1. import sympy as sp

```
def find_extrema(function, variable, interval):
    # Define the variable and function
    x = sp.Symbol(variable)
    f = sp.sympify(function)

    # Compute the derivative of the function
    f_prime = sp.diff(f, x)

    # Solve for critical points
    critical_points = sp.solveset(f_prime, x, domain=sp.S.Reals)

    # Evaluate the function at critical points and at the interval endpoints
    endpoints = [interval[0], interval[1]]
    critical_values = [f.subs(x, point) for point in critical_points if point.is_real and interval[0] <= point <= interval[1]]
    endpoint_values = [f.subs(x, point) for point in endpoints]

    # Combine and evaluate all points
    all_values = critical_values + endpoint_values
    max_value = max(all_values)
    min_value = min(all_values)

    return max_value, min_value

# Example usage
function = "x**3 - 3*x**2 + 4"
variable = "x"
interval = (0, 3)

max_val, min_val = find_extrema(function, variable, interval)
print(f"Maximum value: {max_val}")
print(f"Minimum value: {min_val}")\
```

2. def merge_sort(arr):
    if len(arr) > 1:

```
    mid = len(arr) // 2  # Finding the mid of the array
    left_half = arr[:mid]  # Dividing the elements into 2 halves
    right_half = arr[mid:]

    merge_sort(left_half)  # Sorting the first half
    merge_sort(right_half)  # Sorting the second half

    i = j = k = 0

    # Copy data to temp arrays L[] and R[]
    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            arr[k] = left_half[i]
            i += 1
        else:
            arr[k] = right_half[j]
            j += 1
        k += 1

    # Checking if any element was left
    while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1

    while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1

def print_list(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()

# Example usage
```

```
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is")
    print_list(arr)
    merge_sort(arr)
    print("Sorted array is")
    print_list(arr)
```

3.
```
def quick_sort(arr, low, high):
    if low < high:
        # pi is partitioning index, arr[p] is now at the right place
        pi = partition(arr, low, high)

        # Separately sort elements before partition and after partition
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

def partition(arr, low, high):
    pivot = arr[high]  # pivot
    i = low - 1  # Index of smaller element

    for j in range(low, high):
        # If current element is smaller than or equal to pivot
        if arr[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]  # Swap

    arr[i + 1], arr[high] = arr[high], arr[i + 1]  # Swap pivot element with the element at i + 1
    return i + 1

def print_list(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()

# Example usage
```

```
if __name__ == '__main__':
    arr = [10, 7, 8, 9, 1, 5]
    n = len(arr)
    print("Given array is")
    print_list(arr)
    quick_sort(arr, 0, n - 1)
    print("Sorted array is")
    print_list(arr)
```

4.
```
def binary_search(arr, low, high, x):
    # Check base case
    if high >= low:
        mid = (high + low) // 2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it can only be present in left subarray
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)

        # Else the element can only be present in right subarray
        else:
            return binary_search(arr, mid + 1, high, x)

    else:
        # Element is not present in array
        return -1

def print_result(index):
    if index != -1:
        print(f"Element is present at index {index}")
    else:
        print("Element is not present in array")

# Example usage
```

```
if __name__ == "__main__":
    arr = [2, 3, 4, 10, 40]
    x = 10

    # Function call
    result = binary_search(arr, 0, len(arr)-1, x)
    print_result(result)
```

5.
```
import numpy as np

def add_matrices(A, B):
    return [[A[i][j] + B[i][j] for j in range(len(A[0]))] for i in range(len(A))]

def subtract_matrices(A, B):
    return [[A[i][j] - B[i][j] for j in range(len(A[0]))] for i in range(len(A))]

def split_matrix(matrix):
    row, col = len(matrix), len(matrix[0])
    row2, col2 = row // 2, col // 2
    return matrix[:row2][:col2], matrix[:row2][col2:], matrix[row2:][:col2], matrix[row2:][col2:]

def strassen_multiply(A, B):
    if len(A) == 1:
        return [[A[0][0] * B[0][0]]]

    A11, A12, A21, A22 = split_matrix(A)
    B11, B12, B21, B22 = split_matrix(B)

    M1 = strassen_multiply(add_matrices(A11, A22), add_matrices(B11, B22))
    M2 = strassen_multiply(add_matrices(A21, A22), B11)
    M3 = strassen_multiply(A11, subtract_matrices(B12, B22))
    M4 = strassen_multiply(A22, subtract_matrices(B21, B11))
    M5 = strassen_multiply(add_matrices(A11, A12), B22)
    M6 = strassen_multiply(subtract_matrices(A21, A11), add_matrices(B11, B12))
    M7 = strassen_multiply(subtract_matrices(A12, A22), add_matrices(B21, B22))

    C11 = add_matrices(subtract_matrices(add_matrices(M1, M4), M5), M7)
```

```
    C12 = add_matrices(M3, M5)
    C21 = add_matrices(M2, M4)
    C22 = add_matrices(subtract_matrices(add_matrices(M1, M3), M2), M6)

    new_matrix = [[0 for _ in range(len(A))] for _ in range(len(A))]

    for i in range(len(C11)):
        for j in range(len(C11)):
            new_matrix[i][j] = C11[i][j]
            new_matrix[i][j + len(C11)] = C12[i][j]
            new_matrix[i + len(C11)][j] = C21[i][j]
            new_matrix[i + len(C11)][j + len(C11)] = C22[i][j]

    return new_matrix

# Example usage
A = [[1, 2, 3, 4],
     [5, 6, 7, 8],
     [9, 10, 11, 12],
     [13, 14, 15, 16]]

B = [[16, 15, 14, 13],
     [12, 11, 10, 9],
     [8, 7, 6, 5],
     [4, 3, 2, 1]]

print("Matrix A:")
print(np.matrix(A))

print("Matrix B:")
print(np.matrix(B))

C = strassen_multiply(A, B)
print("Product Matrix
6. def karatsuba(x, y):
    # Base case for recursion
```

```
    if x < 10 or y < 10:
        return x * y

    # Calculate the size of the numbers
    n = max(len(str(x)), len(str(y)))
    m = n // 2

    # Split x and y
    x1, x0 = divmod(x, 10**m)
    y1, y0 = divmod(y, 10**m)

    # 3 recursive calls to Karatsuba
    z0 = karatsuba(x0, y0)
    z2 = karatsuba(x1, y1)
    z1 = karatsuba(x0 + x1, y0 + y1) - z0 - z2

    # Combine the results
    return (z2 * 10**(2*m)) + (z1 * 10**m) + z0

# Example usage
x = 1234
y = 5678

print(f"Multiplication of {x} and {y} using Karatsuba algorithm is {karatsuba(x, y)}")
```

7.
```
import math

# Helper function to calculate the Euclidean distance between two points
def dist(p1, p2):
    return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

# Brute force method to find the smallest distance between points in a subset
def brute_force(points):
    min_dist = float('inf')
    n = len(points)
    for i in range(n):
        for j in range(i + 1, n):
```

```
            if dist(points[i], points[j]) < min_dist:
                min_dist = dist(points[i], points[j])
    return min_dist

# Function to find the smallest distance in a strip of given size
def strip_closest(strip, d):
    min_dist = d
    strip.sort(key=lambda point: point[1])  # Sort strip according to y coordinate

    for i in range(len(strip)):
        for j in range(i + 1, len(strip)):
            if (strip[j][1] - strip[i][1]) < min_dist:
                min_dist = dist(strip[i], strip[j])
    return min_dist

# Recursive function to find the closest pair of points
def closest_pair_rec(points_sorted_x):
    n = len(points_sorted_x)

    # Use brute force if there are 3 or fewer points
    if n <= 3:
        return brute_force(points_sorted_x)

    # Find the midpoint
    mid = n // 2
    mid_point = points_sorted_x[mid]

    # Divide points in left and right halves
    left_half = points_sorted_x[:mid]
    right_half = points_sorted_x[mid:]

    # Recursively find the smallest distances in both subarrays
    dl = closest_pair_rec(left_half)
    dr = closest_pair_rec(right_half)

    # Find the smaller of the two distances
```

```
    d = min(dl, dr)

    # Build a strip of points close to the dividing line
    strip = [point for point in points_sorted_x if abs(point[0] - mid_point[0]) < d]

    # Find the closest points in the strip
    return min(d, strip_closest(strip, d))

# Main function to find the closest pair of points
def closest_pair(points):
    points_sorted_x = sorted(points, key=lambda point: point[0])
    return closest_pair_rec(points_sorted_x)

# Example usage
points = [(2, 3), (12, 30), (40,
8. def partition(arr, low, high, pivot):
    i = low
    j = high
    while True:
        while arr[i] < pivot:
            i += 1
        while arr[j] > pivot:
            j -= 1
        if i >= j:
            return j
        arr[i], arr[j] = arr[j], arr[i]
        i += 1
        j -= 1

def median_of_medians(arr, k):
    n = len(arr)
    if n <= 5:
        return sorted(arr)[k]

    # Step 1: Divide the array into groups of 5
    sublists = [arr[j:j+5] for j in range(0, n, 5)]
```

```
    # Step 2: Sort each group and find the median
    medians = [sorted(sublist)[len(sublist)//2] for sublist in sublists]

    # Step 3: Find the median of these medians
    median_of_medians_value = median_of_medians(medians, len(medians)//2)

    # Step 4: Partition the array around the median of medians
    pivot_index = partition(arr, 0, n - 1, median_of_medians_value)

    # Step 5: Recursively apply to find the kth smallest element
    if k == pivot_index:
        return arr[k]
    elif k < pivot_index:
        return median_of_medians(arr[:pivot_index], k)
    else:
        return median_of_medians(arr[pivot_index+1:], k - pivot_index - 1)

# Example usage
arr = [12, 3, 5, 7, 4, 19, 26]
k = 3  # Looking for the 4th smallest element (0-based index)
print(f"The {k+1}th smallest element is {median_of_medians(arr, k)}
```

9.
```
from itertools import chain, combinations

def all_subsets(nums):
    """Generate all subsets of a list of numbers."""
    return chain(*map(lambda x: combinations(nums, x), range(0, len(nums)+1)))

def meet_in_the_middle(nums, target):
    """Determine if there's a subset whose sum is equal to the target sum using meet-in-the-middle."""
    n = len(nums)
    if n == 0:
        return target == 0

    # Split the list into two halves
```

```
    left_half = nums[:n//2]
    right_half = nums[n//2:]

    # Generate all subset sums for the left and right halves
    left_sums = {sum(subset) for subset in all_subsets(left_half)}
    right_sums = {sum(subset) for subset in all_subsets(right_half)}

    # Check if there's a subset in left_sums or right_sums that equals the target
    if target in left_sums or target in right_sums:
        return True

    # Check if any pair of sums from left_sums and right_sums adds up to the target
    for left_sum in left_sums:
        if (target - left_sum) in right_sums:
            return True

    return False

# Example usage
nums = [3, 34, 4, 12, 5, 2]
target = 9
if meet_in_the_middle(nums, target):
    print(f"There is a subset with sum {target}")
else:
    print(f"There is no subset with sum {target}")
```

10.