Problem 1

```
python
Copy code
def process_list(lst):
    return sorted(lst)

# Test Cases
print(process_list([]))            # Output: []
print(process_list([1]))           # Output: [1]
print(process_list([7, 7, 7, 7]))  # Output: [7, 7, 7, 7]
print(process_list([-5, -1, -3, -2, -4]))  # Output: [-5, -4, -3, -2, -1]
```

Problem 2: Selection Sort Explanation

Selection Sort Algorithm:

1. Divide the array into sorted and unsorted regions.
2. Repeatedly select the smallest element from the unsorted region.
3. Swap it with the leftmost unsorted element.
4. Move the boundary of the sorted region one element to the right.

Example:

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

Step-by-Step:

1. Find the smallest element in the unsorted region [5, 2, 9, 1, 5, 6] -> 1.
2. Swap it with the first element: [1, 2, 9, 5, 5, 6].
3. Repeat for the remaining unsorted region: [2, 9, 5, 5, 6], [9, 5, 5, 6], [9, 5, 6], [9, 6].

Selection Sort Code:

```
python
Copy 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

# Test Cases
print(selection_sort([5, 2, 9, 1, 5, 6]))  # Output: [1, 2, 5, 5, 6, 9]
print(selection_sort([10, 8, 6, 4, 2]))    # Output: [2, 4, 6, 8, 10]
print(selection_sort([1, 2, 3, 4, 5]))     # Output: [1, 2, 3, 4, 5]
```

Why Selection Sort is Simple but Inefficient:

- Simple to understand and implement.
- Inefficient for large datasets (O(n^2) time complexity).

Problem 3: Optimized Bubble Sort

```
python
Copy 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

# Test Cases
print(bubble_sort([64, 25, 12, 22, 11]))  # Output: [11, 12, 22, 25, 64]
print(bubble_sort([29, 10, 14, 37, 13]))  # Output: [10, 13, 14, 29, 37]
print(bubble_sort([3, 5, 2, 1, 4]))       # Output: [1, 2, 3, 4, 5]
print(bubble_sort([1, 2, 3, 4, 5]))       # Output: [1, 2, 3, 4, 5]
print(bubble_sort([5, 4, 3, 2, 1]))       # Output: [1, 2, 3, 4, 5]
```

Problem 4: Insertion Sort with Duplicates

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

# Test Cases
print(insertion_sort([3, 1, 4, 1, 5, 9, 2, 6, 5, 3]))  # Output: [1, 1, 2,
3, 3, 4, 5, 5, 6, 9]
print(insertion_sort([5, 5, 5, 5, 5]))                # Output: [5, 5, 5,
5, 5]
print(insertion_sort([2, 3, 1, 3, 2, 1, 1, 3]))        # Output: [1, 1, 1,
2, 2, 3, 3, 3]
```

Problem 5: Find Kth Missing Positive Number

```
python
Copy code
def find_kth_missing(arr, k):
    missing_count = 0
    current = 1
    i = 0
    while missing_count < k:
        if i < len(arr) and arr[i] == current:
            i += 1
        else:
            missing_count += 1
        current += 1
```

```
    return current - 1

# Test Cases
print(find_kth_missing([2, 3, 4, 7, 11], 5))  # Output: 9
print(find_kth_missing([1, 2, 3, 4], 2))      # Output: 6
```

Problem 6: Find Peak Element

```
python
Copy code
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

# Test Cases
print(find_peak_element([1, 2, 3, 1]))          # Output: 2
print(find_peak_element([1, 2, 1, 3, 5, 6, 4])) # Output: 5
```

Problem 7: First Occurrence of Needle in Haystack

```
python
Copy code
def str_str(haystack, needle):
    return haystack.find(needle)

# Test Cases
print(str_str("sadbutsad", "sad"))  # Output: 0
print(str_str("leetcode", "leeto")) # Output: -1
```

Problem 8: Substring in Words

```
python
Copy code
def string_matching(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
    return res

# Test Cases
print(string_matching(["mass","as","hero","superhero"])) # Output:
["as","hero"]
print(string_matching(["leetcode","et","code"]))         # Output:
["et","code"]
print(string_matching(["blue","green","bu"]))            # Output: []
```

Problem 9: Closest Pair of Points (Brute Force)

```
python
```

```
Copy code
import math

def closest_pair(points):
    def euclidean_distance(p1, p2):
        return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

    min_distance = float('inf')
    closest_points = None

    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            dist = euclidean_distance(points[i], points[j])
            if dist < min_distance:
                min_distance = dist
                closest_points = (points[i], points[j])

    return closest_points, min_distance

# Test Case
points = [(1, 2), (4, 5), (7, 8), (3, 1)]
closest_points, min_distance = closest_pair(points)
print(f"Closest pair: {closest_points} Minimum distance: {min_distance}")
# Output: Closest pair: ((1, 2), (3, 1)) Minimum distance:
1.4142135623730951
```

Problem 10: Brute Force Convex Hull

```
python
Copy code
def convex_hull(points):
    def cross_product(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] -
o[0])

    points = sorted(set(points))
    if len(points) < 3:
        return points

    lower, upper = [], []

    for p in points:
        while len(lower) >= 2 and cross_product(lower[-2], lower[-1], p) <=
0:
            lower.pop()
        lower.append(p)

    for p in reversed(points):
        while len(upper) >= 2 and cross_product(upper[-2], upper[-1], p) <=
0:
            upper.pop()
        upper.append(p)

    return lower[:-1] + upper[:-1]

# Test Case
points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
print(convex_hull(points)) # Output: [(0, 0), (1, 1), (8, 1), (4, 6)]
```

Problem 11: TSP using Exhaustive Search

```
python
Copy code
import itertools

def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

def tsp(cities):
    min_distance = float('inf')
    best_path = None

    for perm in itertools.permutations(cities[1:]):
        current_path = [cities[0]] + list(perm) + [cities[0]]
        current_distance = sum(distance(current_path[i], current_path[i+1])
for i in range(len(current_path) - 1))

        if current_distance < min_distance:
            min_distance = current_distance
            best_path = current_path

    return min_distance, best_path

# Test Cases
cities1 = [(1, 2), (4, 5), (7, 1), (3, 6)]
print(tsp(cities1))  # Output: (Shortest Distance and Path)

cities2 = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]
print(tsp(cities2))  # Output: (Shortest Distance and Path)
```

Problem 12: Assignment Problem using Exhaustive Search

```
python
Copy 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_assignment = None

    for perm in itertools.permutations(range(n)):
        current_cost = total_cost(perm, cost_matrix)
        if current_cost < min_cost:
            min_cost = current_cost
            best_assignment = perm

    return best_assignment, min_cost

# Test Cases
cost_matrix1 = [[3, 10, 7], [8, 5, 12], [4, 6, 9]]
print(assignment_problem(cost_matrix1))  # Output: Optimal Assignment and
Total Cost

cost_matrix2 = [[15, 9, 4], [8, 7, 18], [6, 12, 11]]
print(assignment_problem(cost_matrix2))  # Output: Optimal Assignment and
Total Cost
```

Problem 13: 0-1 Knapsack Problem using Exhaustive Search

```
python
Copy code
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)
    max_value = 0
    best_items = None

    for r in range(n + 1):
        for subset in itertools.combinations(range(n), r):
            if is_feasible(subset, weights, capacity):
                current_value = total_value(subset, values)
                if current_value > max_value:
                    max_value = current_value
                    best_items = subset

    return best_items, max_value

# Test Cases
weights1 = [2, 3, 1]
values1 = [4, 5, 3]
capacity1 = 4
print(knapsack(weights1, values1, capacity1))  # Output: Optimal Selection
and Total Value

weights2 = [1, 2, 3, 4]
values2 = [2, 4, 6, 3]
capacity2 = 6
print(knapsack(weights2, values2, capacity2))
```