```python
# coding: utf-8

# In[1]:

from numpy.random import rand
from numpy import mean, roots
import numba
import numpy as np
from scipy.linalg import pascal
get_ipython().magic('load_ext autotime')


# In[2]:

def probability_deterministic(win_prob, games=82):
    """
    The goal is to find the probabilitiy of a team making it through all games
        without consecutive losses.
    In order to succeed, a team will always be on either a 0 or 1 game losing
        streak.
    The approach uses dynamic programming to take advantage of previous
        calculations.
    To start, you calculate the probabilities of ending the second game with a
        0 and 1 game losing streak.
    From these calculations, you can use the following formula to iterate
        through any number of games:
        format: prob(length of losing streak, number of games completed)
        prob(0, n) = [prob(0, n-1) + prob(1, n-1)] * win_prob
        prob(1, n) = prob(0, n-1) * (1-win_prob)
    The final answer is the addition of prob(0, n) and prob(1, n).
    """
    num_columns = games-1
    matrix = np.zeros((2,num_columns))
    matrix[0][0] = win_prob * win_prob + (1-win_prob) * win_prob
    matrix[1][0] = win_prob * (1-win_prob)
    for i in range(1, num_columns):
        prob_0_win = matrix[0][i-1]
        prob_1_win = matrix[1][i-1]
        matrix[0][i] = win_prob * (prob_0_win + prob_1_win)
        matrix[1][i] = (1-win_prob) * prob_0_win
    return matrix[0][num_columns-1] + matrix[1][num_columns-1]


# In[3]:

def probability_pascal(win_prob, games=82):
    """
    This method takes advantage of pascal's triangle, finding all the possible
        ways to lose consecutive games.
    Beginning 3 games into the season, create this table:
        The rows represent how many games can be lost before the 2 game losing
            streak (no consecutive losses prior)
        The columns represent the number of games played thus far
        The cells are the number of occurences among each row/column
            combination
```

```
    This table is a modified upper pascal triangle, with slight adjustments to
        the preceeding zeros in each row.
    Since pascal's traingle can be easily computed, the table can be filled out
        for an arbitrary number of games.
    Each cell represents a number of combinations for a single probability,
        calculted with this formula:
        prob = win_prob**(2 + column value - row value)*(1 - win_prob)**(row
            value + 2)
    Multiply each probability by the value of the cell and sum all cells.
    Add this number to the probability of two consecutive losses after 3 games.
    """
    num_rows = int(games/2)
    num_columns = games-3
    triangle = pascal(num_columns, kind='upper')
    matrix = np.zeros((num_rows,num_columns))
    for row in range(num_rows):
        if row == 0:
            matrix[row] = list(triangle[row])
        elif row == 1:
            row_list = list(triangle[row])
            del row_list[0]
            row_list.append(row_list[-1]+1)
            matrix[row] = row_list
        else:
            row_list = list(triangle[row])
            for _ in range(row-2):
                row_list.insert(0,0)
                del row_list[-1]
            matrix[row] = row_list
    probs = (1-win_prob)**2 + win_prob*(1-win_prob)**2
    for row in range(num_rows):
        for column in range(num_columns):
            combos = matrix[row][column]
            probs += combos*(win_prob)**(2+column-row)*(1-win_prob)**(row+2)
    return 1-probs


# In[4]:

def probability_approximation(win_prob, games=82, streak=2):
    """
    This method uses distributions to derive the probability of interest.
    After deriving the probability generating function, the distribution is
        slightly skewed.
    Partial fraction expansion is then used to approximate the tail of the
        distribution.
    This leads to the formula:
        p: win probability
        q: loss probability
        r: length of losing streak
        n: number of games
        prob ~ (1 - p*x)/((r + 1 - r*x)*q) * 1/(x**(n+1))
        where x is the positive root, not equal to 1/p in the following
            equation:
            1 - x + q*(p**r)*x**(r+1) = 0
    """
```

```
    loss_prob = 1 - win_prob
    x_vals = roots(np.append(np.append([win_prob*loss_prob**streak], np.repeat(0,streak-1)), [-1, 1]))
    for val in x_vals:
        if val != 1/loss_prob and val > 0:
            x = val
    return (1 - loss_prob*x)/((streak + 1 - streak*x)*win_prob) * 1/(x**(games+1))


# In[5]:

@numba.jit(nopython=True, cache=False, nogil=True)
def probability_simulation(win_prob, simulations=1000000, games=82):
    """
    This method uses repeated simulation to approximate the probability of interest
    For each simulated game, a random number from 0 to 1 is generated.
    If the number is less than the losing probability, the game is deemed a loss.
    This process is repeated for all 82 games or unless there are two consecutive losses.
    At the end of the simulation, it is recorded whether or not there was a string of consecutive losses.
    This is then repeated for n seasons and the average number of seasons with consecutive losses is calculated.
    """
    two_losses = 0
    for _ in range(simulations):
        lost_previous = False
        for _ in range(games):
            if rand() <= 1-win_prob:
                if lost_previous:
                    two_losses += 1
                    break
                lost_previous = True
            else:
                lost_previous = False
    return 1 - two_losses/simulations


# In[6]:

def win_prob_to_prevent_losses(win_prob=.8, increment=.0001, outcome_prob=.5,
    method=probability_deterministic):
    prob = method(win_prob)
    while prob < outcome_prob:
        win_prob += increment
        prob = method(win_prob)
    return win_prob
```