```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Standard Library
import random
from fractions import Fraction
from statistics import stdev
from typing import List, Tuple

FracVec = List[Fraction]
FracMatrix = List[FracVec]


def transition_probs(
        ssp: FracVec,
        trans_probs=[[Fraction(0, 1) for _ in range(1, 10)]
                     for _ in range(1, 10)]
) -> FracMatrix:

    # in Python upper bound is exclusive
    for s1 in range(1, 10):
        for s2 in range(1, 10):
            # indicies are biased
            # state 1, for example, is stored in idx 0
            acc_prob: Fraction = min(Fraction(1, 1),
                                     Fraction(ssp[s2 - 1],
                                              ssp[s1 - 1]))

            sum_of_ps = sum(trans_probs[s1 - 1])

            if sum_of_ps < 1:
                trans_probs[s1 - 1][s1 - 1] = Fraction(1 - sum_of_ps, 1)

            r_num, r_denom = random.random().as_integer_ratio()

            r = Fraction(r_num, r_denom)

            trans_probs[s1 - 1][s2 - 1] = acc_prob * r

    return trans_probs
```

```
def print_matrix(m: FracMatrix) -> None:
    for i in range(len(m)):
        for j in range(len(m[i])):
            if j == 0:
                print('[ ', end='')
            print(m[i][j], end=' ')
        print(']')


def run_markov(trans_probs: FracMatrix, state=random.randrange(1, 10), steps=1) -> int:
    """Proposes a random state given on supplied transition probabilities.
    """

    r_num, r_denom = random.random().as_integer_ratio()
    r = Fraction(r_num, r_denom)

    # associate all probabilities with state number *before*
    # sorting so info about what it is a transition to isn't lost
    ordered_tp: List[Tuple[int, Fraction]] = [(0, Fraction(0, 1))] + \
            sorted(enumerate(trans_probs[state - 1], start=1),
                   reverse=True,
                   key=(lambda pair: pair[1]))

    def between(x, m, n) -> bool: return x > m and x <= n

    for _ in range(steps):
        # tower sampling
        for i in range(len(ordered_tp) - 1):  # -1 because I am looking ahead `i + 1`
            # indicies upper-exclusive so bised
            if between(r, sum([pair[0] for pair in ordered_tp[:i + 1]]),
                       sum([pair[0] for pair in ordered_tp[:i + 2]])):
                state = ordered_tp[i + 1][0]
                break

    return state


def exercise_1() -> FracMatrix:

    # 9 states with equal probability of being in every one
    # so the probability is 1/9 for being in each of them
```

```
    ssp: FracVec = [Fraction(1, 9) for i in range(1, 10)]

    return transition_probs(ssp)


def exercise_2() -> FracMatrix:

    # the sum of all ssp needs to be 1.0
    ssp: FracVec = [

        # s1 [ BOT ROW ] all in mid row need to add up to 1/6
        Fraction(1, 18),
        # s2 [ BOT ROW ] because toprow = 1/18 + 1/18 + 1/18 = 1/18 * 3 = 3/18 = 1/6
        Fraction(1, 18),
        # s3 [ BOT ROW ]
        Fraction(1, 18),

        # s4 [ MID ROW ] all in mid row need to add up to 2/6
        Fraction(2, 18),
        # s5 [ MID ROW ] because midrow = 2/18 + 2/18 + 2/18 = 2/18 * 3 = 6/18 = 2/6
        Fraction(2, 18),
        # s6 [ MID ROW ]
        Fraction(2, 18),

        # s7 distribute remaining probability evenly across the last row ]
        Fraction(1, 6),
        # s8 this gives us (1 - (p(top) + p(bot))) / 3 = 1/6 for every bottom probability
        Fraction(1, 6),
        # s9 p = 1.0 = p(top) + p(bot) + p(bot) = 1/6 + 2/6 + 3/6
        Fraction(1, 6),
    ]

    return transition_probs(ssp)


def exercise_3() -> Tuple[float, Fraction, Fraction, Fraction]:

    results = [run_markov(trans_probs=exercise_2(), steps=3)
               for _ in range(10000)]
    results_s1 = Fraction(
        len([state for state in results if state == 1]),
        len(results))
```

```
    results_s3 = Fraction(
        len([state for state in results if state == 3]),
        len(results))

    results_s9 = Fraction(
        len([state for state in results if state == 9]),
        len(results))

    return stdev(results), results_s1, results_s3, results_s9


def exercise_4() -> Tuple[float, Fraction, Fraction, Fraction]:

    results = [run_markov(trans_probs=exercise_2(), steps=1)
               for _ in range(1000000)]
    results_s1 = Fraction(
        len([state for state in results if state == 1]),
        len(results))

    results_s3 = Fraction(
        len([state for state in results if state == 3]),
        len(results))

    results_s9 = Fraction(
        len([state for state in results if state == 9]),
        len(results))

    return stdev(results), results_s1, results_s3, results_s9


# execution & pprinting
if __name__ == '__main__':
    from sys import argv
    if len(argv) > 1:
        exercise = int(argv[1])
        heading = f'Solution to exercise {exercise}'
        print(heading)
        print('-' * len(heading))
        if exercise == 1:
            print_matrix(exercise_1())
        elif exercise == 2:
```

```
            print_matrix(exercise_2())
        elif exercise == 3:
            std, p1, p3, p9 = exercise_3()
            print(f'probability for state 1: {p1}+-{std}')
            print(f'probability for state 3: {p3}+-{std}')
            print(f'probability for state 9: {p9}+-{std}')
        elif exercise == 4:
            std, p1, p3, p9 = exercise_3()
            print(f'probability for state 1: {p1}+-{std}')
            print(f'probability for state 3: {p3}+-{std}')
            print(f'probability for state 9: {p9}+-{std}')
        else:
            raise NotImplementedError(
                f'exercise number "{exercise}" is invalid, try 1-4')
    else:
        print(f'''
{__file__} - solutions to stochastic systems assessment

Usage:
    {__file__} < 1 | 2 | 3 | 4 >
'''.strip())
```