Dynamic Programing – Presentation 1

1) The problem:

You and a friend are playing a game using an n-sided dice labeled from 1 to n. You first throw the dice and record the number showing up. Then your friend throws the dice. If his toss is larger than yours, than your friend wins. If his toss is the same, then he toss again. Otherwise, it is your turn to toss the dice. You win if your toss is larger than your friend's toss. Otherwise, you keep tossing if it's the same, and lose your turn if it's smaller. The game continues until one of you win. Given n, return the probability that you win.

Some things to notice:

1: You cannot lose against a roll of 1 (since you re-roll ties).

2: You cannot beat a roll of n.

3: The game necessarily terminates after at most n-1 rounds. (Since a loss implies that the new number to beat is smaller than the previous number).

2) Simulation results:

Because my probability skills are a little rusty, I wrote a quick little simulator to estimate what the probabilities should look like:

```
private boolean playGame(int n)
{
	Random random = new Random();
	int toBeat = random.nextInt(n)+1;
	int turn = 0;
	while(true)
	{
		int roll = random.nextInt(n)+1;
		while(roll == toBeat)
			roll =  random.nextInt(n)+1;
		if (roll > toBeat)
			return turn%2 != 0;
		toBeat = roll;
		turn ++;
	}
}
```

20000000 games with n from 2 to 20:

N	Wins	Percentage
2	999233	0.4996165
3	833520	0.41676
4	796221	0.3981105
5	779738	0.389869
6	769729	0.3848645
7	764494	0.382247
8	761285	0.3806425
9	758771	0.3793855
10	753560	0.37678
11	753584	0.376792
12	750990	0.375495
13	748517	0.3742585

14	747935	0.3739675
15	746843	0.3734215
16	746375	0.3731875
17	745597	0.3727985
18	746141	0.3730705
19	745482	0.372741
20	744136	0.372068

3) Solution:

A. Recursive DP Solution

```
private static double[] memo;

private static double recursiveSolutionPublic{
	memo = new double[n];
	for (int i = 0; i < n;i++)
		memo[i] = -1;

	double probWin = 0;

	for (int i = 1; i<=n;i++)
	{
		probWin += 1.0/((double)n) * recursiveSolution(n,i);
	}
	return probWin;
}


/**
 * Recursive solution
 * @param n
 * @param toBeat
 * @return
 */
public static final double recursiveSolution(int n, int toBeat)
{
	if (memo[toBeat-1]>-1)
		return memo[toBeat-1];

	double probWin =(n-toBeat )/(double)(n - 1);//subtract one because the
result toBeat is not allowed to occur
	for (int i = 1; i < toBeat;i++)
	{
		probWin += 1.0/((double)(n-1)) * (1.0- probabilityDP(n,i));
	}
	memo[toBeat-1] = probWin;
	return probWin;
}
```

B. Iterative DP Solution

```
public static final double probabilityDPIterative(int n)
{
	double[] memo = new double[n];
	for (int i = 0; i < n;i++)
		memo[i] = -1;

	memo[0] = 1.0;

	for ( int toBeat = 2;toBeat<= n;toBeat++)
	{
		double probWin =(n-toBeat )/(double)(n - 1);//subtract one
because the result toBeat is not allowed to occur
		for (int i = 1; i < toBeat;i++)
		{
			probWin += 1.0/((double)(n-1)) * (1.0- memo[i-1]);
		}
		memo[toBeat-1] = probWin;
	}

	double probWin = 0;

	for (int i = 1; i<=n;i++)
	{
		probWin += 1.0/((double)n) * memo[i-1];
	}
	return probWin;
}
```

Note: With a little bit of cleverness, you could almost certainly re-arrange this to eliminate the inner loop and reduce the memo array to just one, or perhaps two, numbers.