```
def process_list(lst):
    return lst
print(process_list([]))
print(process_list([1]))
print(process_list([7, 7, 7, 7]))
print(process_list([-5, -1, -3, -2, -4]))

def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
print(selection_sort([5, 2, 9, 1, 5, 6]))
print(selection_sort([10, 8, 6, 4, 2]))
print(selection_sort([1, 2, 3, 4, 5]))

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
print(bubble_sort([64, 25, 12, 22, 11]))
print(bubble_sort([29, 10, 14, 37, 13]))
print(bubble_sort([3, 5, 2, 1, 4]))
print(bubble_sort([1, 2, 3, 4, 5]))
print(bubble_sort([5, 4, 3, 2, 1]))

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
print(insertion_sort([3, 1, 4, 1, 5, 9, 2, 6, 5, 3]))
print(insertion_sort([5, 5, 5, 5, 5]))
print(insertion_sort([2, 3, 1, 3, 2, 1, 1, 3]))

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
        if missing_count < k:
            current += 1
    return current
print(find_kth_missing([2, 3, 4, 7, 11], 5))
print(find_kth_missing([1, 2, 3, 4], 2))

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]))

def str_str(haystack, needle):
    return haystack.find(needle)
print(str_str("sadbutsad", "sad"))
print(str_str("leetcode", "leeto"))

def find_substrings(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(find_substrings(["mass", "as", "hero", "superhero"]))
print(find_substrings(["leetcode", "et", "code"]))
print(find_substrings(["blue", "green", "bu"]))

import math

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

def orientation(p, q, r):
    val = (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] - q[1])
    if val == 0:
        return 0
    elif val > 0:
        return 1
    else:
        return 2
```

```
def convex_hull(points):
    n = len(points)
    if n < 3:
        return []

    hull = []
    leftmost = 0
    for i in range(1, n):
        if points[i][0] < points[leftmost][0]:
            leftmost = i

    p = leftmost
    while True:
        hull.append(points[p])
        q = (p + 1) % n
        for i in range(n):
            if orientation(points[p], points[i], points[q]) == 2:
                q = i
        p = q
        if p == leftmost:
            break
    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))

def convex_hull_brute_force(points):
    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])
    points = sorted(points)
    lower = []
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)
    return lower[:-1] + upper[:-1]
points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
print(convex_hull_brute_force(points))
```

```
from itertools import permutations
import math
def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0]) ** 2 + (city1[1] - city2[1]) ** 2)
def tsp(cities):
    n = len(cities)
    min_path = None
    min_distance = float('inf')
    for perm in 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(n))
        if current_distance < min_distance:
            min_distance = current_distance
            min_path = current_path
    return min_distance, min_path
```

```
cities1 = [(1, 2), (4, 5), (7, 1), (3, 6)]
cities2 = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]
print(tsp(cities1))
print(tsp(cities2))
```

```
from itertools import permutations
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 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
cost_matrix1 = [
    [3, 10, 7],
    [8, 5, 12],
    [4, 6, 9]
]
cost_matrix2 = [
    [15, 9, 4],
    [8, 7, 18],
    [6, 12, 11]
]
print(assignment_problem(cost_matrix1))
print(assignment_problem(cost_matrix2))
```

```
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_01(weights, values, capacity):
    n = len(weights)
    max_value = 0
    best_combination = []
    for r in range(n + 1):
        for comb in combinations(range(n), r):
            if is_feasible(comb, weights, capacity):
                current_value = total_value(comb, values)
                if current_value > max_value:
                    max_value = current_value
                    best_combination = comb
    return list(best_combination), max_value
weights1 = [2, 3, 1]
values1 = [4, 5, 3]
capacity1 = 4
weights2 = [1, 2, 3, 4]
values2 = [2, 4, 6, 3]
capacity2 = 6
print(knapsack_01(weights1, values1, capacity1))
print(knapsack_01(weights2, values2, capacity2
```