PERCOLATION
Class Meeting #2
COS 226 — Spring 2018

Based on slides by
Jérémie Lumbroso
— Motivation
— Problem description
— API
— Backwash
— Empirical Analysis
— Memory Analysis
What does Percolation model?
Likelihood of percolation

Depends on site vacancy probability $p$.

- $p$ low (0.4) does not percolate
- $p$ medium (0.6) percolates?
- $p$ high (0.8) percolates
Percolation phase transition

When \( N \) is large, theory guarantees a sharp threshold \( p^* \).
- \( p > p^* \): almost certainly percolates.
- \( p < p^* \): almost certainly does not percolate.

Q. What is the value of \( p^* \) ?

Other examples:
- Water freezing
- Ferromagnetic effects
Monte Carlo simulation

- Initialize $N$-by-$N$ whole grid to be blocked.
- Declare random sites open until top connected to bottom.
- Vacancy percentage estimates $p^\ast$. 

$N = 20$

135 open sites
Dynamic connectivity solution to estimate percolation threshold

Q. How to check whether an $N$-by-$N$ system percolates?
Dynamic connectivity solution to estimate percolation threshold

Q. How to check whether an $N$-by-$N$ system percolates?
- Create an object for each site and name them $0$ to $N^2 - 1$. 

$N = 5$

- open site
- blocked site
private int getIntFromCoord(int row, int col) {
    return N * row + col;
}

Or perhaps since this function will be used a lot, should it have a shorter name?

For ex.: site or location or cell or grid, etc., …
Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new site?
A. Mark new site as open; connect it to all of its adjacent open sites.

$N = 5$

open this site

up to 4 calls to union()
Dynamic connectivity solution to estimate percolation threshold

Clever trick. Introduce 2 virtual sites (and connections to top and bottom).
- Percolates iff virtual top site is connected to virtual bottom site.

efficient algorithm: only 1 call to connected()

\[ N = 5 \]

- open site
- blocked site

virtual top site

top row

bottom row

virtual bottom site
public class UF

UF(int N) initialize union-find data structure with N objects (0 to N – 1)

void union(int p, int q) add connection between p and q

boolean connected(int p, int q) are p and q in the same component?

int find(int p) component identifier for p (0 to N – 1)

int count() number of components

what **you** must do

both are APIs

what is provided
Why an API?
API = Application Programming Interface
—a contract between a programmers
—be able to know about the functionality without details from the implementation

Each of these modules could be programmed by anybody / implemented anyway
Example 1: Car

```java
public class Car {
    void turnLeft()
    void turnRight()
    void shift(int gear)
    void break()
}
```
A. Electrical?
B. Hybrid?
C. Gasoline?
D. Diesel?
E. Hydrogen cell?
Example 1: Car

```java
public class Car {
    void turnLeft()
    void turnRight()
    void shift(int gear)
    void break()
}
```
Example 2: Electrical Outlets

original API

API with added public members

is incompatible with rest of the clients
Why is it so important to implement the prescribed API? Writing to an API is an important skill to master because it is an essential component of modular programming, whether you are developing software by yourself or as part of a group. When you develop a module that properly implements an API, anyone using that module (including yourself, perhaps at some later time) does not need to revisit the details of the code for that module when using it. This approach greatly simplifies writing large programs, developing software as part of a group, or developing software for use by others.

Most important, when you properly implement an API, others can write software to use your module or to test it. We do this regularly when grading your programs. For example, your PercolationStats client should work with our Percolation data type and vice versa. If you add an extra public method to Percolation and call them from PercolationStats, then your client won't work with our Percolation data type. Conversely, our PercolationStats client may not work with your Percolation data type if you remove a public method.
Backwash problem

% java PercolationVisualizer input10.txt
Empirical Analysis
THEORY + PRACTICE
Power Law Running Times

— Typically most running times that are empirically measure are **power laws**

\[ cN^a \]

constant factor  \hspace{1cm} \text{exponent}

exponent  \hspace{1cm} \text{parameter (size of the instance)}

— Usually when other running times are involved such as \( N \cdot \log N \), \( N \cdot \alpha(n) \), \( \exp(N) \), it will be because of a known sub-algorithm
Doubling Hypothesis (1)

Assuming the running time is of the form:

\[ t(N) := c \cdot N^a \]

then, to find the exponent \( a \):

\[
\frac{t(2N)}{t(N)} = \frac{c \cdot (2N)^a}{c \cdot N^a} = \frac{c \cdot 2^aN^a}{c \cdot N^a} = 2^a
\]
**Doubling Hypothesis (2)**

**Recipe:**
- timing in N and 2N
- take log base 2 of ratio
- repeat for several points

\[
\log_2 \left( \frac{t(2N)}{t(N)} \right) = a
\]
Doubling Hypothesis (3)

**Tips:** 1) pick largest points; 2) repeat couple times

<table>
<thead>
<tr>
<th>N</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>800</th>
<th>1600</th>
<th>3200</th>
<th>6400</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>0.538473</td>
<td>0.0932774</td>
<td>0.163298</td>
<td>0.744645</td>
<td>2.5858</td>
<td>18.5561</td>
<td>141.455</td>
</tr>
</tbody>
</table>

 Weird value is an artifact

\[
\log_2\left(\frac{t(200)}{t(100)}\right) \approx -2.52927
\]

first try with N too small (noise)

\[
\log_2\left(\frac{t(6400)}{t(3200)}\right) \approx 2.93038
\]

then we get a more realistic value

\[
\log_2\left(\frac{t(3200)}{t(1600)}\right) \approx 2.84321
\]

and we can try to confirm

(if not, try to get larger point, such as N=12800)
What to do...

— ... to determine the constant?
Once exponent(s) is found, obtain by simple division.

— you have **two** variables (such as N and T)
Treat each separately (by making one variable vary, while the other remains constant).
sw = new Stopwatch(); // timer starts

ps = new PercolationStats(N, T); // operation we // want to measure

Timing = sw.elapsedTime; // time in seconds since // the Stopwatch was // created

if single observations too fast to measure, measure several operations at a time and average
Memory Analysis
Memory

In describing a program, memory usage can come in two forms: space and time. The space that you need to store data is measured in bytes, and the time that you need to process the data depends on the number of memory references required. Memory usage includes the size of the program itself and the size of the data that the program needs to operate on. Space is measured in bytes, and time is measured in milliseconds or seconds.

Memory footprint is the size of all the objects that are currently in memory. Each object takes up a certain amount of space, which depends on the type of object and its size. The size of an object is determined by the size of its references and the size of its fields. For example, a String object takes up 24 bytes of memory, and an int object takes up 4 bytes of memory.

When you allocate an object in memory, the JVM creates an instance of the object and stores it in memory. The object's memory is allocated in a heap, which is the area of memory that is used to store objects. The heap is divided into two parts: the young generation and the old generation. The young generation is used to store new objects, and the old generation is used to store objects that have been around for a while.

When an object is garbage collected, its memory is deallocated and returned to the heap. The garbage collector is responsible for finding and freeing memory that is no longer needed by the program. The garbage collector runs automatically in the background, and it is responsible for finding and freeing memory that is no longer needed by the program.

The garbage collector is an important part of the Java virtual machine, and it helps to ensure that the memory usage of a program is efficient. The garbage collector is able to allocate memory for an object and then free it when it is no longer needed. This allows the program to allocate memory dynamically as needed, and it helps to ensure that the memory usage of a program is efficient.

In the next section, we will talk about how to measure the memory usage of a program. We will talk about how to use the JVM's heap dump tools to measure the memory usage of a program. We will also talk about how to use the JVM's heap dump tools to measure the memory usage of a program.
Memory (2)

```java
public class Stack {
    private int N;  // size of the stack
    private Node first;  // top of stack

    private class Node {
        private double item;
        private Node next;
    }
    ...
}
```

### Memory Types and Sizes

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean[]</td>
<td>N + 24</td>
</tr>
<tr>
<td>char[]</td>
<td>2N + 24</td>
</tr>
<tr>
<td>int[]</td>
<td>4N + 24</td>
</tr>
<tr>
<td>double[]</td>
<td>8N + 24</td>
</tr>
</tbody>
</table>

### Memory Layouts

**Node**

- **object overhead (16)**
- **extra overhead (8)**
- **item (8)**
- **next (8)**

**Stack**

- **object overhead (16)**
- **N (4)**
- **padding (4)**
- **first (8)**
- **Node (40)** for each
Questions?

More on this in the precept!