Assignment 4 Code:

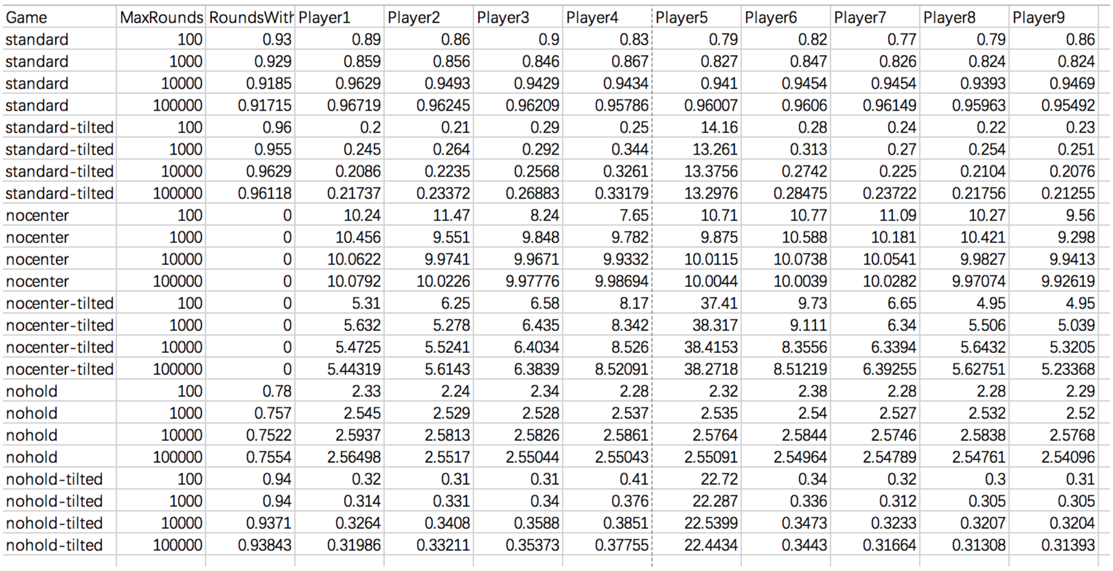
```
#include <fstream>
#include <iostream>
#include <random>
#include <ctime>
using namespace std;

class Player
{
private:
    const int * chipDist;
    int numChips;
    Player* leftPlayer;
    Player* rightPlayer;
public:
    Player(){}
    Player(const int* chipDistribution);
    ~Player(){}
    void setLeftPlayer(Player* p);
    Player* getLeftPlayer();
    void setRightPlayer(Player* p);
    Player* getRightPlayer();
    int getNumChips();
    void setNumChips(int nc);
    int addChips(int nc);
    int play(int d1, int d2, int d3);
    int eachPlay(int d);

};

// Part 1 - Creating Classes
// A.
// a.
Player::Player(const int* chipDistribution)
{
    chipDist = chipDistribution;
}

// b.
void Player::setLeftPlayer(Player *p)
{
    leftPlayer = p;
}

Player* Player::getLeftPlayer()
{
    return leftPlayer;
}

```

```
// c.
void Player::setRightPlayer(Player *p)
{
    rightPlayer = p;
}

Player* Player::getRightPlayer()
{
    return rightPlayer;
}

// d.
int Player::getNumChips()
{
    return numChips;
}

void Player::setNumChips(int nc)
{
    numChips = nc;
}

// e.
int Player::addChips(int nc)
{
    setNumChips(getNumChips() + nc);
    return getNumChips();
}

// f.
int Player::play(int d1, int d2, int d3)
{

    int num = getNumChips();
    int centerChips = 0;
    if(num >= 1)
        centerChips += eachPlay(d1);
    if(num >= 2)
        centerChips += eachPlay(d2);
    if(num >= 3)
        centerChips += eachPlay(d3);

    return centerChips;
}

int Player::eachPlay(int d)
{
    int centerChips = 0;
    if(chipDist[d] == 1)
    {
        this->addChips(-1);
        leftPlayer->addChips(1);
        //cout<<"Move chip to left"<<endl;
    }
    else if(chipDist[d] == 2)
    {
        this->addChips(-1);
        centerChips++;
        //cout<<"Move chip to center"<<endl;
    }
    else if(chipDist[d] == 3)
    {
        this->addChips(-1);
        rightPlayer->addChips(1);
        //cout<<"Move chip to right"<<endl;
    }
    return centerChips;
}


// B.
// a.
class Game
{
private:
    int numP;
    Player** players;
    long seed;
    int* chip_count;
    mt19937 mt_rand;

public:
    Game(int numPlayers, const int* chipDistribution, long seed);
    virtual ~Game();
    int countPlayersWithChips();
    int playRound(int startingPlayer);
    const int* playGame(const int* startingChips, int maxRounds);
};

Game::Game(int numPlayers, const int* chipDistribution, long seed)
{
    numP = numPlayers;
    this->seed = seed;
    players = new Player* [numP];
    chip_count = new int[numP];

    // set Chip Distribution for each player
    for(int i = 0; i < numPlayers; i++)
    {
        players[i] = new Player(chipDistribution);
    }

    // set left player and right player for each player
    for(int i = 0; i < numPlayers; i++)
    {
        int leftIndex = (i + 1 + numPlayers) % numPlayers;
        int rightIndex = (i - 1 + numPlayers) % numPlayers;

        players[i]->setLeftPlayer(players[leftIndex]);
        players[i]->setRightPlayer(players[rightIndex]);

    }
    mt19937 mt_rand(seed);
}


// b.
Game::~Game()
{

    for(int i = 0; i < numP; i++)
    {
        delete players[i];
    }

    delete [] players;
    delete [] chip_count;
}

// c.
int Game::countPlayersWithChips()
{
    int cnt = 0;
    for(int i = 0; i < numP; i++)
    {
        if(players[i]->getNumChips() > 0)
            cnt++;
    }
    return cnt;
}

// d.
int Game::playRound(int startingPlayer)
{
    int cnt = 0;
    uniform_int_distribution<int> dis_unif(0, 5);
    for(int i = 0; i < numP; i++)
    {
        int index = (i + startingPlayer) % numP;

        int d1 = dis_unif(mt_rand);
        int d2 = dis_unif(mt_rand);
        int d3 = dis_unif(mt_rand);
        cnt += players[index]->play(d1, d2, d3);
    }
    return cnt;
}

const int* Game::playGame(const int *startingChips, int maxRounds)
```

```
{

    uniform_int_distribution<int> dis_unif(0, numP);
    int startPlayer = dis_unif(mt_rand);
    for(int i = 0; i < numP; i++)
    {
        players[i]->setNumChips(startingChips[i]);
        chip_count[i] = startingChips[i];
    }

    int chip_center = 0;

    for(int i = 0; i < maxRounds; i++)
    {
        chip_center += playRound(startPlayer);
        if(countPlayersWithChips() == 1)
        {
            break;
        }
    }


    // when the game is concluded
    if(countPlayersWithChips() == 1)
    {
        for(int i = 0; i < numP; i++)
        {
            chip_count[i] = players[i]->getNumChips();
            if(players[i] -> getNumChips() > 0)
                players[i] -> addChips(chip_center);
        }
    }

    return chip_count;
}

void SimulateGame(const char* desc
                  , int numPlayers
                  , const int* chipDistribution
                  , long seed
                  , const int* startingChips
                  , int maxRounds
                  , ostream& outputStream)
{

    // create the game object
    Game game(numPlayers, chipDistribution, seed);

    // initialize our expected chips array
    double* expectedChips= new double[numPlayers];
    for (int i = 0; i < numPlayers; i++) {
        expectedChips[i] = 0;
    }

    double roundsWithWinner = 0;
    // run 100000 simulations
    for (int s = 0; s < 100000; s++) {

        // play a single game
        const int* playerChips = game.playGame(startingChips, maxRounds);

        // keep track of chips held at end of game
        for (int i = 0; i < numPlayers; i++) {
            expectedChips[i] += playerChips[i];
        }

        // was there a single winner?
        if (game.countPlayersWithChips() == 1) {
            roundsWithWinner += 1;
        }

        // for simulation iterations of 100, 1000, 10000 and 100000, write the output
        if ((s + 1) == 100 || (s + 1) == 1000 || (s + 1) == 10000 || (s + 1) == 100000)
 {
            cout << desc << ',' << (s + 1) << endl;
            outputStream << desc << ',' << (s + 1) << ',' << (roundsWithWinner/(s+1));
            for (int i = 0; i < numPlayers; i++) {
                double ev = expectedChips[i] / (s + 1);
                outputStream << ',' << ev;
            }
            outputStream << endl;
        }

    }
}


int main()
{
    ofstream outfile;

    // open the file
    outfile.open("lcr_output.csv");

    // write a header
    outfile << "Game,MaxRounds,RoundsWithWinner" ;
    for (int i = 0; i < 9; i++) {
        outfile << ",Player" << (i + 1);
    }
    outfile << endl;

    // standard game
    int chipDistribution[] = { 0,0,0,1,2,3 };
    int playerChips[] = { 10, 10, 10, 10, 10, 10, 10, 10, 10};
    SimulateGame("standard", 9, chipDistribution, (long)time(0), playerChips, 100, outf
ile);
```

```

    // standard game - one player has more
    int playerChipsTilted[] = { 5, 5, 5, 5, 50, 5, 5, 5, 5, 5 };
    SimulateGame("standard-tilted", 9, chipDistribution, (long)time(0), playerChipsTilted, 100, outfile);

    // game with greater chance of passing but no center
    int chipDistributionNoCenter[] = { 0,0,1,1,3,3 };
    SimulateGame("nocenter", 9, chipDistributionNoCenter, (long)time(0), playerChips, 100, outfile);

    SimulateGame("nocenter-tilted", 9, chipDistributionNoCenter, (long)time(0), playerChipsTilted, 100, outfile);

    // game with greater chance of passing but no 'holding'
    int chipDistributionNoHold[] = { 1,1,2,2,3,3 };
    SimulateGame("nohold", 9, chipDistributionNoHold, (long)time(0), playerChips, 100, outfile);

    SimulateGame("nohold-tilted", 9, chipDistributionNoHold, (long)time(0), playerChipsTilted, 100, outfile);


    // close the file
    outfile.close();

    return 0;
```

PDF document made with CodePrint using Prism lcr_output.csv



Game	MaxRounds	RoundsWitł	Player1	Player2	Player3	Player4	Player5	Player6	Player7	Player8	Player9
standard	100	0.93	0.89	0.86	0.9	0.83	0.79	0.82	0.77	0.79	0.86
standard	1000	0.929	0.859	0.856	0.846	0.867	0.827	0.847	0.826	0.824	0.824
standard	10000	0.9185	0.9629	0.9493	0.9429	0.9434	0.941	0.9454	0.9454	0.9393	0.9469
standard	100000	0.91715	0.96719	0.96245	0.96209	0.95786	0.96007	0.9606	0.96149	0.95963	0.95492
standard-tilted	100	0.96	0.2	0.21	0.29	0.25	14.16	0.28	0.24	0.22	0.23
standard-tilted	1000	0.955	0.245	0.264	0.292	0.344	13.261	0.313	0.27	0.254	0.251
standard-tilted	10000	0.9629	0.2086	0.2235	0.2568	0.3261	13.3756	0.2742	0.225	0.2104	0.2076
standard-tilted	100000	0.96118	0.21737	0.23372	0.26883	0.33179	13.2976	0.28475	0.23722	0.21756	0.21255
nocenter	100	0	10.24	11.47	8.24	7.65	10.71	10.77	11.09	10.27	9.56
nocenter	1000	0	10.456	9.551	9.848	9.782	9.875	10.588	10.181	10.421	9.298
nocenter	10000	0	10.0622	9.9741	9.9671	9.9332	10.0115	10.0738	10.0541	9.9827	9.9413
nocenter	100000	0	10.0792	10.0226	9.97776	9.98694	10.0044	10.0039	10.0282	9.97074	9.92619
nocenter-tilted	100	0	5.31	6.25	6.58	8.17	37.41	9.73	6.65	4.95	4.95
nocenter-tilted	1000	0	5.632	5.278	6.435	8.342	38.317	9.111	6.34	5.506	5.039
nocenter-tilted	10000	0	5.4725	5.5241	6.4034	8.526	38.4153	8.3556	6.3394	5.6432	5.3205
nocenter-tilted	100000	0	5.44319	5.6143	6.3839	8.52091	38.2718	8.51219	6.39255	5.62751	5.23368
nohold	100	0.78	2.33	2.24	2.34	2.28	2.32	2.38	2.28	2.28	2.29
nohold	1000	0.757	2.545	2.529	2.528	2.537	2.535	2.54	2.527	2.532	2.52
nohold	10000	0.7522	2.5937	2.5813	2.5826	2.5861	2.5764	2.5844	2.5746	2.5838	2.5768
nohold	100000	0.7554	2.56498	2.5517	2.55044	2.55043	2.55091	2.54964	2.54789	2.54761	2.54096
nohold-tilted	100	0.94	0.32	0.31	0.31	0.41	22.72	0.34	0.32	0.3	0.31
nohold-tilted	1000	0.94	0.314	0.331	0.34	0.376	22.287	0.336	0.312	0.305	0.305
nohold-tilted	10000	0.9371	0.3264	0.3408	0.3588	0.3851	22.5399	0.3473	0.3233	0.3207	0.3204
nohold-tilted	100000	0.93843	0.31986	0.33211	0.35373	0.37755	22.4434	0.3443	0.31664	0.31308	0.31393

Q1.

From the table above, I notice that the standard-tilted game resulted in the most single -player winners, where a player has much more chips than others, and also there is equal possibility to move to left, center or right. Intuitively, compared to non-tilted games, I think if the difference of chips among all the players is larger, it will be more likely to have one player left when all the others owning less chips finished all the moves. In addition, nocenter games have no possibility of ending with a winner. Also, nohold games are more volatile since every roll dice would lead to a move, which brings more uncertainty.

Thus, the standard-tilted game tends to have more rounds with a winner.

Q2.

In order to judge how quickly the simulated runs converge into stable expected values, we need figure out after how many times of simulations, the expected values tend to be steady. From the table above, I notice that the nohold-tilted game converges the quickest, since the expected values tend to be steady with 1000 times of simulations and are almost same to those with 100000 times of simulations in the nohold-tilted game. Similarly, the standard game converges the slowest, since the expected values are constanly increase in this case even with 100000 times of simulations. As for the reason the convergence time differ, I think first, there is more possibility to move, either left, center or right in the nohold games, which accelerates the convergence rate. Besides, tilted games set less chips for almost all the players except one with much more. Those with less chips will sit out the round quicker.

Thus, nohold-tilted game converges most quickly while standard game converges slowest.