```python
def find_min_max(arr, low, high):
    if low == high:
        return (arr[low], arr[low])

    if high == low + 1:
        if arr[low] < arr[high]:
            return (arr[low], arr[high])
        else:
            return (arr[high], arr[low])

    mid = (low + high) // 2
    min1, max1 = find_min_max(arr, low, mid)
    min2, max2 = find_min_max(arr, mid + 1, high)

    return (min(min1, min2), max(max1, max2))

def find_min_max_values(arr):
    return find_min_max(arr, 0, len(arr) - 1)

# Test Cases
print(find_min_max_values([5, 7, 3, 4, 9, 12, 6, 2]))       # Output: Min = 2, Max = 12
print(find_min_max_values([1, 3, 5, 7, 9, 11, 13, 15, 17])) # Output: Min = 1, Max = 17
print(find_min_max_values([22, 34, 35, 36, 43, 67, 12, 13, 15, 17])) # Output: Min = 12, Max = 67
```

main.py

```python
def largeGroupPositions(s):
    result = []
    n = len(s)
    i = 0

    while i < n:
        j = i
        while j < n and s[j] == s[i]:
            j += 1
        if j - i >= 3:
            result.append([i, j - 1])
        i = j

    return result


print(largeGroupPositions("abbxxxxzzy"))
print(largeGroupPositions("abc"))
```

```
# Test cases
arr1 = [31, 23, 35, 27, 11, 21, 15, 28]
merge_sort(arr1)
print("Sorted array 1:", arr1)

arr2 = [22, 34, 25, 36, 43, 67, 52, 13, 65, 17]
merge_sort(arr2)
print("Sorted array 2:", arr2)
```

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2  # Find the middle point
        L = arr[:mid]  # Split the array into two halves
        R = arr[mid:]

        merge_sort(L)  # Recursively sort the first half
        merge_sort(R)  # Recursively sort the second half

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        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

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

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

main.py

```python
def findSubstrings(words):
    result = set()
    words_set = set(words)  # Use a set for faster look-up

    for word in words:
        for other in words_set:
            if word != other and word in other:
                result.add(word)
                break

    return list(result)

# Example usage
print(findSubstrings(["mass", "as", "hero", "superhero"]))
print(findSubstrings(["leetcode", "et", "code"]))
print(findSubstrings(["blue", "green", "bu"]))
```

onlinegdb.com/online_python_compiler

main.py

```python
def gameOfLife(board):
    m, n = len(board), len(board[0])

    def count_live_neighbors(x, y):
        directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]
        count = 0
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < m and 0 <= ny < n and (board[nx][ny] == 1 or board[nx][ny] == 2):
                count += 1
        return count
    for i in range(m):
        for j in range(n):
            live_neighbors = count_live_neighbors(i, j)
            if board[i][j] == 1 and (live_neighbors < 2 or live_neighbors > 3):
                board[i][j] = 2
            elif board[i][j] == 0 and live_neighbors == 3:
                board[i][j] = 3

    for i in range(m):
        for j in range(n):
            if board[i][j] == 2:
                board[i][j] = 0
            elif board[i][j] == 3:
                board[i][j] = 1

board1 = [[0, 1, 0], [0, 0, 1], [1, 1, 1], [0, 0, 0]]
gameOfLife(board1)
print(board1)

board2 = [[1, 1], [1, 0]]
gameOfLife(board2)
print(board2)
```

main.py

```python
import itertools
import math

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

def shortest_path(cities):
    min_path = None
    min_distance = float('inf')

    # Generate all permutations of the cities
    for perm in itertools.permutations(cities):
        # Calculate the distance of this permutation
        current_distance = sum(distance(perm[i], perm[i + 1]) for i in range(len(perm) - 1))
        current_distance += distance(perm[-1], perm[0])

        if current_distance < min_distance:
            min_distance = current_distance
            min_path = perm

    return min_distance, min_path

# Test Case 1
cities1 = [(1, 2), (4, 5), (7, 1), (3, 6)]
distance1, path1 = shortest_path(cities1)
print(f"Shortest Distance: {distance1}")
print(f"Shortest Path: {path1}")

# Test Case 2
cities2 = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]
distance2, path2 = shortest_path(cities2)
print(f"Shortest Distance: {distance2}")
print(f"Shortest Path: {path2}")
import itertools
import math

def distance(p1, p2):
```

main.py

```python
def strStr(haystack, needle):
    # Edge cases
    if needle == "":
        return 0
    if haystack == "":
        return -1

    haystack_length = len(haystack)
    needle_length = len(needle)

    # Iterate through haystack to find the needle
    for i in range(haystack_length - needle_length + 1):
        if haystack[i:i + needle_length] == needle:
            return i

    return -1
haystack = "sadbutsad"
needle = "sad"
print(strStr(haystack, needle))  # Output: 0
```

```python
def distance(p1, p2):
    return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

def shortest_path(cities):
    min_path = None
    min_distance = float('inf')

    for perm in itertools.permutations(cities):

        current_distance = sum(distance(perm[i], perm[i + 1]) for i in range(len(perm) - 1))
        current_distance += distance(perm[-1], perm[0])

        if current_distance < min_distance:
            min_distance = current_distance
            min_path = perm

    return min_distance, min_path

cities1 = [(1, 2), (4, 5), (7, 1), (3, 6)]
distance1, path1 = shortest_path(cities1)
print(f"Shortest Distance: {distance1}")
print(f"Shortest Path: {path1}")


cities2 = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]
distance2, path2 = shortest_path(cities2)
print(f"Shortest Distance: {distance2}")
print(f"Shortest Path: {path2}")
```

```
def brute_force_search(text, pattern):
    n = len(text)
    m = len(pattern)
    comparisons = 0

    for i in range(n - m + 1):
        j = 0
        while j < m:
            comparisons += 1
            if text[i + j] != pattern[j]:
                break
            j += 1
        if j == m:
            print(f"Pattern found at index {i}")

    return comparisons

# Test case
text = "ACGTACGTACGT"
pattern = "ACG"
comparisons = brute_force_search(text, pattern)
print(f"Total comparisons: {comparisons}")
```

```python

    return distances


adjacency_list = {
    0: [(1, 4), (2, 1)],
    1: [(3, 1)],
    2: [(1, 2), (3, 5)],
    3: [(4, 3)],
    4: []
}


start_vertex = 0
shortest_paths = dijkstra(adjacency_list, start_vertex)

# Print the shortest path from the start vertex to all other vertices
print(f"Shortest paths from vertex {start_vertex}:")
for vertex, distance in shortest_paths.items():
    print(f"Vertex {vertex}: {distance}")
```

main.py

```python
def find_min_max(arr):
    def divide_and_conquer(low, high):
        if low == high:
            return (arr[low], arr[low])

        if high == low + 1:
            if arr[low] < arr[high]:
                return (arr[low], arr[high])
            else:
                return (arr[high], arr[low])

        mid = (low + high) // 2
        min1, max1 = divide_and_conquer(low, mid)
        min2, max2 = divide_and_conquer(mid + 1, high)

        return (min(min1, min2), max(max1, max2))

    return divide_and_conquer(0, len(arr) - 1)

# Test case
arr4 = [3, 5, 1, 9, 2, 8, 4, 7]
min_value, max_value = find_min_max(arr4)
print(f"Min = {min_value}, Max = {max_value}")
```

main.py

```python
def uniquePaths(m, n):
    dp = [[1] * n for _ in range(m)]


    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]

    return dp[m-1][n-1]


print(uniquePaths(7, 3))
print(uniquePaths(3, 2))
```

main.py

```python
def min_coins_greedy(denominations, target):
    denominations.sort(reverse=True)
    count = 0
    for coin in denominations:
        while target >= coin:
            target -= coin
            count += 1
    return count

denominations = [1, 2, 5, 10]
target = 27
print("Minimum number of coins needed:", min_coins_greedy(denominations, target))
```

n.py

```python
def find_min_max(arr):
    def divide_and_conquer(low, high):
        if low == high:
            return (arr[low], arr[low])

        if high == low + 1:
            if arr[low] < arr[high]:
                return (arr[low], arr[high])
            else:
                return (arr[high], arr[low])

        mid = (low + high) // 2
        min1, max1 = divide_and_conquer(low, mid)
        min2, max2 = divide_and_conquer(mid + 1, high)

        return (min(min1, min2), max(max1, max2))

    return divide_and_conquer(0, len(arr) - 1)

# Test case
arr4 = [3, 5, 1, 9, 2, 8, 4, 7]
min_value, max_value = find_min_max(arr4)
print(f"Min = {min_value}, Max = {max_value}")
```

ain.py

```python
def greedy_set_cover(universe, subsets):
    cover = set()
    covered = set()

    while covered != universe:
        # Choose the subset that covers the most uncovered elements
        best_subset = max(subsets, key=lambda s: len(s - covered))
        cover.add(best_subset)
        covered.update(best_subset)
        subsets.remove(best_subset)

    return cover

# Define the universe and subsets
universe = {1, 2, 3, 4, 5}
subsets = [{1, 2}, {2, 3, 4}, {4, 5}]

cover = greedy_set_cover(universe, subsets)

print("Greedy set cover:")
for subset in cover:
    print(subset)
```

ain.py

```python
import heapq

def dijkstra(adjacency_list, start_vertex):

    priority_queue = []
    heapq.heappush(priority_queue, (0, start_vertex))  # (distance, vertex)

    # Initialize distances with infinity
    distances = {vertex: float('infinity') for vertex in range(len(adjacency_list))}
    distances[start_vertex] = 0


    visited = set()

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)
        if current_vertex in visited:
            continue

        visited.add(current_vertex)

        for neighbor, weight in adjacency_list[current_vertex]:
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances
```

nain.py

```python
def min_coins_greedy(denominations, target):
    denominations.sort(reverse=True)
    count = 0
    for coin in denominations:
        while target >= coin:
            target -= coin
            count += 1
    return count

denominations = [1, 2, 5, 10]
target = 27
print("Minimum number of coins needed:", min_coins_greedy(denominations, target))
```

n.py

```python
def binomial_coefficient(n, k):
    C = [[0] * (k+1) for _ in range(n+1)]

    for i in range(n+1):
        for j in range(min(i, k)+1):
            if j == 0 or j == i:
                C[i][j] = 1
            else:
                C[i][j] = C[i-1][j-1] + C[i-1][j]

    return C[n][k]

print("C(5,2) =", binomial_coefficient(5, 2))
```