Precept 1. Percolation, DFS, Quick-Find

□□ Welcome!

Hello, everyone, and welcome to your first COS 226 precept!

We're pretty psyched to get to the course content, but, before we do, a little background:

This is not a test. You are not being evaluated here. The questions and exercises you'll do in this precept are not graded. Instead, they are meant to test the limits of your understanding, so you can surpass those limits!

Reflect as you go. To get the most out of your time here, you'll want to understand *why* we're asking the questions we're asking, not just how to answer them. Sure, getting the right answer is important, but don't let your pursuit of green check marks distract you from evaluating your understanding of the material as you go.

Pair up and share a computer. We've designed these exercises to be discussed and collaborated on. Find a partner to work with, discuss each question, and try your best to support one another.

In-person attendance = participation credit. To earn full participation credit in this course, you'll have to attend in-person precept (or have excused absences). While you're free (and encouraged) to access Ed Lessons from home, this doesn't substitute for attending precept.

Okay, let's get started!

Question *Submitted Sep 3rd 2021 at 3:20:18 pm*

Please enter your NetID here, along with the NetIDs of anyone you're sharing this screen with.

If you're doing these exercises alone, please just put down your own NetID.

i Not sure of someone's NetID? The fastest way to look it up is Advanced People Search.

sw42

betanzos

Percolation

"Does it percolate?"

That's the question that'll motivate the study of our first data structure, the *union-find*. But, before we code with this data structure, we'll need to know more about the important problem we're using it to solve.

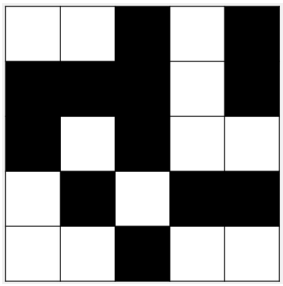
Percolation, a word you might be familiar with if you're a coffee drinker, comes from a root that means "to filter" or "trickle through". To brew a fresh pot, we might pour water over coffee grounds and watch as it slowly *percolates*, finding its way down to the cup or vessel below.

Perhaps some caffeinated beverages were involved when mathematicians, physicists, and even urban planners first realized that percolation could serve as a mathematical model for phenomena they observed in their fields.

In fact, percolation is such an interesting topic that there are entire books written about it and, despite that, there's still lots we don't know.

When used as a model, we first thing we often ask is whether a particular system percolates or not. By this, we mean: is there an open path from the top to the bottom?

Does this system percolate?

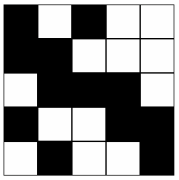


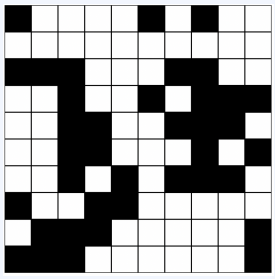
No, because there aren't any contiguous paths of open sites from the top to the bottom.

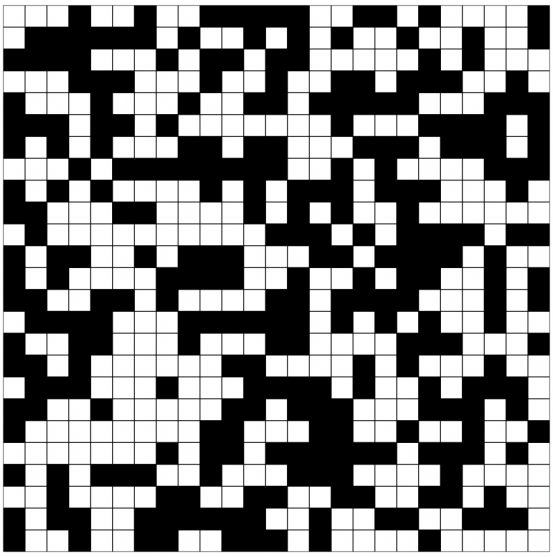
Question 1 *Submitted Sep 3rd 2021 at 1:40:57 pm*

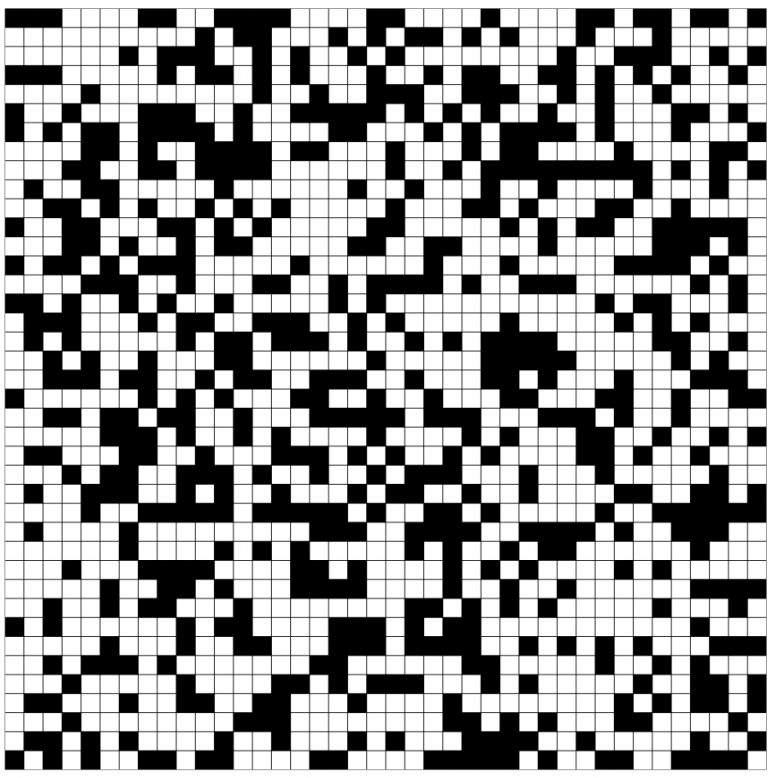
Which of these systems percolate?

Select all that apply.







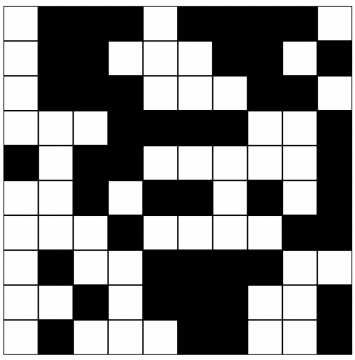


Question 2 *Submitted Sep 3rd 2021 at 1:42:31 pm*

We use two parameters to describe systems like these, n and p.

n tells us the size of the system (n-by-n) and p tells us the likelihood any given site is open.

If you had to guess, what is p in this system?



Give your answer as a decimal value.

0.52

Question 3 *Submitted Sep 3rd 2021 at 1:42:44 pm*

As you open sites at random, you increase the likelihood that a system percolates.

- (●) True
- ( ) False

Question 4 *Submitted Sep 3rd 2021 at 1:43:20 pm*

Opening just one additional site can *drastically* increase the odds that a system percolates.

- (●) True
- ( ) False

Question 5 *Submitted Sep 3rd 2021 at 1:43:42 pm*

We can determine the threshold at which a system is likely to percolate, known as $p_{critical}$, using a simple formula that comes to us from computational chemistry.

- ( ) True
- (●) False

Depth-First Search

On this page, you're asked to fill in the blanks of code that detects whether a system percolates.

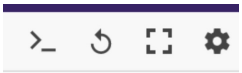
For our first approach, we've gone with a depth-first search. This strategy resembles what we (humans) do when we're solving a maze: follow a path until reaching a dead end, then trace back to before the dead end, and keep following new paths from there.

See if you can fill in the blanks to get this program to work as intended.

Use the "Mark" button to see if you're correct.

When you're ready, you can use the "✓ Solution" button to view the answers.

> In case this is your first Ed Lesson coding challenge: the buttons on the top right of the editor can be very useful! The third one lets you edit your code in full-screen.



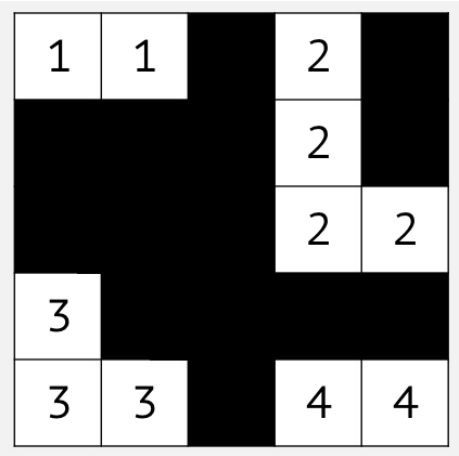
Coding with Clusters

Now we have a tool to test for percolation; depth-first search.

This works, but do we really need to conduct a brand new search every time we open a site?

In lecture, we saw how keeping track of *clusters* of open sites can make detecting percolation a lot easier. Rather than search for a path from top to bottom, we can just see if any top site is in the same cluster as any bottom site.

Let's say we use an `int[][]` called `cluster` to keep track of which sites are in which clusters. We might use it to store values like these:



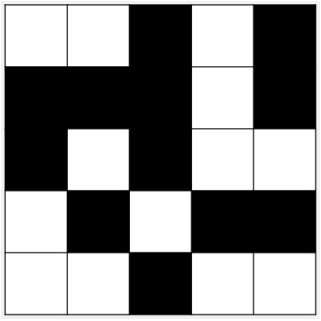
Question 1 *Submitted Oct 25th 2021 at 5:42:13 pm*

What should we do to the `cluster` array when we want to merge two clusters?

> Find each instance of one element and replace them with the other.

Question 2 *Submitted Sep 3rd 2021 at 2:04:30 pm*

If we started with all closed sites and we wanted to end up with this system, how many cluster merges would we have to do?



7

Question 3 *Submitted Sep 3rd 2021 at 2:05:03 pm*

Which of these ways of initializing `cluster` makes the most sense for our use case, assuming all sites are closed when we start?

- ( ) Accept the default value for an `int[][]`; initialize each element to `0`.
- (•) Assign a unique value to each element.

□ Quick-Find

Let's see if we can implement the strategies we discussed on the previous page.

Here, we're asking you to fill in the blanks on a data type that stores cluster information. This program is called `QuickFind2D` because it's an adapted version of the general-purpose `QuickFindUF` that we'll see first thing in lecture next week.

See if you can fill in the blanks to complete this code.

🚨 Before you go!

This page is simply here to tell you that though it's called *quick*-find, it's not the *quickest* or *best* union-find implementation. As you may have noticed, every call to `union()` takes n^2 time because it searches every element (n-by-n) to replace each instance of `xCluster` (with `yCluster`).

We'll hear about a few other kinds of union-finds on Tuesday, including one that cleverly stores its data to achieve significant efficiency gains -- *weighted quick-union*.

Reflection

Question 1 *Submitted Sep 3rd 2021 at 3:25:50 pm*

What were your biggest take-aways from precept today?

I learned more about DFS after only encountering it with Graph Traversals. I also learned that Quick-Find is not exactly as quick as its name suggests.

Question 2 *Submitted Sep 3rd 2021 at 3:25:55 pm*

Do you have any feedback about today's questions that the course staff could use to make this a better precept for future students?

N/A