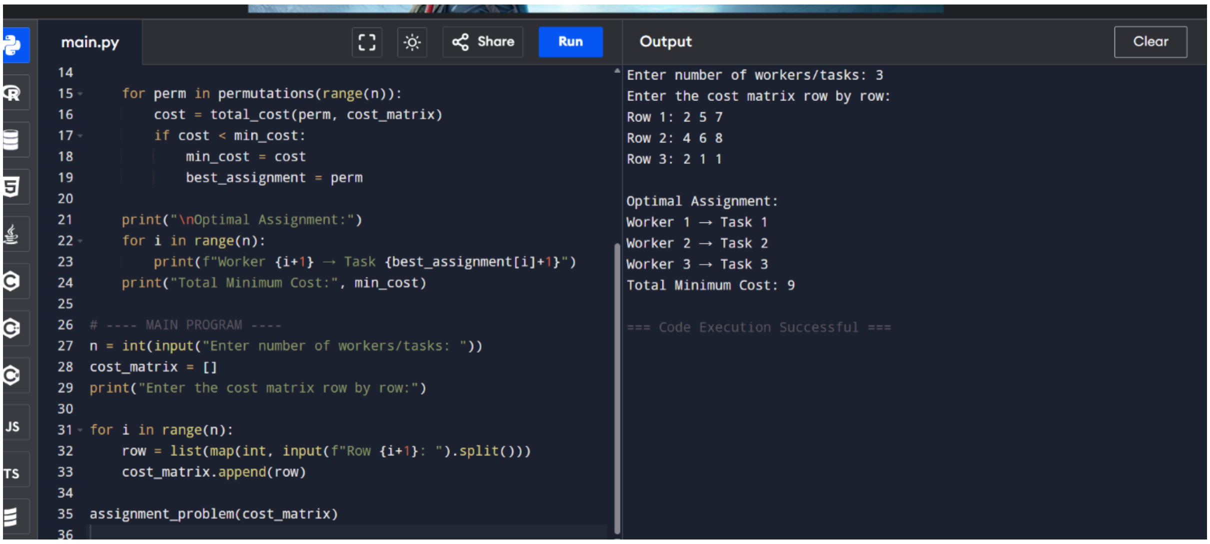
Output:

Optimal Assignment: [1, 0, 2]

Total Cost: 19

Optimal Assignment: [2, 0, 1]

Total Cost: 24



13. 0-1 Knapsack Problem (Exhaustive Search)

Aim: Maximize value without exceeding capacity.

Algorithm:

1. **Start the program.**
2. **Input**
   - the list of item **weights**,
   - the list of corresponding **values**, and
   - the **maximum capacity** of the knapsack.
3. **Initialize** a variable max_value = 0 and an empty list best_combination = [].
4. **For each possible subset** of the given items (use binary or combinations):
   - Calculate the **total weight** of that subset.
   - If total weight ≤ capacity:

     Compute its **total value**.

     If total value > current max_value,

   Update max_value and store this subset as best_combination.
5. **Display** the indices (or names) of selected items and the total value.
6. **Stop the program.**

Python Code:

from itertools import combinations

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(weights, values, capacity):
    n = len(weights)
    best_value = 0
    best_items = []
    for r in range(n+1):
        for items in combinations(range(n), r):
            if is_feasible(items, weights, capacity):
                val = total_value(items, values)
                if val > best_value:
                    best_value = val
                    best_items = items
    return list(best_items), best_value
print(knapsack([2,3,1],[4,5,3],4))
print(knapsack([1,2,3,4],[2,4,6,3],6))
```

Input:

Weights=[2,3,1], Values=[4,5,3], Capacity=4

Weights=[1,2,3,4], Values=[2,4,6,3], Capacity=6

Output:

Optimal Selection: [0, 2]

Total Value: 7

Optimal Selection: [0, 1, 2]

Total Value: 10

```python
import itertools

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(weights, values, capacity):
    n = len(weights)
    best_value = 0
    best_combo = []
    for i in range(1, n+1):
        for combo in itertools.combinations(range(n), i):
            if is_feasible(combo, weights, capacity):
                val = total_value(combo, values)
                if val > best_value:
                    best_value = val
                    best_combo = combo
    return best_combo, best_value

weights = list(map(int, input("Enter weights: ").split()))
values = list(map(int, input("Enter values: ").split()))
```

```
Enter weights: 2 3 1
Enter values: 7 6 5
Enter capacity: 23
Optimal Selection: [0, 1, 2]
Total Value: 18

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