CSE 3100 Systems Programming	Spring 2024
Homework #2	

Please remember to submit your code to gradescope.

Exercise 1. (50 points) Sum of Odd Numbers
In this problem we will write a recursive function to partition a positive integer into sum of multiple distinctive positive odd numbers. Moreover, these positive odd numbers need to be within a given bound.

For example, our recursive function will show how to sum up 12 different positive odd numbers to 200, and each of these odd numbers is within 30. One solution for this problem is

```
1 3 7 13 15 17 19 21 23 25 27 29
```

At the same time, the function will return 1 to indicate the solutions exist.

On the other hand, if there are no solutions for a problem, our recursive function should return 0 to indicate the fact.

The main function of the code is already given, as shown below.

```
int main(int argc, char *argv[])
{
    if(argc != 4) return -1;

    int count = atoi(argv[1]);
    int bound = atoi(argv[2]);
    int value = atoi(argv[3]);

    //oddSum(12,30,200);
    //oddSum(10,20,100);
    //oddSum(20,20,200);
    oddSum(count, bound, value);
    return 0;
}
```

Moreover, the implementation for the function oddSum is also given as below.

```
void oddSum(int count, int bound, int value)
{
     if(value <= 0 || count <= 0 || bound <= 0) return;

     if(bound % 2 == 0) bound -= 1;

     if(!oddSumHelp(count, bound, value)) printf("No solutions.\n");
     else printf("\n");
}
```

The argument count specifies the number of odd numbers used for the sum. The argument bound specifies the bound of each odd numbers. The argument value specifies the intended sum of the odd numbers.

What is left for us to implement is the function oddSumHelp, which has the same parameter list as oddSum. It returns 1 if a solution exists, or it returns 0 if there are no solutions. This function also prints out a solution if a solution exists.

```
int oddSumHelp(int count, int bound, int value)
{


}
```

Note we only print out one solution if multiple solutions exist. Our solution will always try to include the largest possible odd number in the solution when possible. For example, both of the following are solutions to oddSum(12,30,200). But we only print out the first solution since it uses 21 in the solution while the second one skips 21.

```
1 3 7 13 15 17 19 21 23 25 27 29
5 7 9 11 13 15 17 19 23 25 27 29
```

Below are some example outputs of the program.

```
$ ./oddSum 12 50 200
1 3 5 7 9 11 13 15 17 23 47 49
$ ./oddSum 12 30 200
1 3 7 13 15 17 19 21 23 25 27 29
$ ./oddSum 20 10 300
No solutions.
```

Exercise 2. (50 points) Bounded 2D Random Walk
In this problem, we will write a program to simulate a 2D random walk. Imagine a random walker starting at the origin $(0, 0)$ that with equal probabilities goes up, right, down and left. For example, when the walker is at (x, y), with equal probability 1/4, their next location is at $(x, y-1), (x+1, y), (x, y+1)$, or $(x-1, y)$.

Given a positive integer n, a square is defined by the following four points: $(-n, -n), (-n, n), (n, n)$, and $(n, -n)$. We are interested in knowing, on average, what fraction of points within this square the walker visits before they touch one of the edges of the square, given they start their walk from $(0, 0)$.

One extreme example is $n = 1$. When the walker starts from $(0, 0)$, after one step, they will touch one of the edges of the square. There is only 1 point inside the square, and the walk visits this point before they touch one of the edges. Therefore, this fraction is 1 for $n = 1$.

In the starter code 2d-walk.c, implement the following function

```
double two_d_random(int n)
```

This function should return the fraction mentioned above. n is the integer mentioned above to define the square boundary.

In the function, we need to use an array to keep track of which coordinates have been visited inside the squared defined by the two coordinates $(-n, -n)$ and (n, n). When deciding which way to go for the next step, generate a random number as follows.

```
r = rand() % 4;
```

and treat $r = 0, 1, 2, 3$ as going up, right, down and left respectively.
The random walk should stop once the x coordinate or y coordinate reaches $-n$ or n. The function should return the fraction of the visited (x, y) coordinates inside (not including) the square.

Below is the main() function. For each $n = 1, 2, 4, \ldots, 64$, we call the function

`two_d_random(n)`

$1,000$ times and calculate and print out the mean of the covered fractions for the given n.

```
//Do not change the code below
int main(int argc, char* argv[])
{
        int trials = 1000;
        int i, n, seed;
        if (argc == 2) seed = atoi(argv[1]);
        else seed = 12345;

        srand(seed);
        for(n=1; n<=64; n*=2)
        {
                double sum = 0.;
                for(i=0; i < trials; i++)
                {
                        double p = two_d_random(n);
                        sum += p;
                }
                printf("%d %.3lf\n", n, sum/trials);
        }
        return 0;
}
```

Below is the desired output. We can use the output to check our code. You can optionally set the seed for the random numbers generated as a command line argument with `./2d-walk [seed]`

```
$ ./2d-walk
1 1.000
2 0.367
4 0.221
8 0.154
16 0.122
32 0.101
64 0.085
```