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

# Standard Library
from fractions import Fraction as Frac
from random import random, randrange
from statistics import stdev
from typing import Any, Callable, List, Tuple, Union

FracVec = List[Frac]
FracMatrix = List[FracVec]


def get_trans_probs(
        SSP: FracVec,
        trans_table=[
            [Frac(1, 2), Frac(1, 4), Frac(0, 1), Frac(1, 4), Frac(
                0, 1), Frac(0, 1), Frac(0, 1), Frac(0, 1), Frac(0, 1)],
            [Frac(1, 4), Frac(1, 4), Frac(1, 4), Frac(0, 1), Frac(
                1, 4), Frac(0, 1), Frac(0, 1), Frac(0, 1), Frac(0, 1)],
            [Frac(0, 1), Frac(1, 4), Frac(1, 2), Frac(0, 1), Frac(
                0, 1), Frac(1, 4), Frac(0, 1), Frac(0, 1), Frac(0, 1)],
            [Frac(1, 4), Frac(0, 1), Frac(0, 1), Frac(1, 4), Frac(
                1, 4), Frac(0, 1), Frac(1, 4), Frac(0, 1), Frac(0, 1)],
            [Frac(0, 1), Frac(1, 4), Frac(0, 1), Frac(1, 4), Frac(
                0, 1), Frac(1, 4), Frac(0, 1), Frac(1, 4), Frac(0, 1)],
            [Frac(0, 1), Frac(0, 1), Frac(1, 4), Frac(0, 1), Frac(
                1, 4), Frac(1, 4), Frac(0, 1), Frac(0, 1), Frac(1, 4)],
            [Frac(0, 1), Frac(0, 1), Frac(0, 1), Frac(1, 4), Frac(
                0, 1), Frac(0, 1), Frac(1, 2), Frac(1, 4), Frac(0, 1)],
            [Frac(0, 1), Frac(0, 1), Frac(0, 1), Frac(1, 4), Frac(
                1, 4), Frac(0, 1), Frac(1, 4), Frac(1, 4), Frac(1, 4)],
            [Frac(0, 1), Frac(0, 1), Frac(0, 1), Frac(0, 1), Frac(
                0, 1), Frac(1, 4), Frac(0, 1), Frac(1, 4), Frac(1, 2)],
        ]) -> FracMatrix:

    test_square('funct get_trans_probs', trans_table)

    assert sum(SSP) == Frac(1, 1) or \
        sum(SSP) == 1, fail_msg(
        "funct get_trans_probs",
```

```python
        "sum of SSP not equal to 1",
        1.0,
        f"p = sum({', '.join(map(str, SSP))}) = {float(sum(SSP))}")

    # in Python upper bound is exclusive
    # for each pair of states
    for s1 in range(1, 10):
        for s2 in range(1, 10):
            # indices are biased
            # state 1, for example, is stored in index 0
            acc_prob: Frac = min(Frac(1, 1),
                                 Frac(SSP[s2 - 1],
                                      SSP[s1 - 1]))

            r: Frac = trans_table[s1 - 1][s2 - 1]
            #  r: Frac = Frac(*random().as_integer_ratio())
            trans_table[s1 - 1][s2 - 1] = acc_prob * r

        non_self_ps = Frac(0, 1)
        for s3 in range(1, 10):
            if s3 != s1:
                non_self_ps += trans_table[s1 - 1][s3 - 1]

        if non_self_ps < Frac(1, 1):
            trans_table[s1 - 1][s1 - 1] = Frac(1, 1) - non_self_ps

    return trans_table


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

    r: Frac = Frac(*random().as_integer_ratio())

    # 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, Frac]] = [(0, Frac(0, 1))] + \
        sorted(enumerate(trans_table[state - 1], start=1),
               reverse=True,
```

```python
            key=(lambda pair: pair[1]))

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

    for _ in range(steps):
        # tower sampling
        # -1 because I am looking ahead `i + 1`
        for i in range(len(ordered_tp) - 1):
            # indices upper-exclusive so ranges are biased
            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 fail_msg(where: str, issue: str, expected: Any, got: Any) -> str:
    return f"[FAIL] [{where}] {issue}, EXPECTED: {expected}, GOT: {got}"


def test_square(where: str, m: FracMatrix) -> None:
    assert len(m) == len(m[0]), \
        fail_msg(where,
                 'not an NxN matrix',
                 'square matrix',
                 m)


def print_matrix(m: FracMatrix) -> None:
    for i in range(len(m)):
        print('[ ', end='')
        for j in range(len(m[i])):
            print("%2.2f" % float(m[i][j]), end=' ')
        print(']')


def exercise_1() -> Tuple[FracVec, 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 = [Frac(1, 9) for i in range(1, 10)]
```

```python
    return SSP, get_trans_probs(SSP)


def exercise_2() -> Tuple[FracVec, FracMatrix]:

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

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

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

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

    return SSP, get_trans_probs(SSP)


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

    t = 10**4

    _, trans_table = exercise_2()
```

```python
    results = [run_markov(trans_table=trans_table, steps=3)
               for _ in range(t)]

    # count state in the pool of all s
    def prob_of_s(state: int) -> Frac:
        return Frac(
            len([s for s in results if s == state]),
            len(results))

    return stdev(results), prob_of_s(1), prob_of_s(3), prob_of_s(9)


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

    t = 10**6

    _, trans_table = exercise_2()

    results = [run_markov(trans_table=trans_table, steps=1)
               for _ in range(t)]

    # count state in the pool of all s
    def prob_of_s(state: int) -> Frac:
        return Frac(
            len([s for s in results if s == state]),
            len(results))

    return stdev(results), prob_of_s(1), prob_of_s(3), prob_of_s(9)


def test_tranition_probs(trans_table: FracMatrix, exe_no: int) -> None:
    for state in range(len(trans_table)):
        total_p: Union[Frac, int] = sum(trans_table[state])
        assert (total_p == Frac(1, 1)) or (total_p == 1), \
            fail_msg(f"exercise {exe_no}",
                     f"trans probs from state {state + 1} didn't add up to 1.0",
                     1.0,
                     f"p = sum({', '.join(['%2.2f' % float(i) for i in trans_table[state]])}) = {'%2.2f' % float(total_p)}")

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

```python
    for row in range(len(trans_table)):
        for col in range(len(trans_table[row])):
            assert between(
                trans_table[row][col],
                Frac(0, 1),
                Frac(1, 1)), fail_msg(
                f"exercise {exe_no}",
                f'p({row + 1} -> {col + 1}) no in range [0, 1]',
                'p in range [0, 1]',
                '%2.2f' % float(trans_table[row][col]))


def test_exercise_1() -> None:
    SSP, trans_table = exercise_1()
    test_square('test_exercise_1', trans_table)
    test_tranition_probs(trans_table, 1)


def test_exercise_2() -> None:
    SSP, trans_table = exercise_1()
    test_square('test_exercise_2', trans_table)
    test_tranition_probs(trans_table, 2)


def test_exercise_3() -> None:
    std, p1, p3, p9 = exercise_3()

    def valid_p(p: Frac) -> bool:
        return p >= Frac(0, 1) and p <= Frac(1, 1)

    def check(n: int, p: Frac) -> None:
        assert valid_p(p), fail_msg('exercise 3',
                                    f'invalid SSP for state {n}',
                                    'valid SSP in range [0, 1]',
                                    '%2.2f' % float(p))

    check(1, p1)
    check(3, p3)
    check(9, p9)

    assert (p1 + p3 + p9) <= Frac(1, 1), \
        fail_msg('exercise 3',
```

```python
            'invalid SSP for states 1, 3, 9'
            'sum(p(1), p(3), p(9)) < 1.0',
            '%2.2f' % float(p1 + p3 + p9))


def test_exercise_4() -> None:
    std, p1, p3, p9 = exercise_4()

    def valid_p(p: Frac) -> bool:
        return p >= Frac(0, 1) and p <= Frac(1, 1)

    def check(n: int, p: Frac) -> None:
        assert valid_p(p), fail_msg("exercise 4",
                              f"invalid SSP for state {n}",
                              f"p({n}) in range [0, 1]",
                              p)

    assert (p1 + p3 + p9) <= Frac(1, 1), \
        fail_msg("exercise 4",
                 "invalid SSP for states 1, 3, 9",
                 "sum(p(1), p(3), p(9)) < 1.0",
                 p1 + p3 + p9)

    check(1, p1)
    check(3, p3)
    check(9, p9)


# execution & pprinting
if __name__ == '__main__':

    from argparse import ArgumentParser, Namespace
    from os.path import basename

    parser: ArgumentParser = ArgumentParser(
        prog=basename(__file__).replace('.py', ''),
        description='solutions to stochastic systems assessment')

    parser.add_argument(
        'exercise',
        help='the exercise number',
        choices=[1, 2, 3, 4],
```

```python
    type=int)

parser.add_argument(
    '--test',
    help='run tests for the exercise instead of running it',
    action='store_true',
    default=False)

args: Namespace = parser.parse_args()

if args.exercise < 0 or args.exercise > 4:
    raise NotImplementedError(
        f'exercise number "{args.exercise}" is invalid, try 1-4')

def print_heading(heading: str) -> None:
    print(heading, '\n', '-' * len(heading))

if args.test:
    print_heading(f'tests for exercise {args.exercise}')
    if args.exercise == 1:
        test_exercise_1()
    elif args.exercise == 2:
        test_exercise_2()
    elif args.exercise == 3:
        test_exercise_3()
    elif args.exercise == 4:
        test_exercise_4()
else:
    print_heading(f'Solution to exercise {args.exercise}')
    if args.exercise == 1:
        SSP, trans_table = exercise_1()
        print_matrix(trans_table)
    elif args.exercise == 2:
        SSP, trans_table = exercise_2()
        print_matrix(trans_table)
    elif args.exercise == 3 or args.exercise == 4:

        std, p1, p3, p9 = exercise_3() if args.exercise == 3 \
            else exercise_4()

        def show(n: int, p: Frac) -> None:
            print(f'probability for state {n}: {p}+-{std}')
```

```
show(1, p1)
show(3, p3)
show(9, p9)
```