Wrong Winner in Tennis

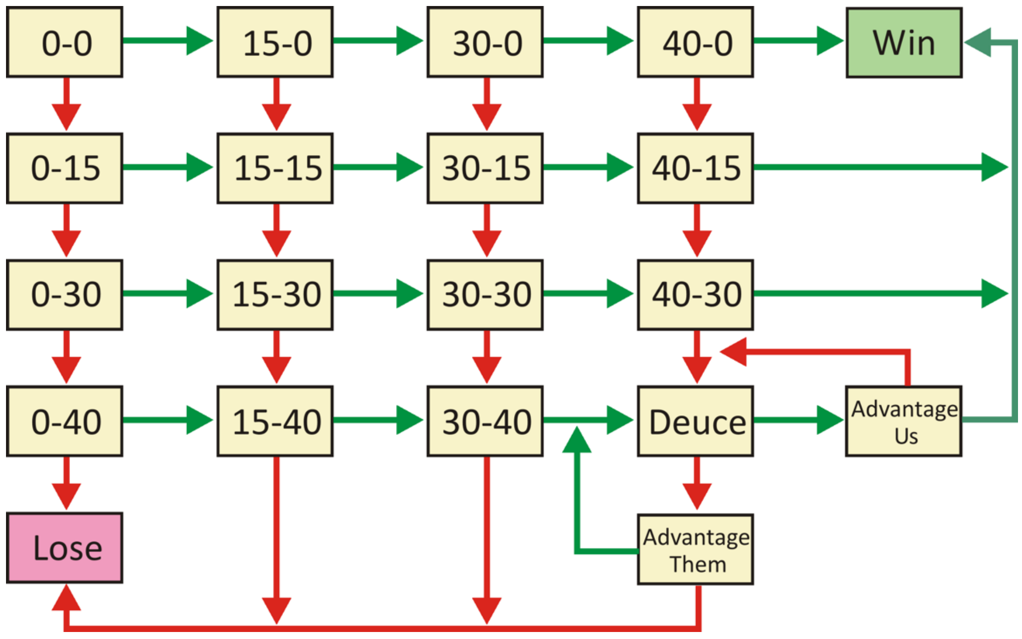
This week's [Fiddler on the Proof](#) (12 July 2024), Extra Credit asks:

> Assume a three-set tennis match features two players who are evenly matched, so that each player has a 50 percent chance of winning any given point. Also, points are independent, so the outcome of one point doesn't affect the probability of who wins subsequent points.
>
> How likely is it that one of the players *loses* a majority of the points in a match while *winning* the match itself? (Here, I mean a *strict* majority, i.e., more than 50 percent of the points.)

States of a Match

If you consider all the possible scores that can happen during a match, there are 2,944 possible "states" that a match can be in. There is one **initial** state (0-0 in sets, games, and points). And there are four **terminal** states (us winning 2-0, us winning 2-1, us losing 2-0, and us losing 2-1). Each state (except for the four terminal states) leads to two other states, which we arrive at with equal likelihood.

To illustrate, here are the states of just one *game* of tennis. The red arrows are points we lose; the green arrows are points we win.



What I would like to do is treat the states of the match as a large [Markov chain](#), but there is a problem. To be able to solve the Markov chain, I need to know the probabilities associated with the terminal states of the match. But I don't (yet) have enough information to do that because I don't know which side won more points.

Accounting for Points Won

I could include point totals (number of points won by each side) in the states of the match, but that leads to another problem. Now the number of possible states is unlimited. That's not very convenient.

But there is another way. Since all I need to know is which side won more points, I can instead keep track of the *point difference* between the players. There is a limit to how big that point difference can be. It is impossible for one player to ever be more than 58 points ahead of the other. Since the point difference is finite, the number of different states is also finite.

Including the point difference in the states of the match increases the number of terminal states from 4 to 344 and increases the number of total states from 2,944 to 137,228. That is a big number, but still managable.

Calculating the Probability

The puzzle asks us how likely it is that the "wrong" player wins the match. (By "wrong," I mean that the player who scored fewer total points wins the match anyway.)

This is a straightforward calculatation for the 344 terminal states. Simply look at the point difference. If the match winner has a negative lead in total points, the probability associated with that state is 1.0. If the match winner has a positive (or no) lead, the probability is 0.0.

My favorite way to solve a Markov chain is by "progressive refinement." Initially, you assign an arbitrary probability (such as 0) to every state other than the terminal states. Then progressively recalculate each state's probability. Do this over and over until the values stop changing.

Recalculating a state's probability is easy. There are two possible future states: the state resulting after we win a point, and the state resulting after we lose a point. The probability of the current state is the average the probabilities of the two future states.

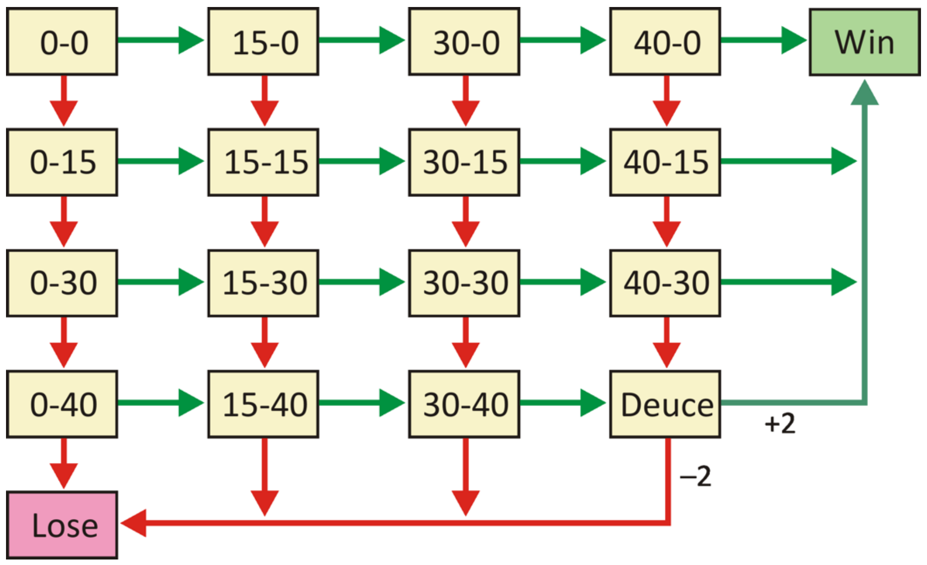
Progressive recalculation causes the known (correct) values of the terminal states to percolate back to the earlier states of the match. When the values stop changing, the probabilities of all the states of the match are consistent with one another.

Now look at the probability associated with the initial state of the match. The number I found was 0.0626723364164183 and (if I didn't make any coding errors) that is the answer to the puzzle.

Short-Circuiting Deuce

After thinking about it some more, it occurs to me that an easier way to calculate the probability of a wrong winner revolves around how we handle deuce scores. "Deuce" in tennis happens when both players are 1 point short of winning the game. Because of the "win by 2" rule, when you reach deuce the winner of the game must win 2 points in a row.

The problem with deuce is that the score can cycle back on itself and cause the infinite recursion. We can avoid this problem by noting that when we reach deuce, the chance of winning the game is the same as the chance of losing it (because the players are equally strong).



Once we reach deuce, the game can end in only two ways: We win (adding two points to the running point difference) or we lose (subtracting two points from the running point difference). Calculating deuce directly like this takes the cycles out of the state diagram and allows us to calculate the probability of the game recursively and, ultimately, the match.

We can further speed things up by using memoization to avoid repeated calculations on the same state.

Performing a recursive calculation, and treating the probability as a rational number, we get

$$\frac{3663828891517944390502183649313847184357853598 29}{5846006549323611672814739330865132078623730171904}$$

which equals:

0.06267 23364 16418 36269 23427 71252 23010 18274 03816 45567
94888 90204 46058 68698 51757 13574 48959 01977 44584 52620
63757 38098 21510 99055 48408 91952 89243 43467 22438 93146
51489 25781 25

Here is the source code that implements the short-circuiting method. The code is simpler and faster than the Markov chain method described above, but it is not as general.

```
// The possible players: "us", "them", or "none"
enum Player { case us, them, none }

// The score in a game, set, or match
struct Score : Hashable
   {
   var us = 0, them = 0
   static let zero = Score()

   // Add a point to one side of a score
   mutating func addPoint( player:Player )
      {
      switch player
         {
      case .us:   self.us += 1
      case .them: self.them += 1
      case .none: fatalError()
         }
```

```
        }
    }

// The state of a tennis match
struct State : Hashable
    {
    var matchWinner = Player.none
    var matchScore = Score.zero
    var setScore   = Score.zero
    var gameScore  = Score.zero
    var pointDiff  = 0

    static let startOfMatch = State()

    // Length of current game, in points
    var gameLength : Int
        {
        let isTiebreak = self.setScore == Score( us:6, them:6 )
        return isTiebreak ? 7 : 4
        }

    // State that results after a player scores one point
    func afterPoint( for player:Player ) -> State
        {
        var state = self

        // Add to player's game score
        state.gameScore.addPoint( player:player )

        // Adjust running point difference
        state.pointDiff += player == .us ? 1 : -1

        // If game is over, clear the game score and add to the set score
        let isGameOver = state.gameScore.us >= gameLength
                      || state.gameScore.them >= gameLength
        guard isGameOver else { return state }
        state.gameScore = .zero ; state.setScore.addPoint( player:player )

        // If set is over, clear the set score and add to the match score
        let isSetOver = state.setScore.us >= 7
                     || state.setScore.us == 6 && state.setScore.them <= 4
                     || state.setScore.them >= 7
                     || state.setScore.them == 6 && state.setScore.us <= 4
        guard isSetOver else { return state }
        state.setScore = .zero ; state.matchScore.addPoint( player:player )

        // If match is over, clear the match score and set the match winner
        let isMatchOver = state.matchScore.us >= 2 || state.matchScore.them >= 2
        guard isMatchOver else { return state }
        state.matchScore = .zero ; state.matchWinner = player

        return state
        }

    typealias Result = Rational<BigUInt>

    // Evaluate the probability that the wrong player will win the match
    func rawEval() -> Result
        {
        // Terminal state
        switch self.matchWinner
            {
            case .us:   return self.pointDiff < 0 ? Result(1) : Result(0)
            case .them: return self.pointDiff > 0 ? Result(1) : Result(0)
            case .none: break
            }

        // Regular point
        var winPoint = self.afterPoint( for:.us )
        var losePoint = self.afterPoint( for:.them )

        // Adjustment for deuce
        let deuce = Score( us:self.gameLength-1, them:self.gameLength-1 )
        if self.gameScore == deuce
            {
            winPoint.pointDiff += 1 ; losePoint.pointDiff -= 1
            }

        return ( winPoint.eval() + losePoint.eval() ) / 2
        }

    // Memoized version of rawEval().
    static var evalMemo = Dictionary< State, Result >()
    func eval() -> Result
        {
        if let answer = Self.evalMemo[ self ] { return answer }
        let answer = self.rawEval()
        Self.evalMemo[ self ] = answer
        return answer
        }
    }

let probability = State.startOfMatch.eval()
print( probability )
print()
print( probability.decimalExpansion( digitsK:162 ) )
```

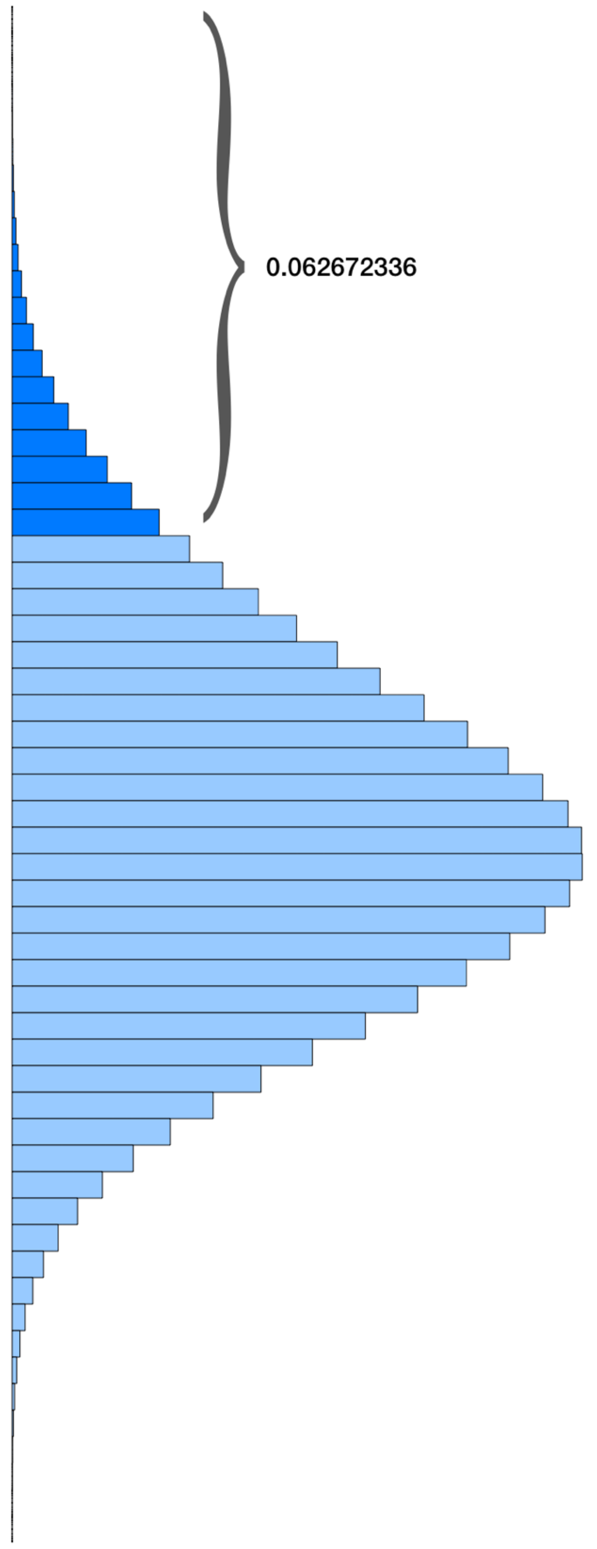
```
366382889151794439050218364931384718435785359829/5846006549323611672814739330865132078623730171904

0.06267233641641836269234277125223010182740381645567948889020446058686985175713574489590197744584526206375738098
2151099055484089195289243434672243893146514892578125
```

Probability Distribution

We can use largely the same code to calculate the exact distribution of probabilities based on the ultimate lead in points for the winner at the end of the match. Here are the results. If you add up the probabilities of the games where the winner ended with a negative point lead, you get the same answer we calculated earlier.

Winner's Point Lead	Probability
−20	0.0000021412
−19	0.0000046672
−18	0.0000097713
−17	0.0000196698
−16	0.0000381076
−15	0.0000711166
−14	0.0001279505
−13	0.0002221178
−12	0.0003723557
−11	0.0006033176
−10	0.0009457016
−9	0.0014355662
−8	0.0021126677
−7	0.0030178144
−6	0.0041894310
−5	0.0056597379
−4	0.0074511585
−3	0.0095737389
−2	0.0120244662
−1	0.0147892425
0	0.0178477444
+1	0.0211801902
+2	0.0247733233
+3	0.0286209977
+4	0.0327145244
+5	0.0370203080
+6	0.0414496889
+7	0.0458330046
+8	0.0499158264
+9	0.0533862191
+10	0.0559319069
+11	0.0573028729
+12	0.0573615737
+13	0.0560954927
+14	0.0536061547
+15	0.0500713250
+16	0.0457180612
+17	0.0407903844
+18	0.0355423212
+19	0.0302181615
+20	0.0250496126
+21	0.0202316373
+22	0.0159150256
+23	0.0121887654
+24	0.0090851986
+25	0.0065843984
+26	0.0046327357
+27	0.0031561948
+28	0.0020748662
+29	0.0013102841
+30	0.0007908329
+31	0.0004536438
+32	0.0002459282
+33	0.0001252948
+34	0.0000596793
+35	0.0000264458
+36	0.0000108553
+37	0.0000041112


0.062672336