```python
import itertools

def count_combinations(target_sum, num_dice):
  if target_sum < num_dice or target_sum > 6 * num_dice:
    return 0

  combinations = 0
  for dice in itertools.product(range(1, 7), repeat=num_dice):
    if sum(dice) == target_sum:
      combinations += 1

  return combinations

total_combinations = 6**5
target_sum = 20
favorable_combinations = count_combinations(target_sum, 5)
print(favorable_combinations)
```

```
651

=== Code Execution Successful ===
```

```python
def knapsack_0_1(weights, values, capacity):
  n = len(weights)
  dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

  for i in range(1, n + 1):
    for w in range(1, capacity + 1):
      if weights[i - 1] > w:
        dp[i][w] = dp[i - 1][w]
      else:
        dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])

  return dp[n][capacity]

weights = [10, 20, 30, 40]
values = [60, 100, 120, 200]
capacity = 50

max_value = knapsack_0_1(weights, values, capacity)
print("Maximum value:", max_value)
```

```
Maximum value: 260

=== Code Execution Successful ===
```

```python
def bellman_ford(graph, source):
  distances = {vertex: float('inf') for vertex in graph}
  distances[source] = 0

  for _ in range(len(graph) - 1):
    for u in graph:
      for v, weight in graph[u]:
        if distances[u] != float('inf') and distances[u] + weight < distances[v]:
          distances[v] = distances[u] + weight

  for u in graph:
    for v, weight in graph[u]:
      if distances[u] != float('inf') and distances[u] + weight < distances[v]:
        print("Negative cycle detected!")
        return

  return distances


graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('C', 2), ('D', 6)],
    'C': [('D', 3)],
    'D': []
}

source = 'A'
shortest_distances = bellman_ford(graph, source)
print(shortest_distances)
```

```
{'A': 0, 'B': 1, 'C': 3, 'D': 6}

=== Code Execution Successful ==
```

```python
def is_valid_coloring(graph, color_map, vertex, color):


  for neighbor in graph[vertex]:
    if neighbor in color_map and color_map[neighbor] == color:
      return False
  return True
```

```
=== Code Execution Successful ===
```

```python
import heapq

def dijkstra(graph, source):
  pq = [(0, source)]
  distances = {source: 0}

  while pq:
    dist, node = heapq.heappop(pq)

    if dist > distances[node]:
      continue

    for neighbor, weight in graph[node]:
      new_dist = distances[node] + weight
      if new_dist < distances.get(neighbor, float('inf')):
        distances[neighbor] = new_dist
        heapq.heappush(pq, (new_dist, neighbor))

  return distances

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

source = 0
shortest_distances = dijkstra(graph, source)
print(shortest_distances)
```

```
{0: 0, 1: 3, 2: 1, 3: 4, 4: 7}

=== Code Execution Successful ===
```

```python
def greedy_set_cover(universe, sets):
    covered = set()
    selected_sets = []
    while covered != universe:
        best_set = max(sets, key=lambda s: len(s - covered))
        covered |= best_set
        selected_sets.append(best_set)
    return selected_sets
universe = {1, 2, 3, 4, 5}
sets = [{1, 2}, {2, 3,4}, {4,5}]
cover = greedy_set_cover(universe, sets)

print("Selected sets for cover:")
for s in cover:
    print(s)
```

```
Selected sets for cover:
{2, 3, 4}
{1, 2}
{4, 5}

=== Code Execution Successful ===
```

```python
import re

text = "ACGTACGTACGT"
pattern = r"ACG"
matches = re.findall(pattern, text)
for match in matches:
    print(match)
```

```
ACG
ACG
ACG

=== Code Execution Successful ===
```

main.py

Share Run

```python
denominations = [10, 5, 2, 1]
target = 27
num_coins = 0
i = 0

while target > 0:
    num_coins += target // denominations[i]
    target %= denominations[i]
    i += 1

print(num_coins)
```

Output

```
4

=== Code Execution Successful ===
```

ain.py

Share Run

Output

```python
def assembly_line_scheduling(a, t, e, x):
    n = len(a[0])
    T1 = [0] * n
    T2 = [0] * n
    T1[0] = e[0] + a[0][0]
    T2[0] = e[1] + a[1][0]
    for i in range(1, n):
        T1[i] = min(T1[i - 1] + a[0][i], T2[i - 1] + t[1][i] + a[0][i])
        T2[i] = min(T2[i - 1] + a[1][i], T1[i - 1] + t[0][i] + a[1][i])
    return min(T1[n - 1] + x[0], T2[n - 1] + x[1])
a = [[4, 5, 3, 2], [2, 10, 1, 4]]
t = [[0, 7, 4, 5], [0, 9, 2, 8]]
e = [10, 12]
x = [18, 7]
min_time = assembly_line_scheduling(a, t, e, x)
print("Minimum time to process all stations:", min_time)
```

```
Minimum time to process all stations: 35

=== Code Execution Successful ===
```

main.py

Share Run

```python
n = 4
dist = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
VISITED_ALL = (1 << n) - 1
memo = [[-1] * n for _ in range(1 << n)]
mask = 1
pos = 0
min_cost = float('inf')
stack = [(mask, pos)]
while stack:
    mask, pos = stack.pop()
    if mask == VISITED_ALL:
        cost = dist[pos][0]
    elif memo[mask][pos] !=-1:
        cost = memo[mask][pos]
    else:
        min_cost = float('inf')
        for city in range(n):
            if mask & (1 << city) == 0:
                new_mask = mask | (1 << city)
                new_pos = city
                stack.append((new_mask, new_pos))
                new_cost = dist[pos][city] + memo[new_mask][new_pos]
                min_cost = min(min_cost, new_cost)
        memo[mask][pos] = min_cost
        cost = min_cost
print(memo[1][0] if memo[1][0] != -1 else "Error in Calculation")
```

Output

```
9

=== Code Executi
```

main.py

Share Run

```python
import math
def binomial_coefficient(n, k):
    if k > n:
        return 0
    if k == 0 or k == n:
        return 1
    k = min(k, n - k)
    numerator = math.factorial(n)
    denominator = math.factorial(k) * math.factorial(n - k)
    return numerator // denominator
n = 5
k = 2
print(f"The binomial coefficient C({n}, {k}) is {binomial_coefficient(n, k)}")
```

Output

```
The binomial coefficient C(5, 2) is 10

=== Code Execution Successful ===
```

.py

Output

```
ef merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]
    left_sorted = merge_sort(left_half)
    right_sorted = merge_sort(right_half)
    return merge(left_sorted, right_sorted)

ef merge(left, right):
    sorted_list = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            sorted_list.append(left[i])
            i += 1
        else:
            sorted_list.append(right[j])
            j += 1
    sorted_list.extend(left[i:])
    sorted_list.extend(right[j:])

    return sorted_list
rray = [38, 27, 43, 3, 9, 82, 10]
orted_array = merge_sort(array)
rint("Sorted array:", sorted_array)
```

```
Sorted array: [3, 9, 10, 27, 38, 43, 82]

=== Code Execution Successful ===
```

```python
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.edges = []
    def add_edge(self, u, v, w):
        self.edges.append((u, v, w))
    def bellman_ford(self, src):
        dist = [float("inf")] * self.V
        dist[src] = 0
        for _ in range(self.V - 1):
            for u, v, w in self.edges:
                if dist[u] != float("inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
        for u, v, w in self.edges:
            if dist[u] != float("inf") and dist[u] + w < dist[v]:
                print("Graph contains negative weight cycle")
                return None
        return dist
g = Graph(5)
g.add_edge(0, 1, -1)
g.add_edge(0, 2, 4)
g.add_edge(1, 2, 3)
g.add_edge(1, 3, 2)
g.add_edge(1, 4, 2)
g.add_edge(3, 2, 5)
g.add_edge(3, 1, 1)
g.add_edge(4, 3, -3)
distances = g.bellman_ford(0)
if distances:
    print("Vertex Distance from Source")
    for i, dist in enumerate(distances):
        print(f"{i}\t\t{dist}")
```

```
Vertex Distance from Source
0       0
1       -1
2       2
3       -2
4       1

=== Code Execution Successful ===
```

```python
def optimal_bst(keys, freq):
    n = len(keys)
    cost = [[0] * n for _ in range(n)]
    sum_freq = [[0] * n for _ in range(n)]
    for i in range(n):
        sum_freq[i][i] = freq[i]
        for j in range(i + 1, n):
            sum_freq[i][j] = sum_freq[i][j - 1] + freq[j]
    for length in range(1, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if length == 1:
                cost[i][j] = freq[i]
            else:
                cost[i][j] = float('inf')
                for r in range(i, j + 1):
                    c = (cost[i][r - 1] if r > i else 0) + \
                        (cost[r + 1][j] if r < j else 0) + \
                        sum_freq[i][j]
                    if c < cost[i][j]:
                        cost[i][j] = c
    return cost[0][n - 1]
keys = [10, 12, 20]
freq = [34, 8, 50]
print("Optimal cost of BST:", optimal_bst(keys, freq))
```

```
Optimal cost of BST: 142

=== Code Execution Successful ===
```

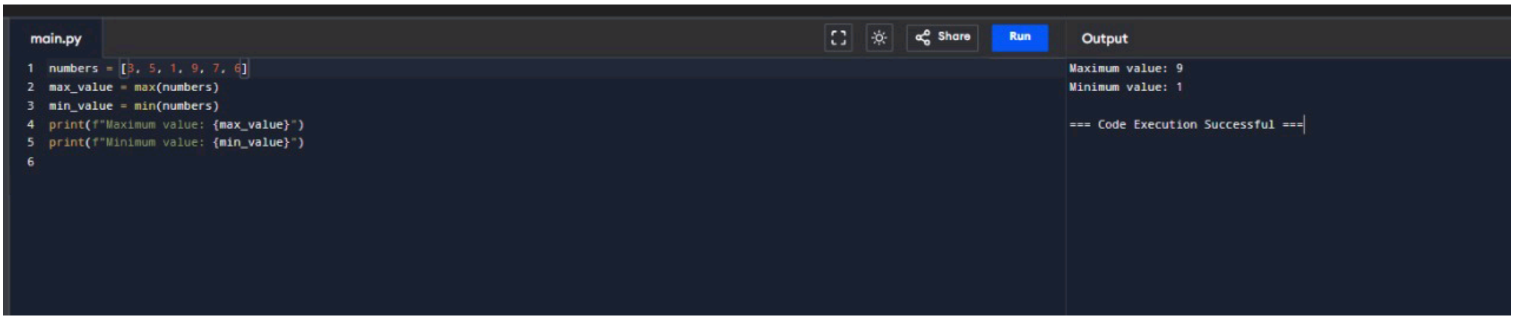
```python
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0] * vertices for _ in range(vertices)]

    def add_edge(self, u, v):
        self.graph[u][v] = 1
        self.graph[v][u] = 1

    def is_safe(self, v, color, c):
        for i in range(self.V):
            if self.graph[v][i] == 1 and color[i] == c:
                return False
        return True
    def graph_coloring_util(self, m, color, v):
        if v == self.V:
            return True
        for c in range(1, m + 1):
            if self.is_safe(v, color, c):
                color[v] = c
                if self.graph_coloring_util(m, color, v + 1):
                    return True
                color[v] = 0
        return False
    def graph_coloring(self, m):
        color = [0] * self.V
        if not self.graph_coloring_util(m, color, 0):
            print("Solution does not exist")
            return False
        print("Solution exists: Following are the assigned colors:")
        for c in color:
            print(c, end=' ')
        print()
        return True

g = Graph(4)
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(1, 3)

m = 3  # Number of colors
g.graph_coloring(m)
```

```
Solution exists: Following are the assigned colors:
1 2 3 1

=== Code Execution Successful ===
```

Airplane mode on

```python
def karatsuba(x, y):
    # Convert numbers to strings for easier manipulation
    x_str = str(x)
    y_str = str(y)

    # Base case for recursion
    if len(x_str) == 1 or len(y_str) == 1:
        return x * y

    # Calculate the size of the numbers
    n = max(len(x_str), len(y_str))
    half_n = n // 2

    # Split the numbers into halves
    x1, x0 = divmod(x, 10 ** half_n)
    y1, y0 = divmod(y, 10 ** half_n)

    # Recursively compute three products
    z2 = karatsuba(x1, y1)
    z0 = karatsuba(x0, y0)
    z1 = karatsuba(x1 + x0, y1 + y0) - z2 - z0

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

# Example usage
a = 1234
b = 5678
result = karatsuba(a, b)
print(f"The product of {a} and {b} is {result}")
```

```
The product of 1234 and 5678 is 7006652

=== Code Execution Successful ===
```


main.py
Share
Run
Output
numbers = [3, 5, 1, 9, 7, 6]
max_value = max(numbers)
min_value = min(numbers)
print(f"Maximum value: {max_value}")
print(f"Minimum value: {min_value}")
Maximum value: 9
Minimum value: 1
=== Code Execution Successful ===

main.py

```python
X = "ABCBDAB"
Y = "BDCAB"
m = len(X)
n = len(Y)
dp = [[0 for _ in range(n+1)] for _ in range(m+1)]
for i in range(1, m+1):
    for j in range(1, n+1):
        if X[i-1] == Y[j-1]:
            dp[i][j] = dp[i-1][j-1] + 1
        else:
            dp[i][j] = max(dp[i-1][j], dp[i][j-1])
i = m
j = n
lcs = ""
while i > 0 and j > 0:
    if X[i-1] == Y[j-1]:
        lcs = X[i-1] + lcs
        i -= 1
        j -= 1
    elif dp[i-1][j] > dp[i][j-1]:
        i -= 1
    else:
        j -= 1
print("Longest Palindromic Subsequence:", lcs)
print("Length:", len(lcs))
```

Share Run

Output

```
Longest Palindromic Subsequence: BDAB
Length: 4

=== Code Execution Successful ===
```