Toán UDTK | Final Project: AMAS

Sinh viên thực hiện:

MSSV: 23120262

MSSV: Tống Dương Thái Hòa

Giáo viên hướng dẫn:

CN. Võ Nam Thục Đoan

ThS. Trần Hà Sơn

ThS. Nguyễn Hữu Toàn

Lê Trọng Anh Tú

Các Class và Hàm cơ bản

In [1]:
```python
import numpy as np
from fractions import Fraction

# Hàm định dạng số, hiển thị Fraction nếu có, nếu không thì là số thập phân
def formatNumber(num) -> str:
    """
    Định dạng một số (Fraction hoặc float) thành chuỗi để hiển thị.
    Nếu là Fraction và mẫu số là 1, hiển thị số nguyên.
    Nếu là Fraction, hiển thị dạng phân số.
    Nếu là float, hiển thị 6 chữ số thập phân.
    """
    if isinstance(num, Fraction):
        if num.denominator == 1:
            return str(int(num))
        return f"{num.numerator}/{num.denominator}"
    return "{:.6f}".format(float(num))

class Ma_Tran:
    """
    Class đại diện cho một ma trận và cung cấp các phép toán ma trận cơ bản.
    """
    def __init__(self, arg):
        """
        Khởi tạo một đối tượng Matrix.
        Có thể khởi tạo từ:
        - Một danh sách các danh sách (2D list) đại diện cho ma trận.
        - Một tuple (rows, columns) để tạo ma trận không (zeros matrix).
        """
        if isinstance(arg, list):
            if not arg or not arg[0]:
                raise ValueError("Matrix cannot be empty.")
            self.matrix = arg
            self.rows = len(arg)
            self.columns = len(arg[0])
            for row in arg:
                if len(row) != self.columns:
                    raise ValueError("All rows in the matrix must have the same number of columns.")
        elif isinstance(arg, tuple) and len(arg) == 2:
            self.rows, self.columns = arg
            self.matrix = [[0 for _ in range(self.columns)] for _ in range(self.rows)]
        else:
            raise ValueError("Invalid argument type. Must be a list (2D matrix) or a tuple (rows, columns).")

    def pprint(self):
        """
        In ma trận ra console với định dạng đẹp, căn chỉnh các số.
        Ưu tiên hiển thị dưới dạng phân số để giữ độ chính xác.
        """
        # Tìm độ dài lớn nhất của chuỗi định dạng cho mỗi số trong ma trận
        max_len = 0
        for row in self.matrix:
            for num in row:
                max_len = max(max_len, len(formatNumber(num)))

        # In ma trận với các số được căn chỉnh
        for i in range(self.rows):
            for j in range(self.columns):
```

```python
                print(f"{formatNumber(self.matrix[i][j]):>{max_len}}", end = " ")
            print()

    def __mul__(self, other):
        """
        Thực hiện phép nhân ma trận với ma trận khác hoặc ma trận với một số vô hướng.
        """
        if isinstance(other, Ma_Tran):
            # Nhân ma trận với ma trận
            if self.columns != other.rows:
                raise ValueError("Số cột của ma trận thứ nhất phải bằng số hàng của ma trận thứ hai để nhân.")

            res = [[0 for _ in range(other.columns)] for _ in range(self.rows)]
            for i in range(self.rows):
                for j in range(other.columns):
                    total_sum = 0
                    for k in range(self.columns):
                        total_sum += self.matrix[i][k] * other.matrix[k][j]
                    res[i][j] = total_sum
            return Ma_Tran(res)
        elif isinstance(other, (int, float, Fraction)):
            # Nhân ma trận với một số vô hướng
            res = [[self.matrix[i][j] * other for j in range(self.columns)] for i in range(self.rows)]
            return Ma_Tran(res)
        else:
            raise TypeError("Phép nhân không hỗ trợ với kiểu dữ liệu này.")

    def __pow__(self, n: int) -> 'Ma_Tran':
        """
        Tính lũy thừa ma trận (chỉ áp dụng cho ma trận vuông).
        n = 0: Trả về ma trận đơn vị.
        n = 1: Trả về chính ma trận đó.
        n > 1: Thực hiện nhân ma trận n lần.
        """
        if self.rows != self.columns:
            raise ValueError("Chỉ ma trận vuông mới có thể tính lũy thừa.")
        if not isinstance(n, int) or n < 0:
            raise ValueError("Số mũ phải là số nguyên không âm.")

        if n == 0:
            # Trả về ma trận đơn vị (identity matrix)
            identity_matrix_list = [[(1 if i == j else 0) for j in range(self.columns)] for i in range(self.rows
            return Ma_Tran(identity_matrix_list)
        elif n == 1:
            return self
        else:
            result = self
            for _ in range(n - 1):
                result = result * self
            return result

    def inv(self) -> 'Ma_Tran':
        """
        Tính ma trận nghịch đảo của ma trận vuông.
        Sử dụng thuật toán khử Gauss-Jordan để tìm ma trận nghịch đảo.
        Chấp nhận Fraction và float.
        """
        if self.rows != self.columns:
            raise ValueError("Chỉ ma trận vuông mới có thể tìm nghịch đảo.")

        n = self.rows
        # Tạo ma trận mở rộng [A|I]
        # Chuyển tất cả các phần tử sang Fraction để tránh lỗi số thập phân
        augmented_matrix = [[Fraction(val) for val in row] + [Fraction(1 if i == j else 0) for j in range(n)]
                            for i, row in enumerate(self.matrix)]

        for i in range(n):
            # Chọn phần tử chốt (pivot) - tìm hàng có giá trị tuyệt đối lớn nhất trong cột i
            pivot_row = i
            for k in range(i + 1, n):
                if abs(augmented_matrix[k][i]) > abs(augmented_matrix[pivot_row][i]):
                    pivot_row = k

            # Hoán đổi hàng pivot nếu cần
            augmented_matrix[i], augmented_matrix[pivot_row] = augmented_matrix[pivot_row], augmented_matrix[i]

            pivot_val = augmented_matrix[i][i]
            if pivot_val == 0:
                raise ValueError("Ma trận không khả nghịch (singular matrix).")

            # Chuẩn hóa hàng pivot để phần tử chốt bằng 1
            for j in range(2 * n):
                augmented_matrix[i][j] /= pivot_val
```

```python
            # Khử các phần tử khác trong cột pivot
            for k in range(n):
                if k != i:
                    factor = augmented_matrix[k][i]
                    for j in range(2 * n):
                        augmented_matrix[k][j] -= factor * augmented_matrix[i][j]

        # Trích xuất ma trận nghịch đảo từ phần bên phải của ma trận mở rộng
        inverse_matrix_list = [row[n:] for row in augmented_matrix]
        return Ma_Tran(inverse_matrix_list)

    def transpose(self) -> 'Ma_Tran':
        """
        Tính ma trận chuyển vị.
        """
        res = [[0 for _ in range(self.rows)] for _ in range(self.columns)]
        for i in range(self.rows):
            for j in range(self.columns):
                res[j][i] = self.matrix[i][j]
        return Ma_Tran(res)

class Vector:
    """
    Class đại diện cho một vector (một danh sách 1D) và cung cấp các phép toán cơ bản.
    """
    def __init__(self, data: list):
        """
        Khởi tạo một đối tượng Vector từ một danh sách (list).
        """
        if not data:
            raise ValueError("Vector cannot be empty.")
        self.data = data
        self.size = len(data)

    def __str__(self) -> str:
        """
        Trả về biểu diễn chuỗi của vector.
        """
        return "[" + ", ".join([formatNumber(x) for x in self.data]) + "]"

    def to_matrix(self) -> Ma_Tran:
        """
        Chuyển đổi vector thành một ma trận cột (Nx1).
        """
        return Ma_Tran([[x] for x in self.data])
```

---

a) Xây dựng ma trận chuyển trạng thái P và Xác định vector phân phối xác suất ban đầu π_0

Mô tả biến ngẫu nhiên X_n phù hợp cho bài toán trên mà có tính chất Markov:

Đề bài cho biết:

- Mỗi lần tung xúc xắc, giá trị cộng thêm vào tổng S_n là một số từ 1 đến 6.
- Ta muốn khảo sát phân phối của giá trị phần dư của S_n khi chia cho 7.

Gọi $X_n = S_n \pmod 7$. Biến ngẫu nhiên X_n có thể nhận các giá trị trong tập trạng thái $S = \{0, 1, 2, 3, 4, 5, 6\}$. Tính chất Markov được thỏa mãn vì xác suất để X_{n+1} chuyển sang một trạng thái nào đó *chỉ phụ thuộc vào trạng thái hiện tại của X_n*, chứ không phụ thuộc vào các trạng thái trước đó $(X_0, X_1, \ldots, X_{n-1})$. Điều này là do mỗi lần tung xúc xắc là một sự kiện độc lập.

Xác định ma trận chuyển trạng thái P:

Ma trận chuyển trạng thái P (kích thước 7×7) có các phần tử p_{ij} biểu thị xác suất chuyển từ trạng thái i sang trạng thái j sau một lần tung xúc xắc. ($P_{row, column}$ theo quy ước phổ biến, khác với $P_{i,j}$ trong đề bài là $P_{destination, source}$). Trong trường hợp này, nếu hệ thống đang ở trạng thái i (tức $S_n \pmod 7 = i$), và kết quả tung xúc xắc là $k \in \{1, 2, 3, 4, 5, 6\}$ (mỗi giá trị có xác suất $1/6$), thì trạng thái tiếp theo sẽ là $(i + k) \pmod 7$.

Do đó, từ mỗi trạng thái i, có 6 trạng thái có thể đến được là $(i + 1) \pmod 7, (i + 2) \pmod 7, \ldots, (i + 6) \pmod 7$. Mỗi trạng thái này có xác suất $1/6$. Trạng thái $(i + 7) \pmod 7$ cũng chính là i, nhưng không thể đạt được bằng một lần tung xúc xắc (vì xúc xắc không có mặt 7). Vì vậy, xác suất chuyển từ i về chính i là 0.

Ví dụ:

- Từ trạng thái 0: có thể đến 1, 2, 3, 4, 5, 6 (mỗi cái xác suất 1/6).
- Từ trạng thái 1: có thể đến 2, 3, 4, 5, 6, 0 (mỗi cái xác suất 1/6).

Ma trận P sẽ có dạng:

$$P = \begin{bmatrix} 0 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 0 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 0 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 0 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 0 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 0 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 0 \end{bmatrix}$$

In [2]:
```
def create_transition_matrix_mod7() -> Ma_Tran:
    """
    Tạo ma trận chuyển trạng thái P cho quá trình Markov với X_n = S_n mod 7,
    trong đó mỗi bước cộng thêm giá trị từ 1 đến 6 (mỗi giá trị có xác suất 1/6).
    """
    size = 7
    # Khởi tạo ma trận P với tất cả các phần tử là Fraction(0) để đảm bảo độ chính xác
    P_list = [[Fraction(0) for _ in range(size)] for _ in range(size)]

    # Duyệt qua từng trạng thái hiện tại (hàng i)
    for i in range(size):
        # Duyệt qua các kết quả có thể của xúc xắc (k từ 1 đến 6)
        for k in range(1, 7):
            # Tính trạng thái kế tiếp (j)
            j = (i + k) % size
            # Cộng xác suất chuyển từ i sang j
            P_list[i][j] += Fraction(1, 6)

    return Ma_Tran(P_list)

P = create_transition_matrix_mod7()
print("Ma trận chuyển trạng thái P là:")
P.pprint()
```

```
Ma trận chuyển trạng thái P là:
  0 1/6 1/6 1/6 1/6 1/6 1/6
1/6   0 1/6 1/6 1/6 1/6 1/6
1/6 1/6   0 1/6 1/6 1/6 1/6
1/6 1/6 1/6   0 1/6 1/6 1/6
1/6 1/6 1/6 1/6   0 1/6 1/6
1/6 1/6 1/6 1/6 1/6   0 1/6
1/6 1/6 1/6 1/6 1/6 1/6   0
```

Xác định vector phân phối xác suất ban đầu π_0:

Tại thời điểm ban đầu (trước khi tung xúc xắc lần nào), tổng các kết quả $S_0 = 0$. Do đó, phần dư của S_0 khi chia cho 7 là $X_0 = S_0 \pmod 7 = 0$. Điều này có nghĩa là hệ thống chắc chắn ở trạng thái "Dư 0" và xác suất ở các trạng thái khác là 0.

Vector phân phối xác suất ban đầu π_0 (vector cột 7×1) sẽ là:

$$\pi_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

In [3]:
```
def initial_distribution_vector() -> Vector:
    """
    Trả về vector phân phối xác suất ban đầu π0,
    tương ứng với trạng thái bắt đầu tại phần dư 0.
    """
    # Vector cột với xác suất 1 tại trạng thái 0, 0 ở các trạng thái khác
    return Vector([Fraction(1), Fraction(0), Fraction(0), Fraction(0), Fraction(0), Fraction(0), Fraction(0)])

pi0 = initial_distribution_vector()
print("\nPhân phối xác suất ban đầu π0 là:")
pi0.to_matrix().pprint() # In dưới dạng ma trận cột để dễ nhìn
```

```
Phân phối xác suất ban đầu π0 là:
1
0
0
0
0
0
0
```

---

b) Viết hàm dùng để tính xác suất xuất hiện các giá trị phần dư của S_n khi chia cho 7 theo bảng sau

Ý tưởng:

Để tính toán phân phối xác suất π_n tại bước n (sau n lần tung xúc xắc), chúng ta sử dụng công thức cơ bản của chuỗi Markov:

$$\pi_n = P^n \times \pi_0$$

Trong đó:

- P: Ma trận chuyển trạng thái.
- π_0: Vector phân phối xác suất ban đầu.
- P^n: Ma trận P được lũy thừa n lần.

Hàm sẽ lặp từ $n = 1$ đến $n = n_{max}$, tính toán π_n cho mỗi bước, và hiển thị kết quả trong một bảng.

In [4]:
```
def compute_mod7_distributions(n_max: int = 10):
    """
    Tính toán và hiển thị xác suất xuất hiện các giá trị phần dư của Sn khi chia cho 7
    cho n từ 1 đến n_max.
    """
    P = create_transition_matrix_mod7()
    pi0 = initial_distribution_vector()

    # In tiêu đề bảng
    header_n = f"{'n':>3}"
    headers_mod = [f"     S_n % 7 = {r}    " for r in range(7)]
    print(header_n + " | " + " | ".join(headers_mod) + " | ")
    print("-" * (len(header_n) + 3 + len(" | ".join(headers_mod)) + 3))

    # Tính toán và in dữ liệu cho từng n
    for n in range(1, n_max + 1):
        # Tính P^n
        P_n = P ** n
        # Tính π_n = P^n * π_0 (kết quả là một ma trận cột)
        pi_n_matrix = P_n * pi0.to_matrix()

        # In số bước n
        print(f"{n:3}", end=" | ")
        # In các xác suất cho từng trạng thái (phần dư)
        for i in range(pi_n_matrix.rows):
            # Chuyển Fraction sang float để định dạng số thập phân, dễ đọc hơn trong bảng lớn
            val = pi_n_matrix.matrix[i][0]
            print(f"{formatNumber(float(val)):>18}", end=" | ")
        print()

print("Bảng xác suất xuất hiện các giá trị phần dư của Sn khi chia cho 7:")
compute_mod7_distributions(10)
```

```
Bảng xác suất xuất hiện các giá trị phần dư của Sn khi chia cho 7:
  n |     S_n % 7 = 0     |     S_n % 7 = 1     |     S_n % 7 = 2     |     S_n % 7 = 3     |     S_n % 7 = 4     |
S_n % 7 = 5     |     S_n % 7 = 6     |
---------------------------------------------------------------------------------------------------------------------
-----------------------------------------
  1 |           0.000000 |           0.166667 |           0.166667 |           0.166667 |           0.166667 |
0.166667 |           0.166667 |
  2 |           0.166667 |           0.138889 |           0.138889 |           0.138889 |           0.138889 |
0.138889 |           0.138889 |
  3 |           0.138889 |           0.143519 |           0.143519 |           0.143519 |           0.143519 |
0.143519 |           0.143519 |
  4 |           0.143519 |           0.142747 |           0.142747 |           0.142747 |           0.142747 |
0.142747 |           0.142747 |
  5 |           0.142747 |           0.142876 |           0.142876 |           0.142876 |           0.142876 |
0.142876 |           0.142876 |
  6 |           0.142876 |           0.142854 |           0.142854 |           0.142854 |           0.142854 |
0.142854 |           0.142854 |
  7 |           0.142854 |           0.142858 |           0.142858 |           0.142858 |           0.142858 |
0.142858 |           0.142858 |
  8 |           0.142858 |           0.142857 |           0.142857 |           0.142857 |           0.142857 |
0.142857 |           0.142857 |
  9 |           0.142857 |           0.142857 |           0.142857 |           0.142857 |           0.142857 |
0.142857 |           0.142857 |
 10 |           0.142857 |           0.142857 |           0.142857 |           0.142857 |           0.142857 |
0.142857 |           0.142857 |
```

---

c) Kiểm tra sự tồn tại của phân phối dừng và tìm thời điểm hội tụ

Khái niệm Phân phối dừng (Stationary Distribution):

Phân phối dừng π_L là một vector xác suất mà khi chuỗi Markov đạt đến phân phối này, nó sẽ duy trì trạng thái đó qua các bước chuyển tiếp tiếp theo. Nghĩa là, nếu $\pi_n = \pi_L$, thì $\pi_{n+1} = \pi_L$ cũng. Công thức toán học của phân phối dừng là:

$$\pi_L = \pi_L P$$

Nếu chuỗi Markov là bất khả quy (irreducible) và phi chu kỳ (aperiodic) – điều kiện thường gặp trong nhiều bài toán thực tế – thì một phân phối dừng duy nhất sẽ tồn tại và chuỗi sẽ hội tụ về phân phối đó theo thời gian, bất kể phân phối ban đầu là gì.

Ý tưởng kiểm tra và tìm thời điểm hội tụ:

Chúng ta sẽ sử dụng phương pháp lặp để kiểm tra sự hội tụ:

1. Bắt đầu với một phân phối ban đầu (π_0 hoặc bất kỳ vector xác suất nào).
2. Tính toán phân phối ở bước tiếp theo: $\pi_{t+1} = P \times \pi_t$.
3. So sánh π_{t+1} với π_t. Nếu sự khác biệt giữa các phần tử của hai vector này nhỏ hơn một ngưỡng tolerance (`tol`) cho trước, ta coi như chuỗi đã hội tụ.
4. Thời điểm t mà tại đó sự hội tụ xảy ra chính là thời điểm phân phối xác suất trở nên ổn định.

In [5]:
```python
def find_stationary_distribution(P: Ma_Tran, pi0: Vector, max_steps: int = 1000, tol: float = 1e-10) -> tuple[V
    """
    Tìm phân phối dừng của xích Markov bằng phương pháp lặp và xác định thời điểm hội tụ.

    Args:
        P (Matrix): Ma trận chuyển trạng thái.
        pi0 (Vector): Vector phân phối xác suất ban đầu.
        max_steps (int): Số bước lặp tối đa để tìm hội tụ.
        tol (float): Ngưỡng sai số để xác định hội tụ.

    Returns:
        tuple[Vector | None, int | None]:
            - Vector: Vector phân phối dừng nếu tìm thấy, ngược lại là None.
            - int: Thời điểm t mà tại đó hội tụ xảy ra, ngược lại là None.
    """
    pi_t_matrix = pi0.to_matrix() # Bắt đầu từ pi0

    for t in range(1, max_steps + 1):
        pi_next_matrix = P * pi_t_matrix # π_{t+1} = P * π_t

        # Kiểm tra điều kiện hội tụ: |π_next[i] - π_t[i]| < tol cho tất cả các trạng thái i
        converged = True
        for i in range(pi_t_matrix.rows):
            # Chuyển sang float để so sánh (vì tol là float)
            diff = abs(float(pi_next_matrix.matrix[i][0]) - float(pi_t_matrix.matrix[i][0]))
            if diff > tol:
                converged = False
                break

        if converged:
```

```
            # Chuyển kết quả về Vector để trả về
            stationary_vector_data = [float(val[0]) for val in pi_next_matrix.matrix]
            return Vector(stationary_vector_data), t

        pi_t_matrix = pi_next_matrix # Cập nhật pi_t cho bước lặp tiếp theo

    return None, None # Không hội tụ trong giới hạn bước lặp

P = create_transition_matrix_mod7()
pi0 = initial_distribution_vector()

pi_stationary, t_converged = find_stationary_distribution(P, pi0)

if pi_stationary:
    print("\n--- Kiểm tra sự tồn tại và thời điểm hội tụ của phân phối dừng ---")
    print(f"Xích Markov hội tụ tại bước t = {t_converged} với phân phối dừng:")
    pi_stationary.to_matrix().pprint() # In dưới dạng ma trận cột
else:
    print("\nKhông tìm thấy phân phối dừng trong giới hạn bước lặp.")
```

```
--- Kiểm tra sự tồn tại và thời điểm hội tụ của phân phối dừng ---
Xích Markov hội tụ tại bước t = 14 với phân phối dừng:
0.142857
0.142857
0.142857
0.142857
0.142857
0.142857
0.142857
```

---

d) Quá trình tung xúc xắc được diễn ra cho đến khi tồn tại $i \in N^*$ sao cho giá trị S_i chia hết cho 7 thì dừng. Viết hàm tính xác suất tung xúc xắc không quá n lần với giá trị n là một trong những đầu vào của hàm.

Mô hình cho bài toán dừng:

Bài toán này mô tả một chuỗi Markov với **trạng thái hấp thụ** (absorbing state). Trạng thái `0` (tức $S_i \pmod 7 = 0$) là trạng thái hấp thụ vì khi hệ thống đạt đến trạng thái này, quá trình sẽ dừng lại.

Ý tưởng tính toán xác suất dừng (Hitting Probability):

Chúng ta muốn tính xác suất để quá trình dừng (tức đạt đến trạng thái 0) *không quá* n lần tung. Điều này tương đương với việc tính tổng xác suất lần đầu tiên đạt đến trạng thái 0 tại bước 1, hoặc bước 2, ..., hoặc bước n.

Để làm điều này, ta sẽ mô phỏng quá trình từng bước và theo dõi xác suất của các trạng thái. Khi một phần xác suất của chuỗi chuyển sang trạng thái 0, phần đó được "hấp thụ" và được cộng vào tổng xác suất dừng. Phần còn lại của xác suất (ở các trạng thái khác 0) sẽ tiếp tục chuyển động trong chuỗi.

Các bước thực hiện:

1. Khởi tạo `current_pi` là `π_0`.
2. Khởi tạo `total_stopping_probability = 0`.
3. Lặp từ `step = 1` đến `n`:
   - Tính phân phối xác suất `pi_at_step` tại bước hiện tại (`P * current_pi`).
   - Xác suất để dừng *đúng tại bước này* là xác suất ở trạng thái 0 của `pi_at_step`.
   - Cộng xác suất này vào `total_stopping_probability`.
   - Để mô phỏng việc "dừng lại", ta sẽ đặt xác suất ở trạng thái 0 của `pi_at_step` về 0 (để phần xác suất này không tiếp tục chuyển động).
   - Gán `pi_at_step` đã được "hấp thụ" làm `current_pi` cho bước tiếp theo.
4. Trả về `total_stopping_probability`.

In [6]:
```
def calculate_stopping_probability(n: int, P: Ma_Tran, pi0: Vector) -> float:
    """
    Tính xác suất tung xúc xắc không quá n lần mà tổng S_i chia hết cho 7 (quá trình dừng).

    Args:
        n (int): Số lần tung xúc xắc tối đa.
        P (Matrix): Ma trận chuyển trạng thái đầy đủ.
        pi0 (Vector): Vector phân phối xác suất ban đầu.

    Returns:
        float: Xác suất để quá trình dừng trong không quá n lần tung.
    """
    if n <= 0:
        return 0.0
```

```python
    current_pi_matrix = pi0.to_matrix() # Bắt đầu với phân phối ban đầu
    total_stopped_prob = 0.0

    for step in range(1, n + 1):
        # Tính phân phối xác suất ở bước hiện tại
        # Đây là π_n = P * π_{n-1}
        # Sau bước này, current_pi_matrix đại diện cho phân phối tại thời điểm `step`
        current_pi_matrix = P * current_pi_matrix

        # Xác suất để dừng ĐÚNG tại bước này (tức là ở trạng thái 0)
        # và chưa dừng ở các bước trước đó.
        # Ta lấy xác suất ở trạng thái 0 và cộng vào tổng xác suất dừng.
        prob_stopped_at_this_step = float(current_pi_matrix.matrix[0][0])
        total_stopped_prob += prob_stopped_at_this_step

        # Làm cho trạng thái 0 trở thành hấp thụ: đặt xác suất ở trạng thái 0 về 0
        # để phần xác suất này không tiếp tục chuyển động trong các bước sau.
        current_pi_matrix.matrix[0][0] = Fraction(0) # hoặc float(0)

        # Để đảm bảo tổng xác suất của các trạng thái khác vẫn là 1 (trong không gian trạng thái chưa dừng)
        # hoặc để các bước nhân sau vẫn có ý nghĩa trong tổng không điều kiện.
        # Không cần normalize lại ở đây vì chúng ta đang tính tổng xác suất không điều kiện.

    return total_stopped_prob

print("\n--- Xác suất dừng <= n lần (tổng S_n chia hết cho 7) ---")
P_full = create_transition_matrix_mod7()
pi0_full = initial_distribution_vector()

for n_val in range(1, 11):
    prob = calculate_stopping_probability(n_val, P_full, pi0_full)
    print(f"Xác suất dừng <= {n_val} lần: {formatNumber(prob)}")
```

```
--- Xác suất dừng <= n lần (tổng S_n chia hết cho 7) ---
Xác suất dừng <= 1 lần: 0.000000
Xác suất dừng <= 2 lần: 0.166667
Xác suất dừng <= 3 lần: 0.305556
Xác suất dừng <= 4 lần: 0.421296
Xác suất dừng <= 5 lần: 0.517747
Xác suất dừng <= 6 lần: 0.598122
Xác suất dừng <= 7 lần: 0.665102
Xác suất dừng <= 8 lần: 0.720918
Xác suất dừng <= 9 lần: 0.767432
Xác suất dừng <= 10 lần: 0.806193
```