Percolation

Percolation. Pour liquid on top of some porous material. Will liquid reach the bottom?

Abstract model.
- N-by-N grid of sites.
- Each site is either blocked or open.

Percolates
- An open site is full if it is connected to the top via open sites.

Applications.
- [Chemistry, materials science]
  - Chromatography.
  - Natural gas through semi-porous rock.
  - How gas in coal mine permeates gas mask filter.
  - Flow of electricity through network of resistors.
  - ...
A Scientific Question

**Random percolation.** Given an N-by-N system where each site is blocked with probability \( p \), what is the probability that the system percolates?

**Remark.** Famous open question in statistical physics, but no known mathematical solution.

**Recourse.** Take a computational approach.

Java Simulation

**Data representation.** Use one N-by-N boolean matrix to store which sites are blocked; use another one to compute which sites are full.

**Boolean matrix library.** Create a library to support common operations on boolean matrices (create, read, and print).

```
public class BooleanMatrix {

    public static boolean[][] read() {
        int N = StdIn.readInt();
        boolean[][] a = new boolean[N][N];
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                if (StdIn.readInt() != 0) a[i][j] = true;
        return a;
    }

    public static void print(boolean[][] a) {
        // print matrix to standard output
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                a[i][j] = StdRandom.bernoulli(p);
        return a;
    }

    public static boolean[][] random(int N, double p) {
        boolean[][] a = new boolean[N][N];
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                a[i][j] = StdRandom.bernoulli(p);
        return a;
    }
}
```
Percolation Scaffolding

**Approach.** Write the easy code first. Fill in details later.

```java
public class Percolation {

    public static boolean[][] flow(boolean[][] blocked)
    {
        // return boolean matrix representing full sites

        public static boolean percolates(boolean[][] blocked) {
            int N = blocked.length;
            boolean[][] full = flow(blocked);
            for (int i = 0; i < N; i++)
                if (full[N-1][i]) return true;
            return false;
        }
    }
}
```

Directed Percolation

**Next step.** Start by solving an easier version of the problem.

**Directed percolation.** Is there a path of open sites from the top to the bottom that never travels up the grid?

**Q.** How to determine which sites in row i+1 are filled?

**A.** Depends solely on which sites in row i+1 are open and which sites in row i are filled.

**Algorithm.**

- Scan row i+1 from left to right, identifying contiguous open sites.
- If any open site is beneath a full site, mark all sites in that contiguous group as full.

<table>
<thead>
<tr>
<th>row i</th>
<th>row i+1</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="closed" alt="blocked" /></td>
<td><img src="open" alt="open" /></td>
</tr>
</tbody>
</table>

Contiguous open sites
Directed Percolation Calculation

Remark. Compact and correct, but not necessarily simple.

```java
public static boolean[][] flow(boolean[][] blocked) {
    int N = a.length;
    boolean[] full = new boolean[N][N];
    for (int j = 0; j < N; j++)
        full[0][j] = !a[0][j];
    for (int i = 1; i < N; i++) {
        for (int j = 0; j < N; j++) {
            boolean fill = false;
            for (int k = j; k < N && !a[i][k]; k++)
                fill = fill || full[i-1][k];
            if (fill)
                full[i][j] = true;
        }
    }
    return b;
}
```

Data Visualization

Visualization. Use standard drawing to visualize large inputs.

```java
public class Visualize {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        double p = Double.parseDouble(args[1]);
        boolean[][] blocked = BooleanMatrix.random(N, p);
        boolean[][] full = Percolation.flow(blocked);
        booleanMatrix.show(full);
    }
}
```

Percolation: Testing

Testing. Use standard input and output to test small inputs.

```java
% java PercolationDown < testT.txt
6
0 1 0 0 0
0 1 1 1 1
0 0 1 1 0
0 0 0 0 1
0 1 1 1 1
1 1 1 0 1
true
```

Percolation Probability Estimate

Percolation probability estimate. Given N and p, run simulation M times and collect statistics.

```java
public class Estimate {
    public static double eval(int N, double p, int M) {
        int cnt = 0;
        for (int k = 0; k < M; k++) {
            boolean[][] blocked = BooleanMatrix.random(N, p);
            if (Percolation.percolates(blocked)) cnt++;
        }
        return 1.0 * cnt / M;
    }

    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        double p = Double.parseDouble(args[1]);
        int M = Integer.parseInt(args[2]);
        System.out.println(eval(N, p, M));
    }
}
```
**Percolation Probability Estimate**

*Percolation probability estimate.* Given N and p, run simulation M times and collect statistics.

- \% java Estimate 32 .2 10
- \% java Estimate 32 .6 10
- \% java Estimate 32 .4 100000
- 0.5809
- \% java Estimate 32 .4 100000
- 0.5816
- \% java Estimate 100 .4 100000
- 0.6783

**Running time.** Proportional to MN². [a lot of computation!]

**Memory consumption.** Proportional to N².

**Adaptive Plot**

*Adaptive plot.* To plot f in the interval \([x_0, x_1]\):
- Stop if interval is sufficiently small.
- Divide interval in half \(x_m = \frac{1}{2} (x_0 + x_1)\) and compute \(f(x_m)\).
- Stop if \(f(x_m)\) is close to \(\frac{1}{2} (f(x_0) + f(x_1))\).
- Recursively plot \(f\) in the interval \([x_0, x_m]\).
- Plot the point \((x_m, f(x_m))\).
- Recursively plot \(f\) in the interval \([x_m, x_1]\).

**Net effect.** Short program that judiciously chooses values of \(p\) to produce a “good” looking curve without excessive computation.

**In Silico Experiment**

*Plot results.* Plot the probability that an N-by-N system percolates as a function of the occupancy probability \(p\).

**Design decisions.**
- How many values of \(p\)?
- For which values of \(p\)?
- How many experiments for each value of \(p\)?

**Percolation Plot: Java Implementation**

```java
public class PercPlot {
    public static void curve(int N, double x0, double y0, double x1, double y1) {
        double xm = (x0 + x1) / 2;
        double ym = (y0 + y1) / 2;
        double fxm = Estimate.eval(N, xm, 10000);
        if (xm - x0 < .01 || Math.abs(ym - fxm) < .0025) {
            StdDraw.line(x0, y0, x1, y1);
            return;
        }
        curve(N, x0, y0, xm, fxm);
        StdDraw.filledCircle(xm, fxm, .005);
        curve(N, xm, fxm, x1, y1);
    }

    public static void main(String args[]) {
        int N = Integer.parseInt(args[0]);
        PercPlot.curve(N, 0.0, 1.0, 1.0, 0.0);
    }
}
```
Phase transition. If $p < .407$, system almost always percolates; if $p > .407$, system almost never percolates.

Depth First Search: Java Implementation

```java
public static boolean[][] flow(boolean[][] blocked) {
    int N = blocked length;
    boolean[][] full = new boolean[N][N];
    for (int j = 0; j < N; j++)
        if (!blocked[0][j]) flow(blocked, full, 0, j);
    return full;
}

public static void flow(boolean[][] blocked, 
    boolean[][] full, int i, int j) {
    int N = a.length;
    if (i < 0 || i >= N || j < 0 || j >= N) return;
    if (blocked[i][j]) return;
    if (full[i][j]) return;
    full[i][j] = true;
    flow(blocked, full, i+1, j);
    flow(blocked, full, i, j+1);
    flow(blocked, full, i, j-1);
    flow(blocked, full, i-1, j);
}
```

Percolation: Recursive Solution

Percolation. Given an N-by-N system, is there any path of open sites from the top to the bottom.

Depth first search. To visit all sites reachable from i-j:
- If i-j already marked as reachable, return.
- Mark i-j as reachable.
- Visit the 4 neighbors of i-j recursively.

Percolation solution.
- Run DFS from each site on top row.
- Check if any sites in bottom row are marked as reachable.

Layers of Abstraction

Function call graph.
Lessons

Expect bugs. Run code on small test cases.

Keep modules small. Enables testing and debugging.

Incremental development. Run and debug each module as you write it.

Solve an easier problem. Provides a first step.

Consider a recursive solution. An indispensable tool.

Build reusable tools.