10.
```
def closest_pair_of_points(points):
    def distance(p1, p2):
        return ((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2) ** 0.5

    def brute_force(points):
        min_dist = float('inf')
        for i in range(len(points)):
            for j in range(i + 1, len(points)):
                if distance(points[i], points[j]) < min_dist:
                    min_dist = distance(points[i], points[j])
        return min_dist

    def closest_pair(points):
        if len(points) <= 3:
            return brute_force(points)

        mid = len(points) // 2
        mid_point = points[mid]

        left_points = points[:mid]
        right_points = points[mid:]

        left_min = closest_pair(left_points)
        right_min = closest_pair(right_points)

        min_dist = min(left_min, right_min)

        strip = [point for point in points if abs(point[0] - mid_point[0]) < min_dist]
        strip.sort(key=lambda x: x[1])

        min_strip = float('inf')
        for i in range(len(strip)):
            j = i + 1
            while j < len(strip) and (strip[j][1] - strip[i][1]) < min_strip:
                min_strip = min(min_strip, distance(strip[i], strip[j]))
                j += 1
```

```
    return min(min_dist, min_strip)

  points.sort()
  return closest_pair(points)
```

9.
```
def merge_sort(arr):
  if len(arr) > 1:
    mid = len(arr) // 2
    L = arr[:mid]
    R = arr[mid:]

    merge_sort(L)
    merge_sort(R)

    i = j = k = 0

    while i < len(L) and j < len(R):
      if L[i] < R[j]:
        arr[k] = L[i]
        i += 1
      else:
        arr[k] = R[j]
        j += 1
      k += 1

    while i < len(L):
      arr[k] = L[i]
      i += 1
      k += 1

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

```
    return arr

arr = [12, 11, 13, 5, 6, 7]
print("Given array is", arr)

sorted_arr = merge_sort(arr)
print("Sorted array is", sorted_arr)

8.def combinationSum(candidates, target):
    def backtrack(start, path, target):
        if target == 0:
            res.append(path[:])
            return
        for i in range(start, len(candidates)):
            if candidates[i] > target:
                continue
            path.append(candidates[i])
            backtrack(i, path, target - candidates[i])
            path.pop()

    res = []
    candidates.sort()
    backtrack(0, [], target)
    return res

# Example
candidates = [2, 3, 6, 7]
target = 7
print(combinationSum(candidates, target))

7.def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] < x:
```

```
            low = mid + 1
        elif arr[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1

# Example Usage
arr = [2, 4, 6, 8, 10, 12, 14, 16]
x = 10
result = binary_search(arr, x)
if result != -1:
    print(f"Element found at index {result}")
else:
    print("Element not found")
6.def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

# Example Usage
arr = [64, 25, 12, 22, 11]
sorted_arr = selection_sort(arr)
print("Sorted array:", sorted_arr)

5.import heapq
import sys

def dijkstra(graph, source):
    num_vertices = len(graph)
    distances = [float('inf')] * num_vertices
```

```
    distances[source] = 0
    visited = [False] * num_vertices
    priority_queue = [(0, source)]  # (distance, vertex)

    while priority_queue:
        dist_u, u = heapq.heappop(priority_queue)

        if visited[u]:
            continue

        visited[u] = True

        for v in range(num_vertices):
            if not visited[v] and graph[u][v] != float('inf'):
                new_dist = dist_u + graph[u][v]
                if new_dist < distances[v]:
                    distances[v] = new_dist
                    heapq.heappush(priority_queue, (new_dist, v))

    return distances

# Example usage:
graph = [
    [0, 7, 9, float('inf'), float('inf'), 14],
    [7, 0, 10, 15, float('inf'), float('inf')],
    [9, 10, 0, 11, float('inf'), 2],
    [float('inf'), 15, 11, 0, 6, float('inf')],
    [float('inf'), float('inf'), float('inf'), 6, 0, 9],
    [14, float('inf'), 2, float('inf'), 9, 0]
]

source_vertex = 0
shortest_distances = dijkstra(graph, source_vertex)

# Print the shortest distances from the source vertex
for i, dist in enumerate(shortest_distances):
```

```
    print(f"Shortest distance from vertex {source_vertex} to vertex {i} is {dist}")
```

4.

```
def rob(nums):
    n = len(nums)

    if n == 0:
        return 0
    elif n == 1:
        return nums[0]

    # Helper function for regular House Robber problem (no circular)
    def house_robber(nums):
        prev1 = 0
        prev2 = 0

        for num in nums:
            temp = prev1
            prev1 = max(prev2 + num, prev1)
            prev2 = temp

        return prev1

    # Rob houses from 0 to n-2 and from 1 to n-1, take the maximum of both
    return max(house_robber(nums[:-1]), house_robber(nums[1:]))

# Example usage:
nums1 = [2, 3, 2]  # Output: 3 (Rob house 1 and 3)
nums2 = [1, 2, 3, 1]  # Output: 4 (Rob house 1 and 3)
nums3 = [0]  # Output: 0 (No houses to rob)

print("Maximum amount of money that can be robbed:", rob(nums1))
print("Maximum amount of money that can be robbed:", rob(nums2))
print("Maximum amount of money that can be robbed:", rob(nums3))
def rob(nums):
    n = len(nums)
```

```
    if n == 0:
        return 0
    elif n == 1:
        return nums[0]

    # Helper function for regular House Robber problem (no circular)
    def house_robber(nums):
        prev1 = 0
        prev2 = 0

        for num in nums:
            temp = prev1
            prev1 = max(prev2 + num, prev1)
            prev2 = temp

        return prev1

    # Rob houses from 0 to n-2 and from 1 to n-1, take the maximum of both
    return max(house_robber(nums[:-1]), house_robber(nums[1:]))

# Example usage:
nums1 = [2, 3, 2]  # Output: 3 (Rob house 1 and 3)
nums2 = [1, 2, 3, 1]  # Output: 4 (Rob house 1 and 3)
nums3 = [0]  # Output: 0 (No houses to rob)

print("Maximum amount of money that can be robbed:", rob(nums1))
print("Maximum amount of money that can be robbed:", rob(nums2))
print("Maximum amount of money that can be robbed:", rob(nums3))

3.def rob(nums):
    n = len(nums)

    if n == 0:
        return 0
    elif n == 1:
        return nums[0]
```

```
    # Helper function for regular House Robber problem (no circular)
    def house_robber(nums):
        prev1 = 0
        prev2 = 0

        for num in nums:
            temp = prev1
            prev1 = max(prev2 + num, prev1)
            prev2 = temp

        return prev1

    # Rob houses from 0 to n-2 and from 1 to n-1, take the maximum of both
    return max(house_robber(nums[:-1]), house_robber(nums[1:]))

# Example usage:
nums1 = [2, 3, 2]  # Output: 3 (Rob house 1 and 3)
nums2 = [1, 2, 3, 1]  # Output: 4 (Rob house 1 and 3)
nums3 = [0]  # Output: 0 (No houses to rob)

print("Maximum amount of money that can be robbed:", rob(nums1))
print("Maximum amount of money that can be robbed:", rob(nums2))
print("Maximum amount of money that can be robbed:", rob(nums3))
def rob(nums):
    n = len(nums)

    if n == 0:
        return 0
    elif n == 1:
        return nums[0]

    # Helper function for regular House Robber problem (no circular)
    def house_robber(nums):
        prev1 = 0
        prev2 = 0
```

```
        for num in nums:
            temp = prev1
            prev1 = max(prev2 + num, prev1)
            prev2 = temp

        return prev1

    # Rob houses from 0 to n-2 and from 1 to n-1, take the maximum of both
    return max(house_robber(nums[:-1]), house_robber(nums[1:]))

# Example usage:
nums1 = [2, 3, 2]  # Output: 3 (Rob house 1 and 3)
nums2 = [1, 2, 3, 1]  # Output: 4 (Rob house 1 and 3)
nums3 = [0]  # Output: 0 (No houses to rob)

print("Maximum amount of money that can be robbed:", rob(nums1))
print("Maximum amount of money that can be robbed:", rob(nums2))
print("Maximum amount of money that can be robbed:", rob(nums3))

2.def find_min_max(arr):
    if not arr:
        return None, None

    min_val = arr[0]
    max_val = arr[-1]

    return min_val, max_val

# Example usage:
arr1 = [2, 4, 6, 8, 10, 12, 14, 18]
arr2 = [11, 13, 15, 17, 19, 21, 23, 35, 37]
arr3 = [22, 34, 35, 36, 43, 67]

min1, max1 = find_min_max(arr1)
min2, max2 = find_min_max(arr2)
```

```
min3, max3 = find_min_max(arr3)

print(f"Input: {arr1}")
print(f"Output: Min = {min1}, Max = {max1}\n")

print(f"Input: {arr2}")
print(f"Output: Min = {min2}, Max = {max2}\n")

print(f"Input: {arr3}")
print(f"Output: Min = {min3}, Max = {max3}\n")

1.def max_regions_colored(adj_list):
    num_regions = len(adj_list)
    coloring = [-1] * num_regions  # -1 means uncolored
    available_colors = set(range(num_regions))  # Initial available colors

    regions_colored_by_you = 0

    for region in range(num_regions):
        if coloring[region] == -1:  # If region is uncolored
            neighbors = adj_list[region]
            used_colors = set(coloring[n] for n in neighbors if coloring[n] != -1)

            # Find the first available color
            for color in available_colors:
                if color not in used_colors:
                    coloring[region] = color
                    break

            # If we successfully colored this region
            if coloring[region] != -1:
                regions_colored_by_you += 1
                available_colors.remove(coloring[region])  # Remove used color

    return regions_colored_by_you
```

```
# Example usage:
adj_list = [
    [1, 2],     # Region 0 is adjacent to 1 and 2
    [0, 2, 3],  # Region 1 is adjacent to 0, 2, and 3
    [0, 1, 3],  # Region 2 is adjacent to 0, 1, and 3
    [1, 2],     # Region 3 is adjacent to 1 and 2
]

max_colored = max_regions_colored(adj_list)
print("Maximum regions colored by you:", max_colored)
```