```python
# Standard Library
from fractions import Fraction as Frac
from random import random, randrange
from statistics import stdev
from typing import List, Tuple, Dict

# Relative
from tests import  test_is_square, test_probs_eq_1
from utils import ONE, ZERO, FracVec, FracMatrix

# ADJECANCY_LIST[state - 1] == neighbours for state
ADJECANCY_LIST: List[List[int]] = [
    [2, 4],
    [1, 3, 5],
    [2, 6],
    [1, 5, 7],
    [2, 4, 6, 8],
    [3, 5, 9],
    [4, 8],
    [5, 7, 9],
    [6, 8],
]

def random_idx(xs: List[int]):
    return randrange(0, len(xs))

def propse_state(current_state: int) -> int:
    return random_idx(ADJECANCY_LIST[current_state])

def get_trans_probs(SSP: FracVec) -> FracMatrix:

    trans_table: FracMatrix = [[ZERO for _ in range(1, 10)] for _ in range(1, 10)]

    test_is_square('funct get_trans_probs', trans_table)
    test_probs_eq_1("funct get_trans_probs", SSP)

    # in Python upper bound is exclusive
    # for each pair of states
    for s1 in range(1, 10):
        # indices are biased
        # state 1, for example, is stored in index 0
        for s2 in ADJECANCY_LIST[s1 - 1]:
            acc_prob: Frac = min(ONE,
                                 Frac(SSP[s2 - 1],
                                      SSP[s1 - 1]))
            # r: Frac = Frac(*random().as_integer_ratio())
            prop_prob: Frac = Frac(1, len(ADJECANCY_LIST[s1 - 1]))
            trans_table[s1 - 1][s2 - 1] = acc_prob * prop_prob

        non_self_ps = ZERO
        for neighbour in set(range(1, 10)) - {s1}:
            non_self_ps += trans_table[s1 - 1][neighbour - 1]

        if non_self_ps < ONE:
            print(f"non-self transitions don't add up to 1 (got {non_self_ps}), adding self-
transition")
            trans_table[s1 - 1][s1 - 1] = ONE - 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, ZERO)] + \
        sorted(enumerate(trans_table[state - 1], start=1),
               reverse=True,
               key=(lambda pair: pair[1]))

    def between_upper_inc(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_upper_inc(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() -> 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)]

    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 bot row need to add up to 1/6
        # because top_row = 1/18 + 1/18 + 1/18 = 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 = 6/18 = 2/6
        Frac(2, 18),  # s4
        Frac(2, 18),  # s5
        Frac(2, 18),  # s6

        # [ TOP ROW ]
        # distribute remaining probability evenly across the top row
        #
        #   (1 - (p(top) + p(bot))) / 3 = 1/6
        #
        # 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()

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

    # count state in the pool of all s
    def p(state: int) -> Frac:
```

```
        return Frac(len([s for s in results if s == state]),
                    len(results))

    return stdev(results), p(1), p(3), p(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 p(state: int) -> Frac:
        return Frac(
            len([s for s in results if s == state]),
            len(results))

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


# execution & pprinting (command line interface) defined in cli.py
```