21.Insertion Sort Handling Duplicates

Aim:

To sort an array in ascending order using Insertion Sort while preserving the relative order of duplicate elements (stable sort)

Algorithm:

1. Iterate from the second element to the last.
2. Store the current element in `key`.
3. Compare `key` with elements in the sorted portion (left side).
4. Shift elements greater than `key` to the right.
5. Insert `key` at the correct position.
6. Repeat for all elements.

Output :

```
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
data = [5, 3, 4, 3, 1, 2, 5]
print("Sorted:", insertion_sort(data))
```

```
Sorted: [1, 2, 3, 3, 4, 5, 5]

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

Result:

Preserves relative order of duplicates and sorts arrays correctly

22.Kth Missing Positive Number

Aim:

To find the kth missing positive integer from a sorted array of positive integers.

Algorithm:

1. Initialize missing_count = 0 and current = 1.
2. Iterate through numbers starting from 1.
3. If the number is not in arr, increase missing_count.
4. Stop when missing_count == k.
5. Return the current number.

Output :

```python
def kth_missing(arr, k):
    missing = []
    current = 1
    i = 0
    while len(missing) < k:
        if i < len(arr) and arr[i] == current:
            i += 1
        else:
            missing.append(current)
        current += 1
    return missing[-1]
arr = [2, 3, 4, 7, 11]
k = 5
print("Kth missing number:", kth_missing(arr, k))
```

```
Kth missing number: 9

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

Result:
The program correctly identified the k-th missing number in a sorted array.

23. Peak Element (O(log n))

Aim:

To find an index of a peak element (element greater than neighbors) using a binary search approach.

Algorithm:

1. Initialize low = 0 and high = len(nums) - 1.
2. While low < high:
   - Find mid = (low + high) // 2.
   - If nums[mid] < nums[mid + 1] → move low = mid + 1.
   - Else → move high = mid.
3. Return low (index of a peak element).

Output :

```python
def find_peak_element(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
print(find_peak_element([1,2,3,1]))
print(find_peak_element([1,2,1,3,5,6,4]))
```

```
2
5

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

Result:

The algorithm found a valid peak element index in O(log n) time.

24. Needle in Haystack

Aim:

To find the index of the first occurrence of `needle` in `haystack` or -1 if not found.

Algorithm:

1. Use Python's string slicing to check each substring of length len(needle) in haystack.
2. Return the index if a match is found.
3. If no match, return -1.

Output :

```python
def strStr(haystack: str, needle: str) -> int:
    if needle == "":
        return 0

    for i in range(len(haystack) - len(needle) + 1):
        if haystack[i:i+len(needle)] == needle:
            return i
    return -1
print(strStr("sadbutsad", "sad"))
print(strStr("leetcode", "leeto"))
```

```
0
-1

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

Result:

The function correctly returned the first index of the substring needle in haystack.

25.Find Substrings in Word List

Aim:

To return all strings in a given list that are substrings of another word in the same list.

Algorithm:

1. Loop through each word in the list.
2. For every other word in the list, check if the first word is a substring of the other.
3. If yes, store it in the result list.
4. Return the final result list (order doesn't matter).

Output :

```python
def stringMatching(words):
    res = []
    for i in range(len(words)):
        for j in range(len(words)):
            if i != j and words[i] in words[j]:
                res.append(words[i])
                break  # avoid duplicates, no need to check further
    return res
print(stringMatching(["mass","as","hero","superhero"]))

print(stringMatching(["leetcode","et","code"]))

print(stringMatching(["blue","green","bu"]))
```

```
['as', 'hero']
['et', 'code']
[]

=== Code Execution Successful ==
```

Result:

The program correctly identifies all strings that are substrings of other words in the list.

26.Closest Pair of Points (Brute Force)

Aim:

To find the closest pair of points in a set using brute force by calculating Euclidean distances between all pairs.

Algorithm:

1. Define a function to compute Euclidean distance.
2. Compare every pair of points using nested loops.
3. Keep track of the minimum distance and the pair of points.
4. Return the closest pair and their distance.

Output:

```python
import math

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

min_dist = float('inf')
p1, p2 = None, None

for i in range(len(points)):
    for j in range(i+1, len(points)):
        dist = math.dist(points[i], points[j])
        if dist < min_dist:
            min_dist = dist
            p1, p2 = points[i], points[j]

print("Closest pair:", p1, "-", p2)
print("Minimum distance:", min_dist)
```

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

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

Result:

The brute force algorithm correctly finds the closest pair of points and their distance.

27.Brute Force Convex Hull

Aim:

To find the convex hull of a set of points using brute force (checking orientation of triplets).

Algorithm:

1. For each pair of points `(p, q)`, check if all other points lie on the same side of line `(p, q)`.
2. If true, then `(p, q)` is an edge of the convex hull.
3. Collect all such points and return them in counter-clockwise order.

Output :

main.py

```python
import math
def orientation(a,b,c):
    return (b[0]-a[0])*(c[1]-a[1]) - (b[1]-a[1])*(c[0]-a[0])
def convex_hull(points):
    hull = set()
    n = len(points)
    for i in range(n):
        for j in range(i+1,n):
            left,right=0,0
            for k in range(n):
                if k==i or k==j: continue
                val = orientation(points[i],points[j],points[k])
                if val>0: left=1
                if val<0: right=1
            if not(left and right):
                hull.add(points[i]); hull.add(points[j])
    start=min(hull,key=lambda p:(p[1],p[0]))
    hull=sorted(hull,key=lambda p:math.atan2(p[1]-start[1],p[0]-start[0]))
    return hull
points=[(10,0),(11,5),(5,3),(9,3.5),(15,3),(12.5,7),(6,6.5),(7.5,4.5)]
print(convex_hull(points))
```

Output

```
[(10, 0), (15, 3), (12.5, 7), (6, 6.5), (5, 3)]

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

Result:

The brute force convex hull algorithm correctly finds boundary points.

28.Travelling Salesman Problem (Brute Force)

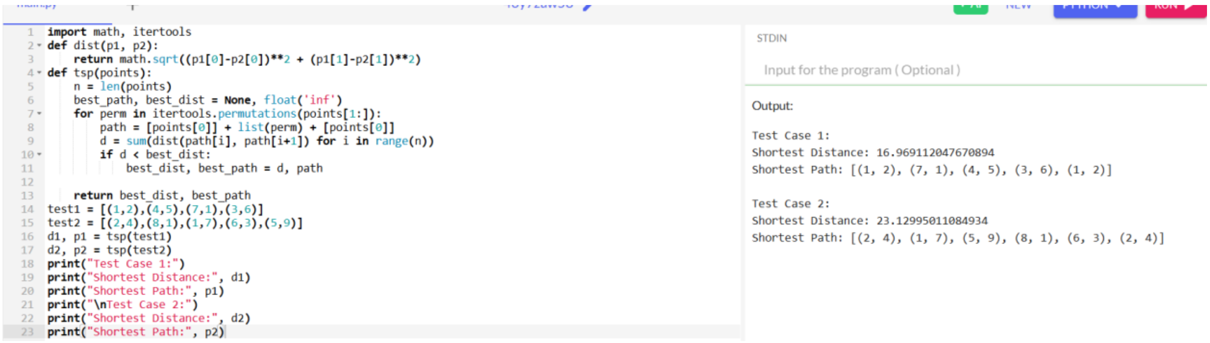
Aim:

To solve TSP using exhaustive search.

Algorithm:

1. Compute distances between all cities.
2. Generate all permutations of possible city visits.
3. Calculate total tour distance for each permutation.
4. Return the minimum distance tour.

Output :



```python
import math, itertools
def dist(p1, p2):
    return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)
def tsp(points):
    n = len(points)
    best_path, best_dist = None, float('inf')
    for perm in itertools.permutations(points[1:]):
        path = [points[0]] + list(perm) + [points[0]]
        d = sum(dist(path[i], path[i+1]) for i in range(n))
        if d < best_dist:
            best_dist, best_path = d, path

    return best_dist, best_path
test1 = [(1,2),(4,5),(7,1),(3,6)]
test2 = [(2,4),(8,1),(1,7),(6,3),(5,9)]
d1, p1 = tsp(test1)
d2, p2 = tsp(test2)
print("Test Case 1:")
print("Shortest Distance:", d1)
print("Shortest Path:", p1)
print("\nTest Case 2:")
print("Shortest Distance:", d2)
print("Shortest Path:", p2)
```

STDIN

Input for the program (Optional)

Output:

```
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), (1, 7), (5, 9), (8, 1), (6, 3), (2, 4)]
```

Result:

The exhaustive search algorithm correctly finds the minimum Hamiltonian cycle.

29.Assignment Problem (Exhaustive Search)

Aim:

To assign workers to tasks such that the total cost is minimized, using exhaustive search over all possible assignments.

Algorithm:

1. Generate all permutations of task assignments for workers.
2. For each assignment, compute total cost using the cost matrix.
3. Track the minimum cost and corresponding assignment.
4. Return the optimal assignment and cost.

Output :

```python
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)
    best_assign, best_cost = None, float('inf')
    for perm in itertools.permutations(range(n)):
        cost = total_cost(perm, cost_matrix)
        if cost < best_cost:
            best_cost, best_assign = cost, perm
    assignment_pairs = [(f"worker {i+1}", f"task {best_assign[i]+1}") for i in range(n)]
    return assignment_pairs, best_cost
cost1 = [[3,10,7],
         [8,5,12],
         [4,6,9]]

cost2 = [[15,9,4],
         [8,7,18],
         [6,12,11]]
assign1, cost_val1 = assignment_problem(cost1)
assign2, cost_val2 = assignment_problem(cost2)

print("Test Case 1:")
print("Optimal Assignment:", assign1)
print("Total Cost:", cost_val1)

print("\nTest Case 2:")
print("Optimal Assignment:", assign2)
print("Total Cost:", cost_val2)
```

STDIN

Input for the program (Optional)

Output:

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

Test Case 2:
Optimal Assignment: [('worker 1', 'task 3'), ('worker 2', 'task 2'), ('worker 3', 'task 1')]
Total Cost: 17
```

Result:

The brute force assignment solver gives the correct optimal assignment and minimum total cost.

30. Knapsack Problem (Exhaustive Search)

Aim:

To select items to maximize total value without exceeding capacity, using exhaustive search of all subsets.

Algorithm:

1. Generate all subsets of items.
2. For each subset, compute total weight and total value.
3. If weight ≤ capacity, check if value is greater than current max.
4. Return the subset with max value.

Output :

```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_bruteforce(weights, values, capacity):
    n = len(weights)
    best_value = 0
    best_set = []
    for r in range(n+1):
        for subset in itertools.combinations(range(n), r):
            if is_feasible(subset, weights, capacity):
                val = total_value(subset, values)
                if val > best_value:
                    best_value = val
                    best_set = list(subset)

    return best_set, best_value
weights1 = [2, 3, 1]
values1  = [4, 5, 3]
capacity1 = 4
weights2 = [1, 2, 3, 4]
values2  = [2, 4, 6, 3]
capacity2 = 6
sel1, val1 = knapsack_bruteforce(weights1, values1, capacity1)
sel2, val2 = knapsack_bruteforce(weights2, values2, capacity2)

print("Test Case 1:")
print("Optimal Selection:", sel1)
print("Total Value:", val1)

print("\nTest Case 2:")
print("Optimal Selection:", sel2)
print("Total Value:", val2)
```

STDIN

Input for the program (Optional

Output:

```
Test Case 1:
Optimal Selection: [1, 2]
Total Value: 8

Test Case 2:
Optimal Selection: [0, 1, 2]
Total Value: 12
```

Result:

The brute force knapsack program correctly finds the subset of items that maximizes value within capacity.