Chem 260 Project B

Due by midnight Wednesday, December 16, 2015 (no credit if turned in late)

Largest of N normally distributed numbers

The failure of biological and physical systems often depends not on the average behavior of its components, but on the behavior of one or a few "outliers". For example, a tumor can begin when one cell out of the trillions in the body is transformed into a cancerous cell, and a physical structure can fail when a single critical connector or support fails, even if all of the other components are fine.

You can create a simple model of how the risk of a system failure grows with the number of components by generating a set of *N* normally distributed random numbers (that is, a set of *N* numbers that would form a bell curve if made into a histogram), and then look at the value of the largest number in that set of *N* values. If you repeat this calculation many times, you will get an estimate of how far the largest single individual value deviates from the average of a set of *N* values, which is an approximate estimate of how the likelihood of a system failure grows with the number of critical components. If the set includes a lot of numbers, the largest value will be much larger than the average value of the set. For example, the following ten normally distributed numbers have a mean of 0.025 (and a standard deviation of 0.969), but the largest value in the set is 1.783.

(-0.125, -0.247, -0.049, -1.468, 1.416, 0.389, 1.783, -0.124, -0.652, -0.675)

In this project you will write a C program (called **`maxnorm.c`**) that will investigate how the maximum value in a set of normally distributed random numbers (with mean=0.0 and sd=1.0) changes with the number of values in that set. Specifically, your program will generate sets of *N* normally distributed random numbers and pick the largest from that set. You will do this for a range of *N*'s (from 1 to 200) and you will do this 5000 times for each value of *N* and calculate the average largest value for each *N*. Here is the pseudocode for this project:

```
Loop over values of N (1 to 200):
   Initialize sum variable to zero for calculating average maximum value
   Loop over trial (0 to ntrials-1):
        Loop over i (0 to N-1):
             Calculate normally distributed random number
             Put that number into an array
        Sort the array of N random numbers
        Select largest value (i.e. the final value in the sorted list)
        Add largest value to sum variable
   Divide sum by number of trials to get average maximum value
   Print out N and average maximum value
```

To generate a normally distributed random number you can use the function **`rnorm()`** that is provided in the projectB directory in the file "**`rnorm.c`**". This function will automatically seed the random number generator when first called. Here's an example of how to use and compile a program using the **`rnorm()`** function, using the program **`testrnorm.c`** which is in the projectB directory.

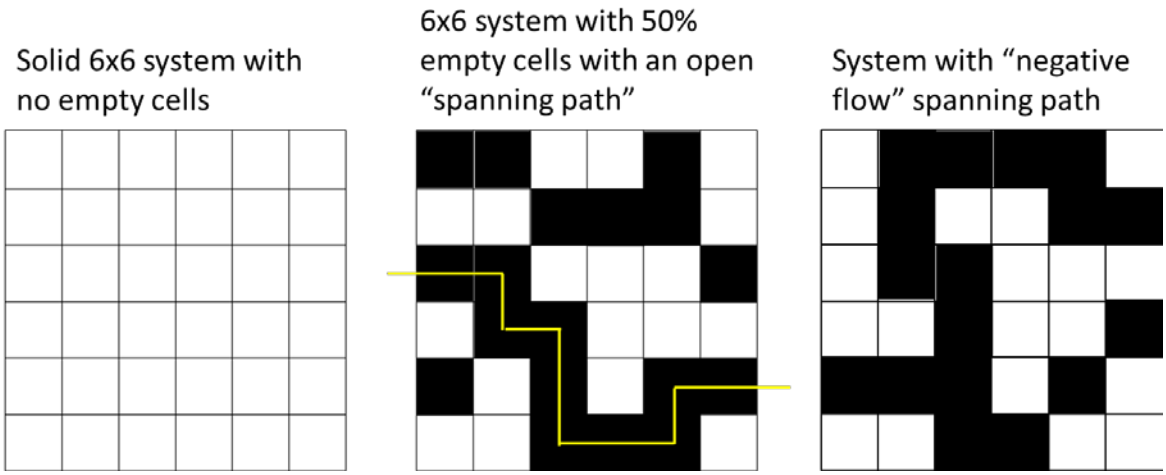
```
#include <stdio.h>
float rnorm(void); /* Need to include this prototype */
int main(void) {
  int i;
  for (i=0; i<10; i++) {
    printf("%f\n",rnorm());
  }
}
```

To compile: **`gcc –o testrnorm testrnorm.c rnorm.c -lm`**

[Extra Credit problem on next page]

Extra credit problem (challenging): Percolation in 2D media

In Project 2, we studied the percolation problem by determining if a diffusing particle could get from the center of the grid to the edge within some fixed number of steps. In this extra credit problem, you will write a program to study the problem using a different approach: calculating if there is an edge-to-edge path through which a particle could diffuse. The grid is represented by an *N*x*N* grid of cells just like in the diffuse.py, dla.py, and perc.py programs we saw previously, where each cell is either filled or empty. The system is considered permeable if there is a "spanning path" of empty cells from the left side to the right side of the system—see middle figure below where the black squares are the empty cells. (Note that diagonal empty cells are not considered connected.) When calculating whether there is a spanning path between the left and right sides of the system, you can simplify your program by ignoring paths that have "negative flow" as shown at right in the figure. That is, you can assume that your path from the left to the right side of the system will either be moving to the right or up or down, but never back towards the left. However, you are welcome to include such paths if you would prefer.



Your program should read in two values from stdin:

1. N: the number of cells along one edge of the system
2. Ntrials: the number of systems to construct and test at each density

Your program should loop over the fraction of empty cells (i.e. the density) from 0.0 to 1.0 in steps of 0.01. At each density do the following, Ntrials times:

1. Randomly fill the system with empty cells to reach the current density.
2. Determine whether there is a spanning path from the left to right
3. If there is a spanning path, increment a counter

After the loop over Ntrials, print out the density and the probability of finding a spanning path at that density, and then continue to the next density.