In [88]:

```python
import random

# Problem 1

from operator import itemgetter

def minimumCost(n: int, connections: list[list[int]]):
    # sorting the connection array based on cost in increasing order
    connections.sort(key=itemgetter(2))
    # initializing variables
    p = [k for k in range(n+1)]
    ct = 0
    collective_cost = 0

    for c in connections:
        x, y, cst = c

        # finding the parents of x and y
        p_x = get_parent(p, x)
        p_y = get_parent(p, y)

        # Include in minimum total cost in case parents are different
        if p_x != p_y:
            collective_cost += cst
            p[p_x] = p_y
            ct += 1
        if ct == n-1:
            return collective_cost

    # if all cities are not connected then returning -1
    return -1

def get_parent(p, node):
    # finding the parent of node recursively
    if p[node] != node:
        p[node] = get_parent(p, p[node])
    return p[node]
```

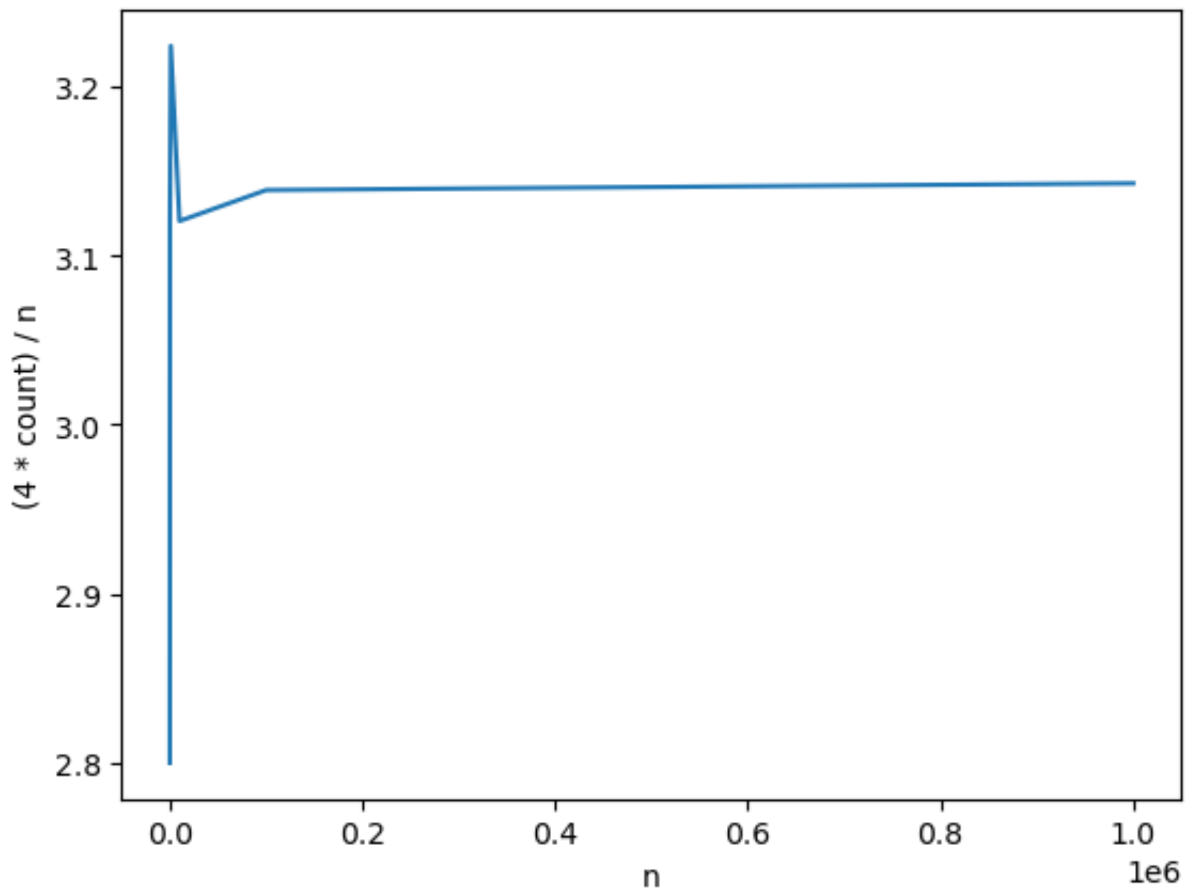
In [89]:
```python
# Problem 2

def is_feasible_graph(n, edges_list):
    #When there is only one node
    if n == 1:
        return True

    # Implementing Depth First Search for coloring the nodes
    def dfs(nd, cl):
        clr[nd] = cl
        vsted[nd] = True
        for nbr in adjacent[nd]:
            if not vsted[nbr]:
                if not dfs(nbr, 1-cl):
                    return False
            elif clr[nbr] == cl:
                return False
        return True

    #Intializing the adjacency matrix
    adjacent = [[] for _ in range(n)]
    for s, d in edges_list:
        adjacent[s].append(d)
        adjacent[d].append(s)

    #Initializing the variables
    clr = [-1] * n
    vsted = [False] * n

    # Go through every node and give them a color if they have not been colored yet.
    for k in range(n):
        if not vsted[k]:
            if not dfs(k, 0):
                return False
    return True
```

In [90]:
```
# Problem 3

def word2int(word):
    val = 0
    for k, c in enumerate(word[::-1]):
        val = val + (ord(c) - ord('a') + 1) * (32 ** k)
    return val

def get_min_C_value(words):
    # Get the word2int values of the words
    w2i = []
    for word in words:
        sum = 0
        for k, c in enumerate(reversed(word)):
            sum += word2int(c) * 32 ** k
        w2i.append(sum)

    c = 1
    while True:
        # Creating a hash table
        t = [-1] * len(words)

        for k in range(len(words)):
            #Calculating the hash value
            h_val = (c // w2i[k]) % len(words)
            #Incase of collision stop and try different value
            if t[h_val] != -1: break
            t[h_val] = k
        # If there are no collisions, then return the value of c.
        else:
            return c
        c += 1
```

In [101…

```
#Problem 4
import random

import matplotlib.pyplot as plt

n_list = [10, 100, 1000, 10000, 100000, 1000000]
rts = []
for n in n_list:
    #Generating the pints
    ct = 0
    n1 = n
    while n1 != 0:
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)
        if (x**2 + y**2) < 1:
            ct += 1
        n1 = n1 - 1

    rt = (4 * ct) / n
    rts.append(rt)
    print("n =", n, "ratio =", rt)

plt.plot(n_list, rts)
plt.xlabel("n")
plt.ylabel("(4 * count) / n")
plt.show()


#Observation:

# As n increases, the value of (4 * count) / n tends to approach pi.
# For small values of n, the ratios exhibit some degree of variation.
# But as n becomes larger, the variation diminishes and the ratios converge to a singular value.
# The rate of convergence to pi is gradual, such that even with n=1,000,000, the ratio approximates pi with only about two decimal places of accuracy.
# Consequently, generating random points is not an especially efficient means of computing pi, although it serves as a straightforward and intuitive illustration of the
```

```
n = 10 ratio = 2.8
n = 100 ratio = 3.12
n = 1000 ratio = 3.224
n = 10000 ratio = 3.1204
n = 100000 ratio = 3.13872
n = 1000000 ratio = 3.142688
```



In [ ]: