DESIGN AND ANALYSIS OF ALGORITHMS LAB
TOPIC 2: BRUTE FORCE (Questions 1–14)

Q1. Program to handle different types of lists

Aim: To sort different types of lists using brute force.

Procedure: Compare every element with all others and swap when required.

Code:

```
def brute_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(i+1, n):
            if arr[i] > arr[j]:
                arr[i], arr[j] = arr[j], arr[i]
    return arr

print(brute_sort([]))
print(brute_sort([1]))
print(brute_sort([7,7,7]))
print(brute_sort([-3,-1,-2]))
```

Output: Sorted lists are displayed.

Result: Different list types are sorted successfully.

Q2. Selection Sort

Aim: To implement Selection Sort algorithm.

Procedure: Select minimum element and place it at correct position.

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

print(selection_sort([5,2,9,1,5]))
```

Output: Sorted array is displayed.

Result: Selection sort implemented correctly.

Q3 & Q4. Optimized Bubble Sort

Aim: To optimize bubble sort using early termination.

Procedure: Stop the algorithm if no swaps occur in a pass.

Code:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(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

    print(bubble_sort([64,34,25,12,22,11,90]))
```

Output: Sorted array is printed.

Result: Optimized bubble sort reduces unnecessary iterations.

Q5. Insertion Sort with Duplicate Elements

Aim: To sort array containing duplicate elements.

Procedure: Insert each element into its correct position.

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

    print(insertion_sort([3,1,4,1,5]))
```

Output: Sorted array with duplicates is printed.

Result: Insertion sort handles duplicates correctly.

Q6. Kth Missing Positive Number

Aim: To find the kth missing positive integer.

Procedure: Traverse integers and count missing values.

Code:

```
    def find_kth_missing(arr, k):
        missing = 0
        current = 1
        i = 0
        while missing < k:
            if i < len(arr) and arr[i] == current:
                i += 1
            else:
                missing += 1
                if missing == k:
                    return current
            current += 1

    print(find_kth_missing([2,3,4,7,11], 5))
```

Output: Kth missing number is printed.

Result: Correct missing number is obtained.

Q7. Peak Element

Aim: To find a peak element using binary search.

Procedure: Compare mid element with its neighbors.

Code:

```
    def find_peak(nums):
        l, r = 0, len(nums)-1
        while l < r:
            mid = (l+r)//2
            if nums[mid] > nums[mid+1]:
                r = mid
```

```
            else:
                l = mid+1
        return l

    print(find_peak([1,2,3,1]))
```

Output: Peak index is displayed.

Result: Peak element found efficiently.

Q8. First Occurrence of Substring

Aim: To find the first occurrence of a substring.

Procedure: Compare substring with main string character by character.

Code:

```
    def strStr(haystack, needle):
        for i in range(len(haystack)-len(needle)+1):
            if haystack[i:i+len(needle)] == needle:
                return i
        return -1

    print(strStr("sadbutsad","sad"))
```

Output: Substring index printed.

Result: Substring found successfully.

Q9. Words that are Substrings of Another Word

Aim: To find words that are substrings of other words.

Procedure: Compare each word with others.

Code:

```
    def string_matching(words):
        result = []
        for i in range(len(words)):
            for j in range(len(words)):
                if i != j and words[i] in words[j]:
                    result.append(words[i])
                    break
        return result

    print(string_matching(["mass","as","hero","superhero"]))
```

Output: Substring words printed.

Result: Substring words identified correctly.

Q10. Closest Pair of Points

Aim: To find the closest pair of points.

Procedure: Compute distance between all point pairs.

Code:

```
    import math
    def closest_pair(points):
        min_dist = float('inf')
        pair = None
        for i in range(len(points)):
            for j in range(i+1, len(points)):
                d = math.dist(points[i], points[j])
                if d < min_dist:
                    min_dist = d
                    pair = (points[i], points[j])
        return pair, min_dist
    print(closest_pair([(1,2),(4,5),(3,1)]))
```

Output: Closest pair is displayed.

Result: Closest points found correctly.

Q11. Convex Hull using Brute Force

Aim: To find convex hull of given points.

Procedure: Check orientation of all point pairs.

Code:

```
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):
    hull = set()
    n = len(points)
    for i in range(n):
        for j in range(i+1, n):
            pos = neg = 0
            for k in range(n):
                val = orientation(points[i], points[j], points[k])
                if val > 0:
                    pos += 1
                elif val < 0:
                    neg += 1
            if pos == 0 or neg == 0:
                hull.add(points[i])
                hull.add(points[j])
    return list(hull)

print(convex_hull([(0,0),(1,1),(2,2),(0,2),(2,0)]))
```

Output: Convex hull points printed.

Result: Convex hull determined successfully.

Q12. Travelling Salesman Problem

Aim: To solve TSP using brute force approach.

Procedure: Generate all possible routes and calculate cost.

Code:

```
import itertools, math
def tsp(cities):
    min_dist = float('inf')
    best_path = None
    for perm in itertools.permutations(cities):
        dist = sum(math.dist(perm[i], perm[i+1]) for i in range(len(perm)-1))
        if dist < min_dist:
            min_dist = dist
            best_path = perm
    return min_dist, best_path

print(tsp([(0,0),(1,1),(2,0)]))
```

Output: Minimum distance printed.

Result: Optimal tour found.

Q13. Assignment Problem

Aim: To minimize total assignment cost.

Procedure: Check all permutations of task assignments.

Code:

```
import itertools
```

```
def assignment_problem(cost):
    n = len(cost)
    min_cost = float('inf')
    best = None
    for perm in itertools.permutations(range(n)):
        total = sum(cost[i][perm[i]] for i in range(n))
        if total < min_cost:
            min_cost = total
            best = perm
    return best, min_cost

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

Output: Minimum cost printed.

Result: Optimal assignment obtained.

Q14. 0–1 Knapsack Problem

Aim: To solve 0–1 knapsack using brute force.

Procedure: Generate all possible subsets of items.

Code:

```
import itertools
def knapsack(weights, values, capacity):
    n = len(weights)
    max_value = 0
    best = None
    for r in range(1, n+1):
        for comb in itertools.combinations(range(n), r):
            w = sum(weights[i] for i in comb)
            v = sum(values[i] for i in comb)
            if w <= capacity and v > max_value:
                max_value = v
                best = comb
    return best, max_value

print(knapsack([2,3,1],[4,5,3],4))
```

Output: Maximum value printed.

Result: Optimal subset selected.