CSE 3100 Systems Programming	Fall 2025
Homework #3	

Do not forget to submit your work to gradescope before the deadline. A Makefile is provided.

Exercise 1. (50 points) Diffusion

In this exercise, we use a 3D random walk to simulate a diffusion process. Imagine a particle starting at the origin $(0, 0, 0)$ that has equal probabilities to go in 6 possible directions - left, right, backward, forward, down, and up. For example, when the particle is at (x, y, z), with equal probability 1/6, its next location is at $(x-1, y, z), (x+1, y, z), (x, y-1, z), (x, y+1, z), (x, y, z-1)$ or $(x, y, z+1)$.

The particle will conduct the random walk for n steps. We are interested in the distribution of the final locations of particles after each takes n steps. Specifically, we would like to know the distribution of the distance between the final location and the origin. In order to obtain this distribution, we simulate m such particles, and check the proportion of the particles that lies within rn distance from the origin, where r is a real number between 0 and 1. Note all the particles will be within a sphere with radius n since particles only move n steps and the furthest they can go is a distance n from the origin. In our simulation, we will calculate the proportion of particles that are within rn from the origin for $r = 0.05, 0.10, 0.15, \ldots, 0.90, 0.95, 1.00$. Below is the main() function for this program. Note how we use command line arguments.

```
int main(int argc, char *argv[])
{

    if(argc != 3)
    {
        printf("Usage: %s n m\n", argv[0]);
        return 0;
    }
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    srand(12345);
    diffusion(n, m);
    return 0;
}
```

We need to implement the function

```
void diffusion(int n, int m)
{

}
```

In this function, we need to dynamically allocate memory to represent a 3D grid where x, y, z coordinates range from $-n$ to n. For all the possible x, y, z coordinates within the range, we save the number of particles that end up at the location (x, y, z). During the simulation, if the final location of a particle is (x, y, z), the corresponding value will be incremented by 1.

Generate a random number using `rand() % 6`. If this number is 0 or 1, you should move left or right on the x-axis, respectively. If it is 2 or 3, you should move up or down on the y-axis, respectively. If it is 4 or 5, move up or down on the z-axis, respectively.

In the implementation of `main`, we use the below function to print out results.

```
void print_result(int *grid, int n)
{
    printf("radius density\n");
    for(int k = 1; k <= 20; k++)
    {
        printf("%.2lf   %lf\n", 0.05*k, density(grid, n, 0.05*k));
    }
}
```

The function

```
double density(int *grid, int n, double r)
{

}
```

returns the proportion of particles in the grid that are within the sphere of rn radius from the origin. Note that to implement the above function, we need to calculate the Euclidean distance between (x, y, z) and the origin $(0, 0, 0)$. The formula is $\sqrt{x^2 + y^2 + z^2}$. Think about how we can avoid using the square root in our code.

Below are outputs from a few sample runs. We can use these outputs to check our code.

```
$ ./diffusion 10 1000000
radius density
0.05   0.019415
0.10   0.019415
0.15   0.196635
0.20   0.263900
0.25   0.465962
0.30   0.542348
0.35   0.686846
0.40   0.826187
0.45   0.908716
0.50   0.938628
0.55   0.982609
0.60   0.991070
0.65   0.997860
0.70   0.999031
0.75   0.999883
0.80   0.999959
0.85   0.999996
0.90   0.999996
0.95   1.000000
1.00   1.000000
$ ./diffusion 20 1000000
radius density
0.05   0.007120
0.10   0.113322
0.15   0.272127
0.20   0.517046
0.25   0.690763
0.30   0.857808
```

```
0.35   0.934197
0.40   0.979449
0.45   0.994164
0.50   0.998872
0.55   0.999802
0.60   0.999970
0.65   0.999999
0.70   1.000000
0.75   1.000000
0.80   1.000000
0.85   1.000000
0.90   1.000000
0.95   1.000000
1.00   1.000000
```

Exercise 2. (50 points) Monopoly
In this exercise, we will implement a simplified Monopoly game. We will see whether and when pure luck can create extreme inequality in wealth.

Our Monopoly game has two major components: an array of players and an array of properties. The number of players is m and the number of properties is n.

The n properties are arranged in a circular fashion. The index of the properties starts from 0 and goes to $n-1$ in a clockwise direction. Then the index goes back to 0, the starting point.

At the beginning of the game, all the players have n dollars as their initial balance. All the player start at property 0, and players take turns according to the increasing order of their id. When it is a player's turn, the player tosses two dice to determine the number of steps they need to move in a clockwise direction. The face value of the two 6-sided dice are summed to find the number of steps. The player moves this number of steps and lands on the corresponding property. If this property is owned by someone other than this player, this player needs to pay rent to the owner. Otherwise, if this property has no owner yet, it will be owned by this player. Rent increase linearly from 1 to n between the first and final properties.

Whenever a player goes back to or passes over property 0, n dollars will be awarded to the player.

The game ends when a player is required to pay more money than they own. When this happens, that player pays all of their money to the player who owns the property they just landed on. At the end of the game, we will estimate the "inequality" of the players as the highest wealth divided by the average wealth.

To describe a player, we need to use the structure

```
typedef struct Player
{
    int id;
    int loc;
    long balance;
} TPlayer;
```

Here, `id` is in the range $0 \ldots m-1$, `loc` is in the range $0 \ldots n-1$, and `balance` represents the wealth accumulated by the user.

Similarly, to describe a property, we use the following structure

```
typedef struct Property
{
    int id;
    int owner_id;
    int rent;
} TProperty;
```

Here, `id` is in the range $0 \ldots n-1$, `owner_id` is in the range $0 \ldots m-1$, and `rent` is the rent of the property.

Below is the main() function of the program; note how we use the command line arguments. Here m is the number of players, n is the number of properties, and *rounds* is the maximum rounds of game in case the game does not end.

```
int main(int argc, char *argv[])
{
    if(argc != 4)
    {
        printf("Usage: %s m n rounds\n", argv[0]);
        return -1;
    }
    int m = atoi(argv[1]);
    int n = atoi(argv[2]);
    int rounds = atoi(argv[3]);

    TPlayer p[m];
    TProperty prop[n];

    for(int i = 0; i < n; i++)
    {
        prop[i].id = i;
        prop[i].owner_id = -1;
        prop[i].rent = i + 1;
    }

    for(int j = 0; j < m; j++)
    {
        p[j].id = j;
        p[j].loc = 0;
        p[j].balance = n;
    }
    monopoly(m, n, p, prop, rounds);
    return 0;
}
```

Note how we initialize the player array p[m] and the property array prop[n]. The owner_id for each property is set to -1 since, at the beginning, it is not owned by any player. We use -1 instead of 0 because id 0 is the id for the first player.

We need to implement the following function.

```
//Implement the following function
void monopoly(int m, int n, TPlayer p[], TProperty prop[], int max_rounds)
{
```

```
    srand(12345);
    int rounds = 1;




    //The following line of code is needed for correct output
    print_result(m, p);
    printf("after %d rounds\n", rounds);
}
```

Below are outputs from a few sample runs. We can use these outputs to check our code.

```
$ ./monopoly 15 100 100000
      id    balance
       0       67
       1       95
       2      204
       3      244
       4      102
       5      228
       6        0
       7       72
       8       33
       9        0
      10       54
      11       27
      12       76
      13       18
      14      280
average: 100.000000 max: 280, max/average = 2.800000
after 5 rounds

$ ./monopoly 20 100 100000
      id    balance
       0      340
       1      200
       2      203
       3      260
       4       74
       5      137
       6      199
       7       68
       8       40
       9        0
      10       39
      11       46
      12       16
      13      137
      14       47
      15       56
      16       28
```

```
      17       15
      18       57
      19       38
average: 100.000000 max: 340, max/average = 3.400000
after 5 rounds
```