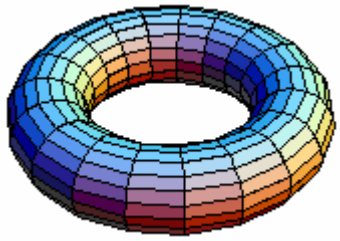
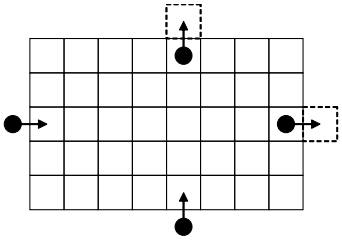
CSCI 135 Software Design and Analysis I
Extra Credit Project
Living on a random torus

Saad Mneimneh
Computer Science
Hunter College of CUNY



What would life on a torus look like? A torus is the two-dimensional surface of a donut shape (shown above). Such a surface can, therefore, be represented as a two dimensional array that wraps around in both directions. This means that the modulo operator should be used when indexing the array. For example, if the array is an $m \times n$ array called a, then $a[i][m-1]$ is to the left of $a[i][0]$, and $a[n-1][i]$ is above $a[0][i]$.

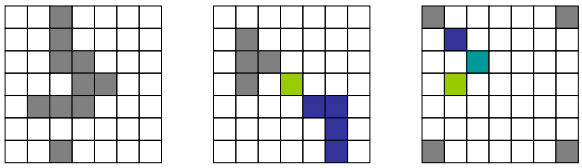


But the size of the torus is not known in advance, so the array will be created dynamically as an $m \times n$ array of spots (a C++ vector can also be used to avoid explicit dynamic memory allocation). Each spot is chosen randomly to be either land or water. For instance, the following simple class can be used to represent a spot:

```
class Spot {
 public:

  bool land;
  int i,j;      //position
  bool visited; //keep reading
};
```

The main goal of this project is to compute the average number of islands produced on a random torus. An island is defined as a set of land spots such that we can go between any two spots by traveling on neighboring land, i.e. only up, down, left, or right, and without crossing any water spots (while keeping in mind that we can wrap around in both directions). Here are some examples:



A suggested skeleton for the project is the following:

```
//torus.h

class Torus {
  Spot ** tor;  //or vector<vector<Spot> > tor;
  int m;
  int n;

 public:
  Torus();  //constructs a 100x100 torus
  Torus(int n); //constructs a nxn torus
  Torus(int m, int n); //constructs an mxn torus
  void regenerate(double p); //regenerates the torus randomly with prob. p for land
  int islands(); //returns number of islands
};


//torus.c
//you have to implement it


//main.c

int main() {
  srand(time(0));
  int m;
  int n;
  cin>>m,n;

  const int trials=...;
  double p=...;
  Torus t=Torus(m,n);
  double average=0;
  for (int i=0; i<trials; i=i+1) {
    t.regenerate(p);
    average=average+(double)t.islands()/trials;
  }
  cout<<average<<''\n'';
}
```

(a) The bulk of the work is the islands() member function. One possible implementation of this function is the following. Start at any spot (i, j) of land (i.e. **tor[i][j].land** is true) and make that spot visited (by setting the boolean **tor[i][j].visited** to true for that spot). Add a pointer to the spot to an empty vector. Repeatedly, remove the last pointer from the vector and check the up, down, left, and right neighbors of the corresponding spot (they all exist due to the modulo operator). For each one, if the spot is an unvisited land (i.e. **tor[i][j].visited** is false), make that spot visited (i.e. set **tor[i][j].visited** to true) and add its pointer to the end of the vector. The process stops when the vector becomes empty, and we've identified **one** island. This process is then repeated until all land spots are visited.

(b) Observe that the vector described in the above process acts like a stack to save some history. Provide an alternative implementation for counting islands that makes use of recursion. This can be done by adding a private function to class Torus:

```
class Torus {

  ...

  void visit(Spot& spot);

 public:

  ...
};
```

The function visit recursively visits all (at most four) unvisited neighboring lands of the given spot, until the entire island is visited. Each time a spot is visited, its boolean is set (so it will not be visited again by the recursive process). In this way, the islands() function can be simply implemented as follows:

```
int Torus::islands() {
  int count=0;
  for (int i=0; i<m; i=i+1)
    for (int j=0; j<n; j=j+1)
      if (tor[i][j].visited==false && tor[i][j].land) {
        count=count+1;  //new island discovered
        visit(tor[i][j]);  //visit it
      }
  return count;
}
```

(c) Use either implementation to compute the average number of islands divided by mn as a function p. Verify that for large values of m and n, and small values of p, this is approximately:

$$p(1-p)(1-p-p^2)$$

(d) Modify your program so that it outputs the average for every value of $p \in \{0.05, 0.1, 0.15, \ldots, 0.9, 0.95\}$ to a file. Each line in the file must have two entries separated by a comma: the value of p, and the computed average divided by mn. Use this file in Excel (or an equivalent tool) to plot this scaled average number of islands vs. p. Superimpose $p(1-p)(1-p-p^2)$ on the same plot.