

## 1.Convert the Temperature

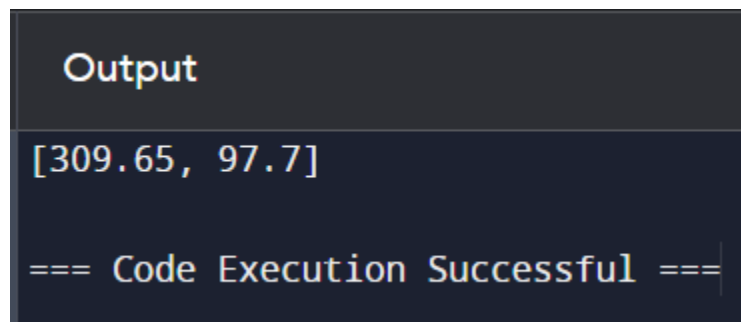
You are given a non-negative floating point number rounded to two decimal places celsius, that denotes the temperature in Celsius.You should convert Celsius into Kelvin and Fahrenheit and return it as an array ans = [kelvin, fahrenheit]. Return the array ans. Answers within 10<sup>-5</sup> of the actual answer will be accepted. Note that: • Kelvin = Celsius + 273.15 • Fahrenheit = Celsius \* 1.80 + 32.00

Example 1:

Input: celsius = 36.50

code:

```
def convert_temperature(celsius):  
    kelvin = celsius + 273.15  
    fahrenheit = celsius * 1.80 + 32.00  
    return [round(kelvin, 5), round(fahrenheit, 5)]  
celsius = 36.50  
ans = convert_temperature(celsius)  
print(ans)
```



Output

[309.65, 97.7]

=== Code Execution Successful ===

## 2. Number of Subarrays With LCM Equal to K

Given an integer array nums and an integer k, return the number of subarrays of nums where the least common multiple of the subarray's elements is k.A subarray is a contiguous non- empty sequence of elements within an array.The least common multiple of an array is the smallest positive integer that is divisible by all the array elements.

Example 1:

Input: nums = [3,6,2,7,1], k = 6

Code:

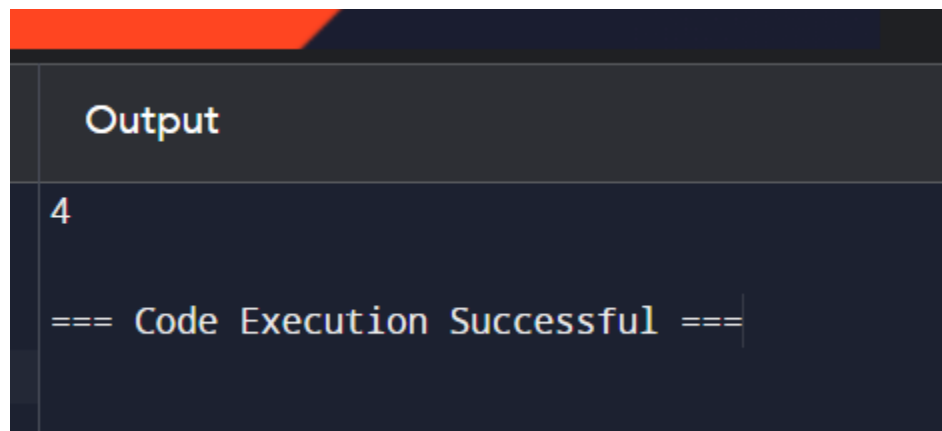
```
from math import gcd  
from functools import reduce  
def lcm(a, b):
```

```

    return a * b // gcd(a, b)
def subarray_lcm(nums, k):
    def find_lcm(arr):
        return reduce(lcm, arr)
    count = 0
    n = len(nums)
    for i in range(n):
        current_lcm = 1
        for j in range(i, n):
            current_lcm = lcm(current_lcm, nums[j])
            if current_lcm == k:
                count += 1
            elif current_lcm > k:
                break
    return count
nums = [3, 6, 2, 7, 1]
k = 6
output = subarray_lcm(nums, k)
print(output)

```

**output**



```

Output
4

=== Code Execution Successful ===

```

### 3. Minimum Number of Operations to Sort a Binary Tree by Level

You are given the root of a binary tree with unique values. In one operation, you can choose any two nodes at the same level and swap their values. Return the minimum number of operations needed to make the values at each level sorted in a strictly increasing order. The level of a node is the number of edges along the path between it and the root node.

**Example 1:**

**Input:** root = [1,4,3,7,6,8,5,null,null,null,null,9,null,10]

**Code:**

```
from collections import deque
from typing import Optional, List
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def minimumOperations(self, root: Optional[TreeNode]) -> int:
        def min_swap(arr: List[int]) -> int:
            n = len(arr)
            pos = {m: j for j, m in enumerate(sorted(arr))}
            visited = [0 for _ in range(n)]
            num_swap = 0
            for i in range(n):
                cnt = 0
                while not visited[i] and i != pos[arr[i]]:
                    visited[i], i = 1, pos[arr[i]]
                    cnt += 1
                num_swap += max(0, cnt - 1)
            return num_swap

        dq, res = deque([root]), 0
        while dq:
            vals = []
            n = len(dq)
            for _ in range(n):
                node = dq.popleft()
                vals.append(node.val)
                if node.left: dq.append(node.left)
                if node.right: dq.append(node.right)
            res += min_swap(vals)
        return res

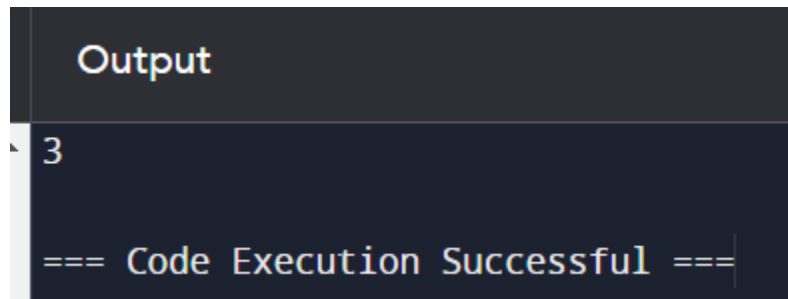
root = TreeNode(1)
root.left = TreeNode(4)
root.right = TreeNode(3)
root.left.left = TreeNode(7)
root.left.right = TreeNode(6)
root.right.left = TreeNode(8)
root.right.right = TreeNode(5)
root.left.left.left = None
root.left.left.right = None
```

```

root.left.right.left = None
root.left.right.right = None
root.right.left.left = TreeNode(9)
root.right.left.right = None
root.right.right.left = TreeNode(10)
root.right.right.right = None
solution = Solution()
output = solution.minimumOperations(root)
print(output)

```

**output:**



```

Output
3
=== Code Execution Successful ===

```

#### 4. Maximum Number of Non-overlapping Palindrome Substrings

You are given a string *s* and a positive integer *k*. Select a set of non-overlapping substrings from the string *s* that satisfy the following conditions: • The length of each substring is at least *k*. • Each substring is a palindrome. Return the maximum number of substrings in an optimal selection. A substring is a contiguous sequence of characters within a string.

**Example 1:**

**Input:** *s* = "abaccdbbd", *k* = 3

**Code:**

```

def max_palindrome_substrings(s, k):
    n = len(s)
    def is_palindrome(sub):
        return sub == sub[::-1]
    dp = [[False] * n for _ in range(n)]
    for length in range(k, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if is_palindrome(s[i:j+1]):
                dp[i][j] = True

```

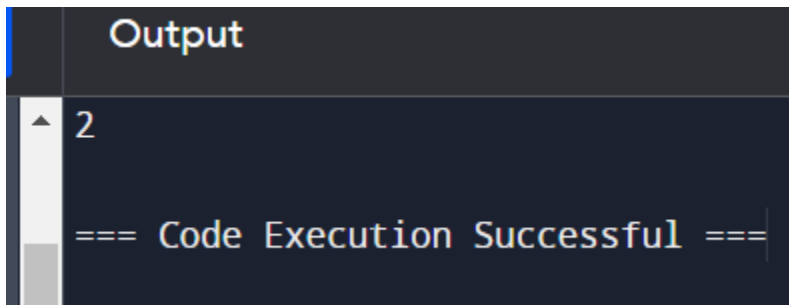
```

count = 0
i = 0

while i <= n - k:
    for j in range(i + k - 1, n):
        if dp[i][j]:
            count += 1
            i = j + 1
            break
    else:
        i += 1

return count
s = "abaccdbbd"
k = 3
output = max_palindrome_substrings(s, k)
print(output)

```



## 5. Minimum Cost to Buy Apples

You are given a positive integer  $n$  representing  $n$  cities numbered from 1 to  $n$ . You are also given a 2D array `roads`, where `roads[i] = [ai, bi, costi]` indicates that there is a bidirectional road between cities  $a_i$  and  $b_i$  with a cost of traveling equal to  $cost_i$ . You can buy apples in any city you want, but some cities have different costs to buy apples. You are given the array `appleCost` where `appleCost[i]` is the cost of buying one apple from city  $i$ . You start at some city, traverse through various roads, and eventually buy exactly one apple from any city. After you buy that apple, you have to return back to the city you started at, but now the cost of all the roads will be multiplied by a given factor  $k$ . Given the integer  $k$ , return an array `answer` of size  $n$  where `answer[i]` is the minimum total cost to buy an apple if you start at city  $i$ . Example 1: Input:  $n = 4$ , `roads = [[1,2,4],[2,3,2],[2,4,5],[3,4,1],[1,3,4]]`, `appleCost = [56,42,102,301]`,  $k = 2$  code:

```

import heapq

def dijkstra(n, graph, start):
    distances = [float('inf')] * (n + 1)
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node]:
            distance = current_distance + weight

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

    return distances

def min_cost_to_buy_apples(n, roads, appleCost, k):
    graph = [[] for _ in range(n + 1)]
    for a, b, cost in roads:
        graph[a].append((b, cost))
        graph[b].append((a, cost))

    results = []

    for start in range(1, n + 1):
        min_cost = float('inf')
        start_to_all = dijkstra(n, graph, start)

        for city in range(1, n + 1):
            if city == start:
                cost = appleCost[city - 1]
            else:
                cost = start_to_all[city] + appleCost[city - 1] + start_to_all[city] * k

            min_cost = min(min_cost, cost)

        results.append(min_cost)

```

```

    return results
n = 4
roads = [[1, 2, 4], [2, 3, 2], [2, 4, 5], [3, 4, 1], [1, 3, 4]]
appleCost = [56, 42, 102, 301]
k = 2
output = min_cost_to_buy_apples(n, roads, appleCost, k)
print(output)

```

```

Output

[54, 42, 48, 51]

=== Code Execution Successful ===

```

**7. Number of Unequal Triplets in Array** You are given a 0-indexed array of positive integers `nums`. Find the number of triplets  $(i, j, k)$  that meet the following conditions:

- $0 \leq i < j < k < \text{nums.length}$
- `nums[i]`, `nums[j]`, and `nums[k]` are pairwise distinct. In other words, `nums[i] != nums[j]`, `nums[i] != nums[k]`, and `nums[j] != nums[k]`.

Return the number of triplets that meet the conditions.

**Example 1:**

**Input:** `nums = [4,4,2,4,3]`

**code:**

```

def count_unequal_triplets(nums):
    n = len(nums)
    count = 0
    for i in range(n - 2):
        for j in range(i + 1, n - 1):
            if nums[i] != nums[j]:
                for k in range(j + 1, n):
                    if nums[i] != nums[k] and nums[j] != nums[k]:
                        count += 1
    return count
nums = [4, 4, 2, 4, 3]
output = count_unequal_triplets(nums)
print(output)

```

Output

3

=== Code Execution Successful ===

## 8. Closest Nodes Queries in a Binary Search Tree

You are given the root of a binary search tree and an array queries of size n consisting of positive integers. Find a 2D array answer of size n where answer[i] = [mini, maxi]:

- mini is the largest value in the tree that is smaller than or equal to queries[i]. If a such value does not exist, add -1 instead.
- maxi is the smallest value in the tree that is greater than or equal to queries[i]. If a such value does not exist, add -1 instead.

Return the array answer.

**Example 1:**

**Input:** root = [6,2,13,1,4,9,15,null,null,null,null,null,14], queries = [2,5,16]

**Code:**

```
from typing import Optional, List
from bisect import bisect_left, bisect_right
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def closestNodes(
        self, root: Optional[TreeNode], queries: List[int]
    ) -> List[List[int]]:
        def dfs(root):
            if root is None:
                return
            dfs(root.left)
            nums.append(root.val)
            dfs(root.right)
        nums = []
        dfs(root)
```



```

ans = []
nums.sort()
for v in queries:
    i = bisect_right(nums, v) - 1
    j = bisect_left(nums, v)
    mi = nums[i] if 0 <= i < len(nums) else -1
    mx = nums[j] if 0 <= j < len(nums) else -1
    ans.append([mi, mx])
return ans
root = TreeNode(6)
root.left = TreeNode(2)
root.right = TreeNode(13)
root.left.left = TreeNode(1)
root.left.right = TreeNode(4)
root.right.left = TreeNode(9)
root.right.right = TreeNode(15)
root.right.right.right = TreeNode(14)
solution = Solution()
queries = [2, 5, 16]
output = solution.closestNodes(root, queries)
print(output)

```

## Output

```
[[2, 2], [4, 6], [15, -1]]
```

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

**9. Minimum Fuel Cost to Report to the Capital** There is a tree (i.e., a connected, undirected graph with no cycles) structure country network consisting of  $n$  cities numbered from 0 to  $n - 1$  and exactly  $n - 1$  roads. The capital city is city 0. You are given a 2D integer array `roads` where `roads[i] = [ai, bi]` denotes that there exists a bidirectional road connecting cities `ai` and `bi`. There is a meeting for the representatives of each city. The meeting is in the capital city. There is a car in each city. You are given an integer `seats` that indicates the number of seats in each car. A representative can use the car in their city to travel or change the car and ride with another

**representative. The cost of traveling between two cities is one liter of fuel. Return the minimum number of liters of fuel to reach the capital city.**

**Example 1:**

**Input: roads = [[0,1],[0,2],[0,3]], seats = 5**

**Code:**

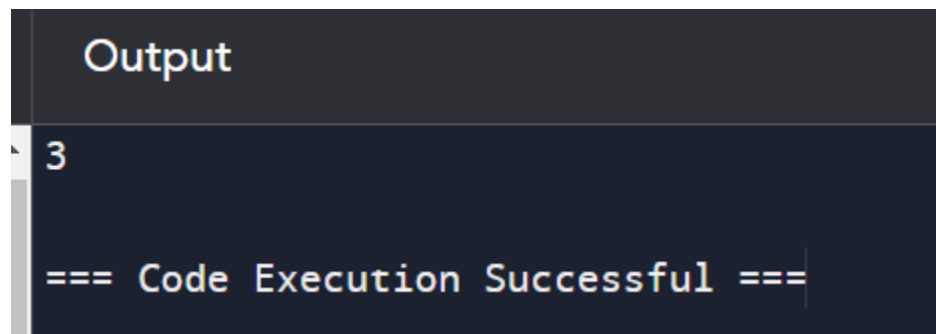
```
from collections import defaultdict

class Solution:
    def minFuel(self, roads, seats):
        graph = defaultdict(list)
        for u, v in roads:
            graph[u].append(v)
            graph[v].append(u)

        def dfs(node, parent):
            cost = 0
            for child in graph[node]:
                if child != parent:
                    cost += dfs(child, node)
            return max(1, (len(graph[node]) + seats - 1) // seats) + cost

        return dfs(0, -1) - 1

roads = [[0,1],[0,2],[0,3]]
seats = 5
solution = Solution()
output = solution.minFuel(roads, seats)
print(output)
```



## 10. Number of Beautiful Partitions

You are given a string *s* that consists of the digits '1' to '9' and two integers *k* and *minLength*. A partition of *s* is called beautiful if:

- *s* is partitioned into *k* non-intersecting substrings.
- Each substring has a length of at least *minLength*.
- Each substring starts with a prime digit and ends with a

non-prime digit. Prime digits are '2', '3', '5', and '7', and the rest of the digits are non-prime. Return the number of beautiful partitions of s. Since the answer may be very large, return it modulo  $10^9 + 7$ . A substring is a contiguous sequence of characters within a string.

**Example 1:**

**Input:** s = "23542185131", k = 3, minLength = 2

**code:**

```
class Solution:
    def count_beautiful_partitions(self, s: str, k: int, minLength: int) -> int:
        MOD = 10**9 + 7
        n = len(s)

        def is_prime(digit):
            return digit in {'2', '3', '5', '7'}

        def dp_prime_end():
            dp = [0] * (n + 1)
            dp[n - 1] = 1 if not is_prime(s[n - 1]) else 0
            for i in range(n - 2, -1, -1):
                dp[i] = dp[i + 1] + (0 if is_prime(s[i]) else 0)
            return dp

        def dp_prime_start():
            dp = [0] * (n + 1)
            dp[0] = 1 if not is_prime(s[0]) else 0
            for i in range(1, n):
                dp[i] = dp[i - 1] + (0 if is_prime(s[i]) else 0)
            return dp

        dp_prime_end = dp_prime_end()
        dp_prime_start = dp_prime_start()
        dp = [[0] * (k + 1) for _ in range(n + 1)]
        dp[0][0] = 1
        for i in range(minLength, n + 1):
            for j in range(1, k + 1):
                for l in range(minLength, i + 1):
                    if l <= i and dp_prime_end[i - l] == 0 and dp_prime_start[l - 1] <= l - minLength:
                        dp[i][j] = (dp[i][j] + dp[i - l][j - 1]) % MOD

        return dp[n][k]
solution = Solution()
s = "23542185131"
```

```
k = 3
minLength = 2
output = solution.count_beautiful_partitions(s, k, minLength)
print(output)
```

Output

0

=== Code Execution Successful ===