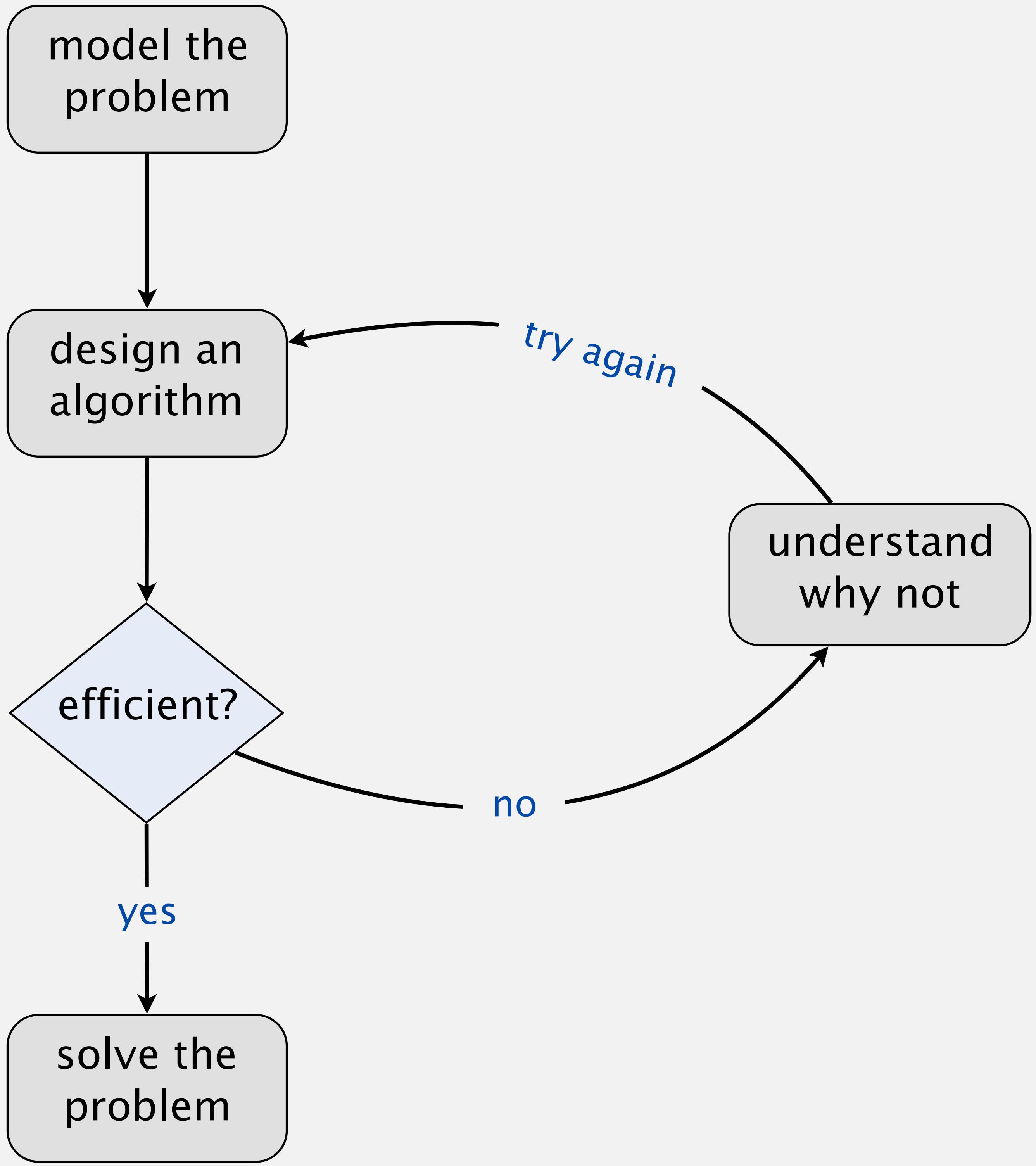
https://algs4.cs.princeton.edu

1.5 UNION–FIND

- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *applications* ← see precept

Subtext of today's lecture (and this course)

Steps to developing a usable algorithm to solve a computational problem.



https://algs4.cs.princeton.edu

1.5 Union–Find

- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *applications*

Union–find data type

Disjoint sets. A collection of sets containing n elements; each element in exactly one set.

Find. Return a "canonical" element in the set containing p ?

Union. Merge the set containing p with the set containing q.

find(1) = find(4) = find(5) = 4

union(2, 5)

`{ 0 } { 1, 4, 5 } { 2, 3, 6, 7 }` → `{ 0 } { 1, 2, 3, 4, 5, 6, 7 }`

8 elements, 3 disjoint sets

2 disjoint sets

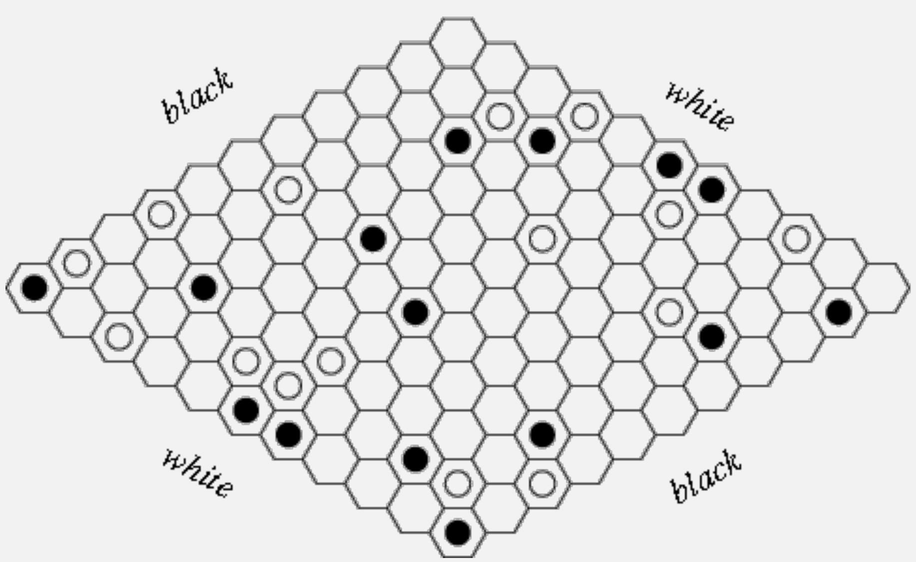
Simplifying assumption. The n elements are named 0, 1, …, $n-1$.

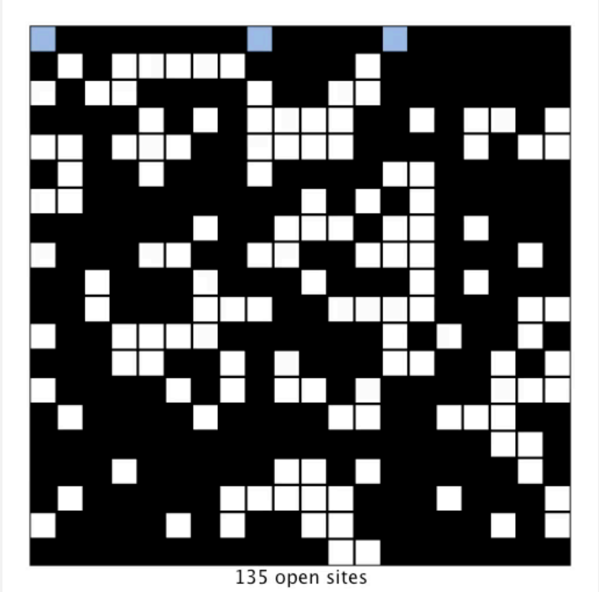
Union-find data type: applications

Disjoint sets can represent:

- Connected components in a graph.
- Interlinked friends in a social network.
- Interconnected devices in a mobile network.
- Equivalent variable names in a Fortran program.
- Clusters of conducting sites in a composite system.
- Contiguous pixels of the same color in a digital image.
- Adjoining stones of the same color in the game of Hex.

see Assignment 1





Union–find data type: API

Goal. Design an efficient union–find data type.

- Number of elements n can be huge.
- Number of operations m can be huge.
- Union and find operations can be intermixed.

```
public class UF
```

	`UF(int n)`	*initialize with n singleton sets* (0 *to* $n-1$)
`void`	`union(int p, int q)`	*merge sets containing elements p and q*
`int`	`find(int p)`	*return canonical element in set containing p*

https://algs4.cs.princeton.edu

1.5 Union–Find

- union–find data type
- *quick-find*
- quick-union
- weighted quick-union
- applications

Quick-find

Data structure.

- Integer array `id[]` of length `n`.
- Interpretation: `id[p]` is canonical element in the set containing `p`.

	0	1	2	3	4	5	6	7	8	9
`id[]`	0	1	1	8	8	0	0	1	8	8

`id[i]` = 0 `id[i]` = 1 `id[i]` = 8

{ 0, 5, 6 } { 1, 2, 7 } { 3, 4, 8, 9 }

3 disjoint sets

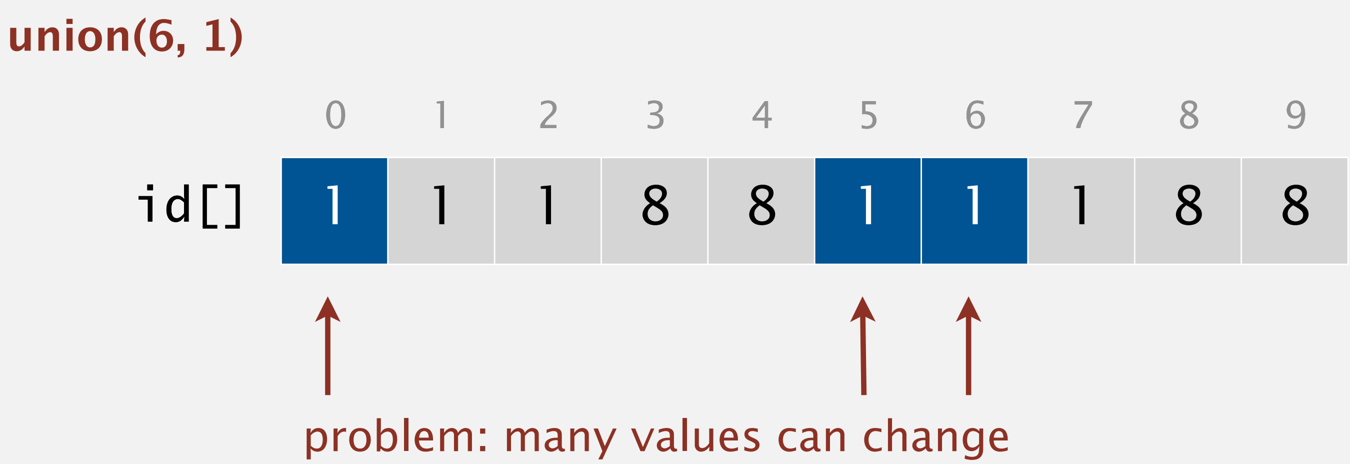
Q. How to implement `find(p)`?

A. Easy, just return `id[p]`.

Quick-find

Data structure.

- Integer array `id[]` of length `n`.
- Interpretation: `id[p]` is canonical element in the set containing `p`.



or vice versa

Q. How to implement `union(p, q)`?

A. Change all entries whose identifier equals `id[p]` to `id[q]`.

Quick-find: Java implementation

```java
public class QuickFindUF
{
   private int[] id;

   public QuickFindUF(int n)
   {
      id = new int[n];
      for (int i = 0; i < n; i++)
         id[i] = i;
   }

   public int find(int p)
   {  return id[p];  }

   public void union(int p, int q)
   {
      int pid = id[p];
      int qid = id[q];
      for (int i = 0; i < id.length; i++)
         if (id[i] == pid) id[i] = qid;
   }
}
```

set id of each element to itself
(n array accesses)

return the id of p
(1 array access)

change all entries with `id[p]` to `id[q]`
($\geq n$ array accesses)

https://algs4.cs.princeton.edu/15uf/QuickFindUF.java.html

Quick-find is too slow

Cost model. Number of array accesses (for read or write).

algorithm	initialize	union	find
quick-find	n	n	1

number of array accesses (ignoring leading constant)

Union is too expensive. Processing a sequence of m union operations on n elements takes $\geq mn$ array accesses.

quadratic!

https://algs4.cs.princeton.edu

1.5 Union–Find

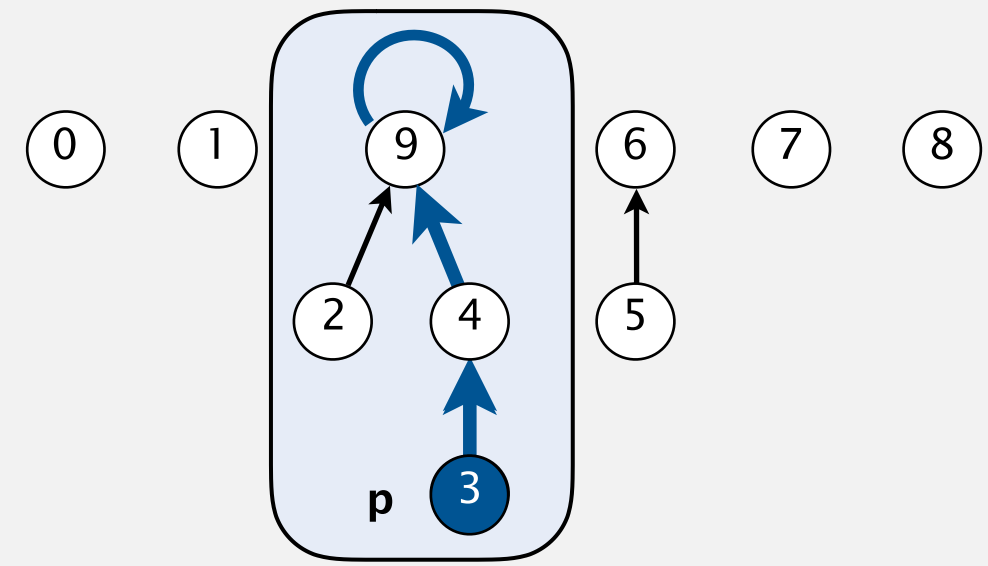
- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *applications*

Quick-union

Data structure: Forest-of-trees.

- Interpretation: elements in one rooted tree correspond to one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of `i` in tree.

	0	1	2	3	4	5	6	7	8	9
parent[]	0	1	9	4	9	6	6	7	8	9



parent of 3 is 4
root of 3 is 9

`find(i) = 9`

`{ 0 } { 1 } { 2, 3, 4, 9 } { 5, 6 } { 7 } { 8 }`

6 disjoint sets (6 trees)

Q. How to implement `find(p)` operation?

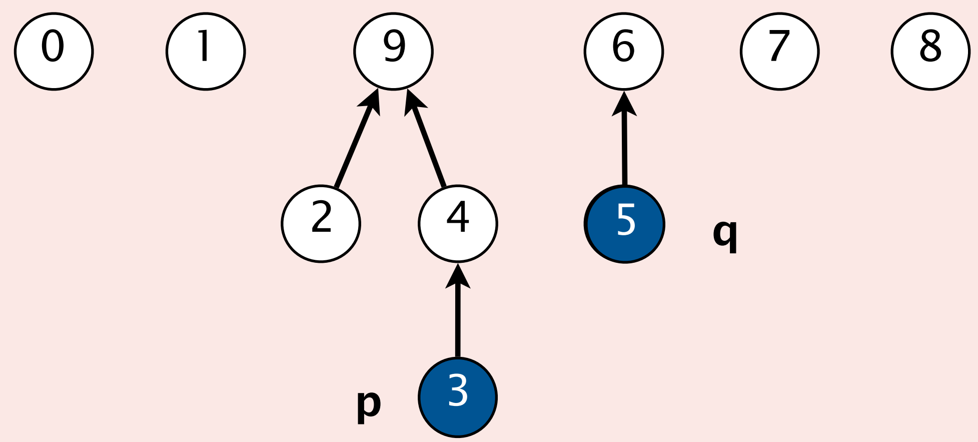
A. Use tree root as canonical element ⇒ return root of tree containing `p`.

Union-find quiz 1

Data structure: Forest-of-trees.

- Interpretation: elements in one rooted tree correspond to one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of `i` in tree.

	0	1	2	3	4	5	6	7	8	9
parent[]	0	1	9	4	9	6	6	7	8	9



Which is not a valid way to implement `union(3, 5)` ?

A. Set `parent[6]` = 9.

B. Set `parent[9]` = 6.

C. Set `parent[3]` = `parent[4]` = `parent[9]` = 6.

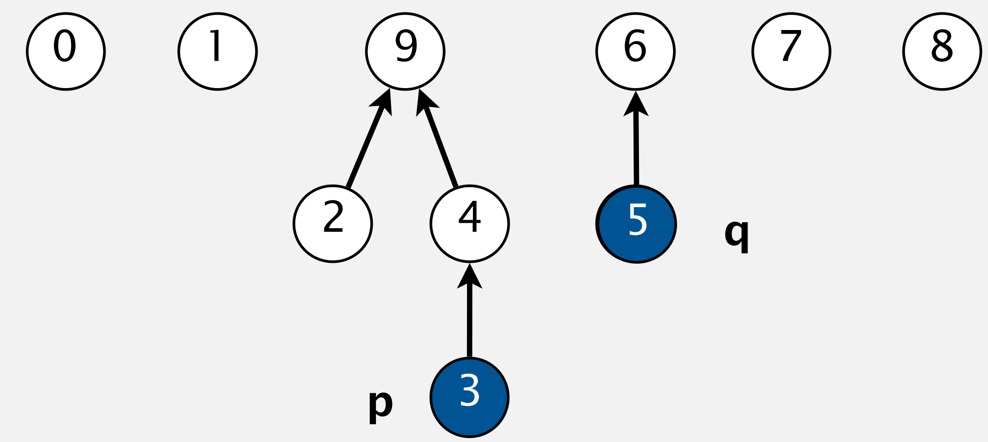
D. Set `parent[3]` = 5.

Quick-union

Data structure: Forest-of-trees.

- Interpretation: elements in one rooted tree correspond to one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of `i` in tree.

	0	1	2	3	4	5	6	7	8	9
union(3, 5)	0	1	9	4	9	6	6	7	8	9



Q. How to implement `union(p, q)`?

A. Set parent of `p`'s root to `q`'s root. ← or vice versa

Quick-union

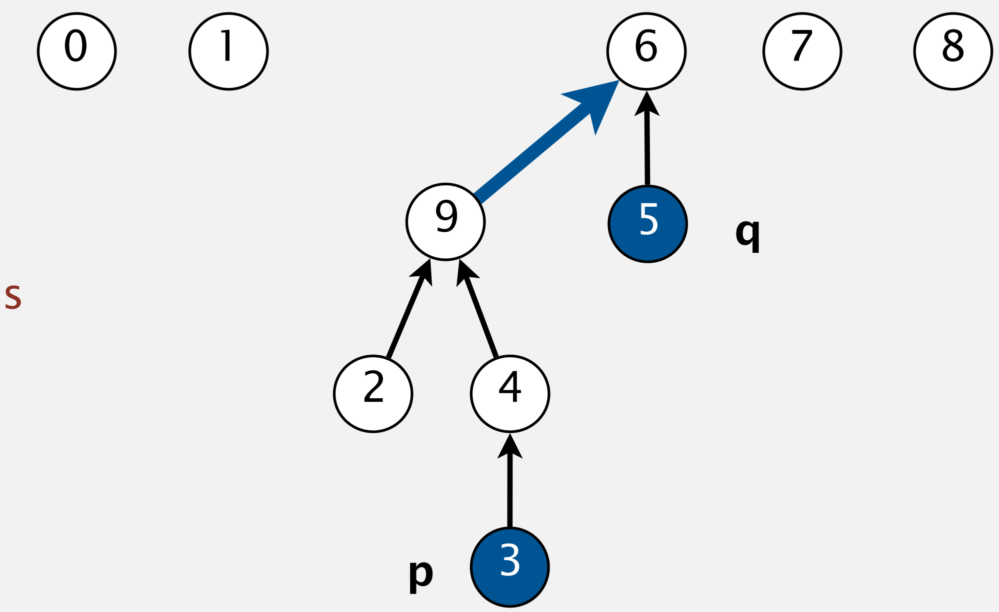
Data structure: Forest-of-trees.

- Interpretation: elements in one rooted tree correspond to one set.
- Integer array `parent[]` of length `n`, where `parent[i]` is parent of `i` in tree.

union(3, 5)

	0	1	2	3	4	5	6	7	8	9
union(3, 5)	0	1	9	4	9	6	6	7	8	6

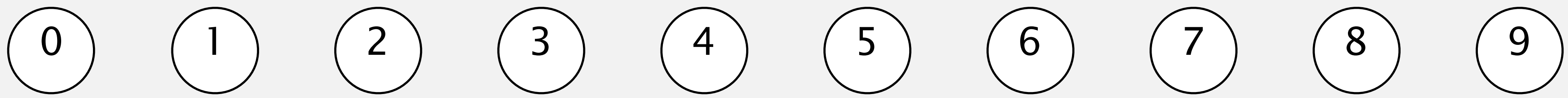
only one value changes



Q. How to implement `union(p, q)`?

A. Set parent of `p`'s root to `q`'s root. ← or vice versa

Quick-union demo



0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

Quick-union: Java implementation

```
public class QuickUnionUF
{
   private int[] parent;

   public QuickUnionUF(int n)
   {
      parent = new int[n];
      for (int i = 0; i < n; i++)
          parent[i] = i;
   }

   public int find(int p)
   {
      while (p != parent[p])
          p = parent[p];
      return p;
   }

   public void union(int p, int q)
   {
      int r1 = find(p);
      int r2 = find(q);
      parent[r1] = r2;
   }
}
```

set parent of each element to itself (to create forest of n singleton trees)

follow parent pointers until reach root

link root of p to root of q

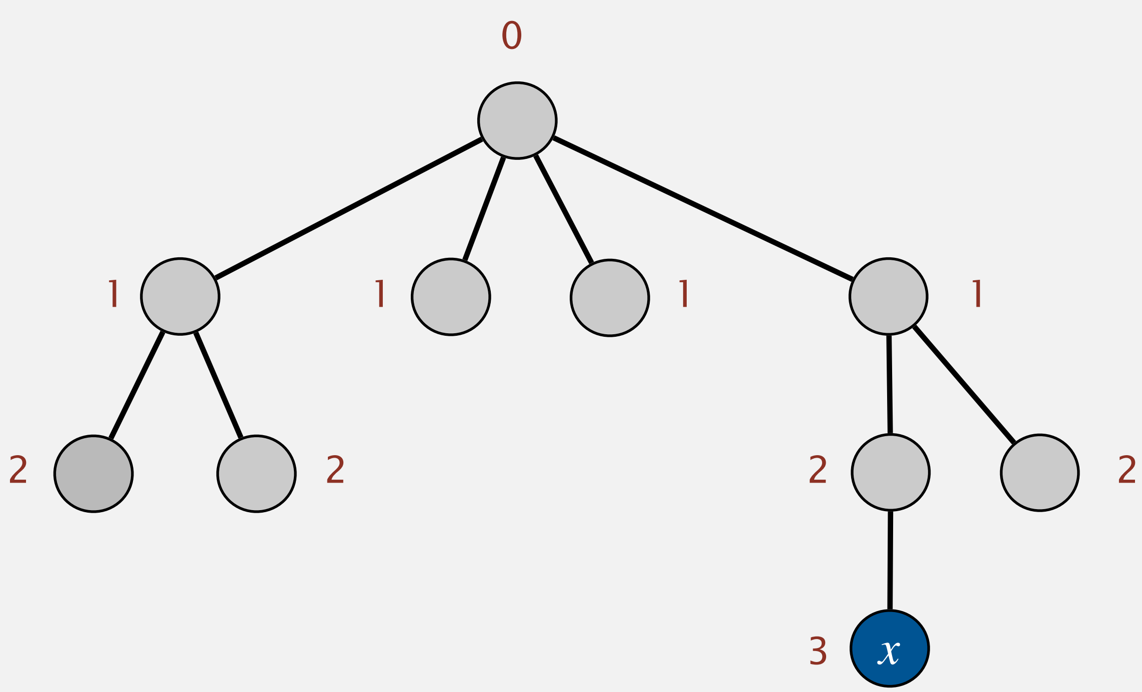
https://algs4.cs.princeton.edu/15uf/QuickUnionUF.java.html

Quick-union analysis

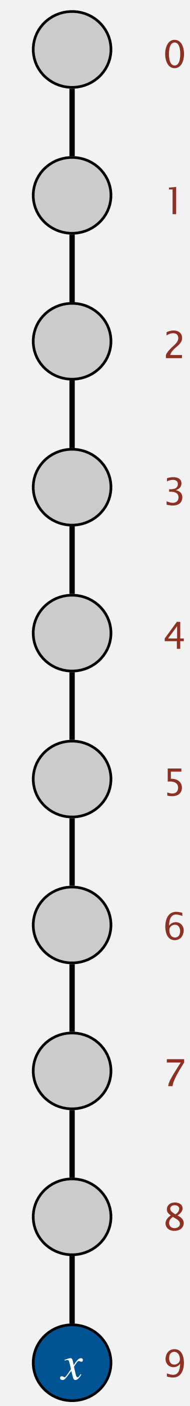
Cost model. Number of array accesses (for read or write).

Running time.

- Union: takes constant time, given two roots.
- Find: takes time proportional to depth of node in tree.



depth(x) = 3



worst-case depth = n–1

Quick-union analysis

Cost model. Number of array accesses (for read or write).

Running time.

- Union: takes constant time, given two roots.
- Find: takes time proportional to depth of node in tree.

algorithm	initialize	union	find
quick-find	n	n	1
quick-union	n	n	n

worst-case number of array accesses (ignoring leading constant)

Too expensive (if trees get tall). Processing some sequences of m union and find operations on n elements takes $\geq mn$ array accesses.

https://algs4.cs.princeton.edu

1.5 UNION–FIND

- *union–find data type*
- *quick-find*
- *quick-union*
- ***weighted quick-union***
- *applications*

Union-find quiz 2

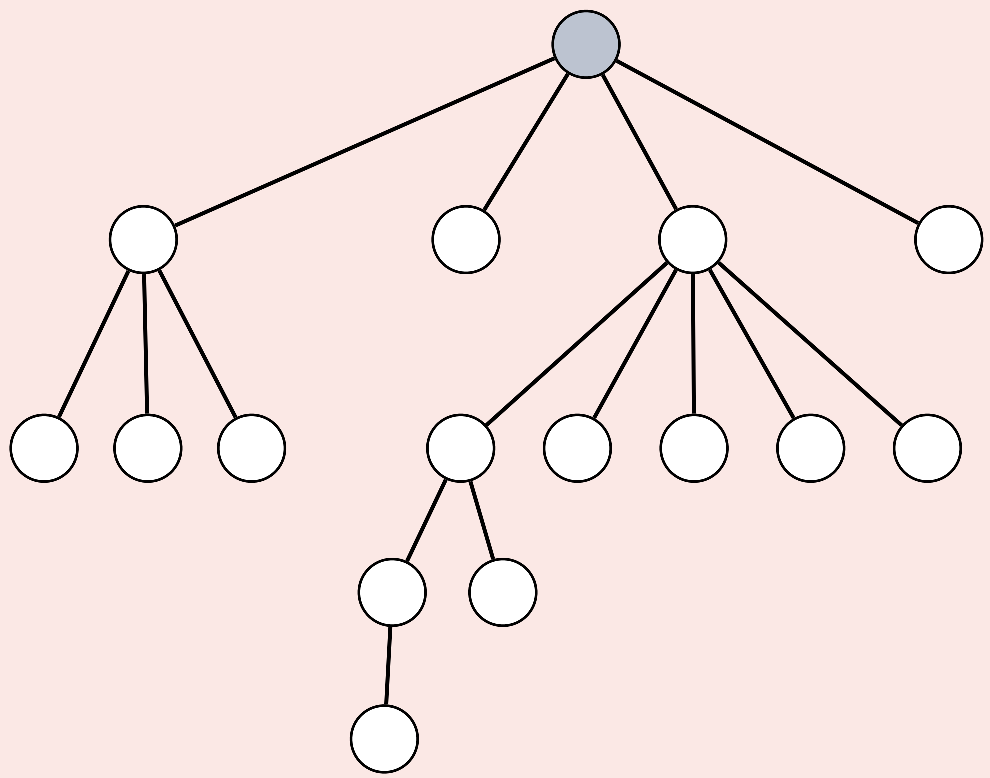
When linking two trees, which strategy is most effective?

A. Link the root of the *smaller* tree to the root of the *larger* tree.

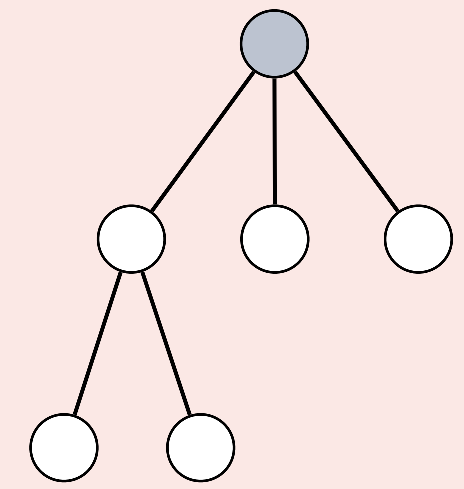
B. Link the root of the *larger* tree to the root of the *smaller* tree.

C. Flip a coin; randomly choose between A and B.

D. Doesn't matter.



larger tree
(size = 16, height = 4)

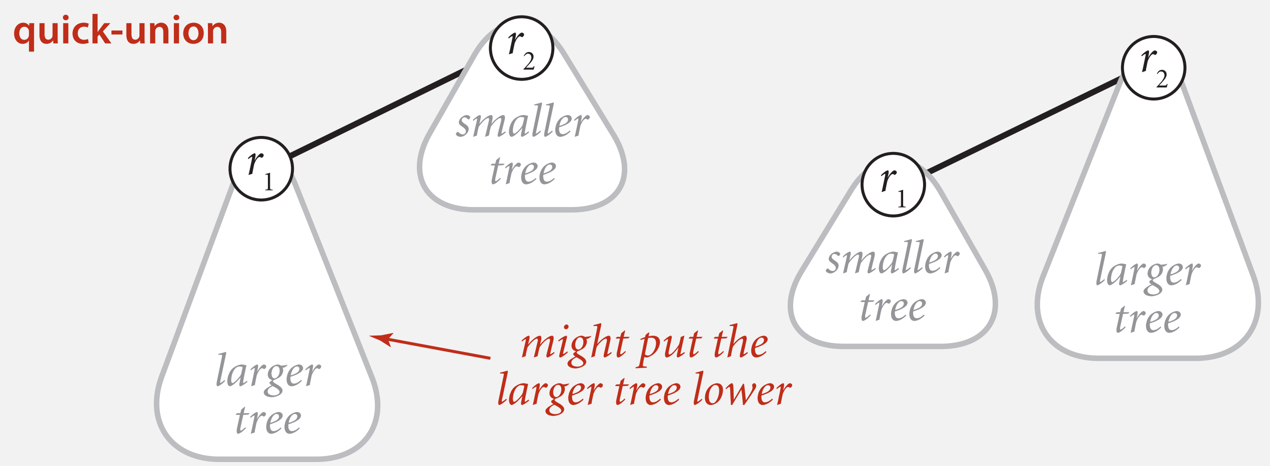


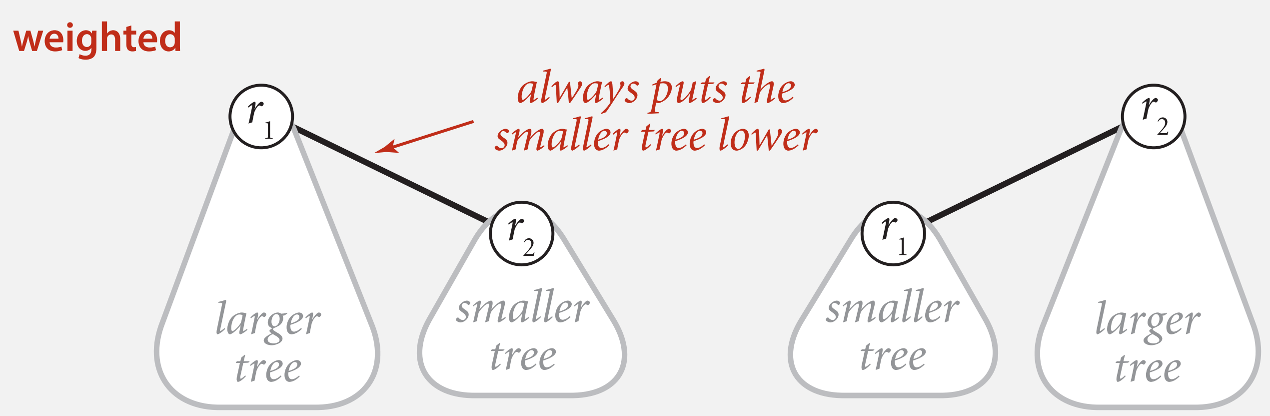
smaller tree
(size = 6, height = 2)

Weighted quick-union (link-by-size)

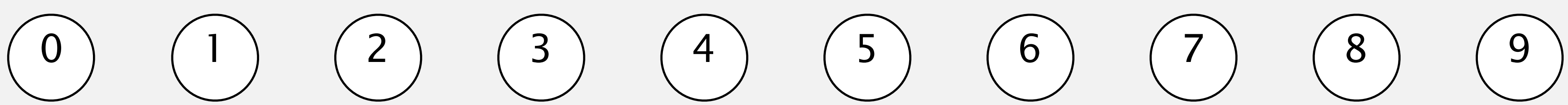
- Modify quick-union to avoid tall trees.
- Keep track of size of each tree = number of elements.
- Always link root of smaller tree to root of larger tree.

reasonable alternative:
link-by-height





Weighted quick-union demo



	0	1	2	3	4	5	6	7	8	9
parent[]	0	1	2	3	4	5	6	7	8	9

Weighted quick-union: Java implementation

Data structure. Same as quick-union, but maintain extra array `size[i]` to count number of elements in the tree rooted at `i`, initially `1`.

- Find: identical to quick-union.
- Union: link root of smaller tree to root of larger tree; update `size[]`.

```
public void union(int p, int q)
{
   int r1 = find(p);
   int r2 = find(q);
   if (r1 == r2) return;

   if (size[r1] >= size[r2])
   { int temp = r1; r1 = r2; r2 = temp; }

   parent[r1] = r2;
   size[r2] += size[r1];

}
```

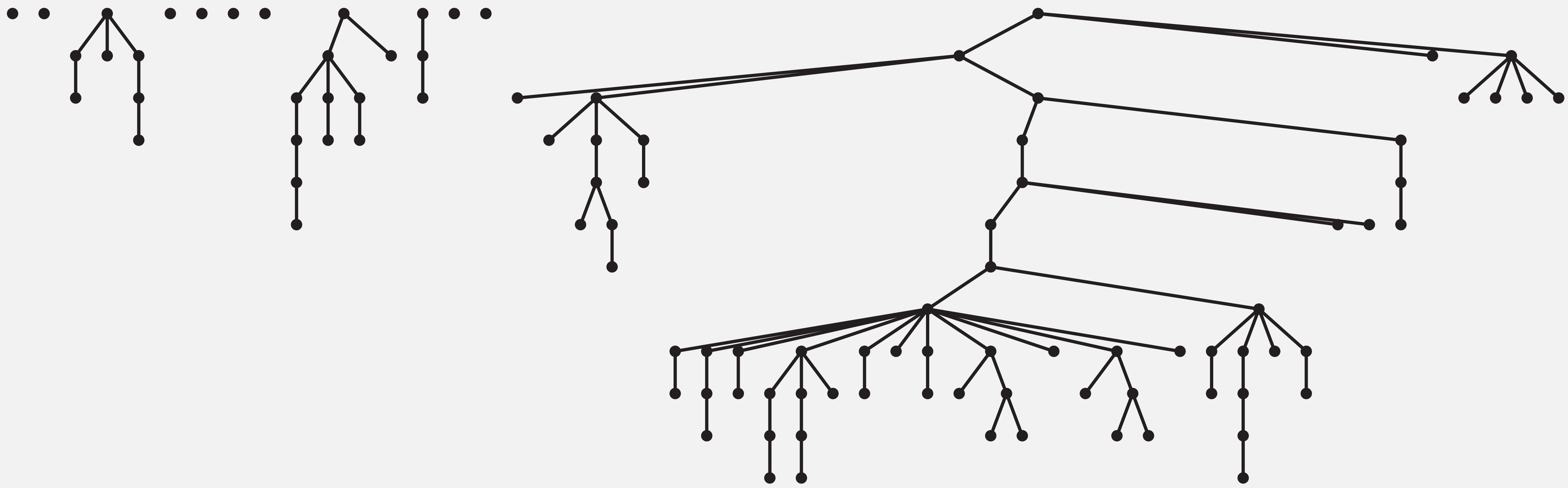
afterwards, `r1` is root of smaller tree

link root of smaller tree to root of larger tree

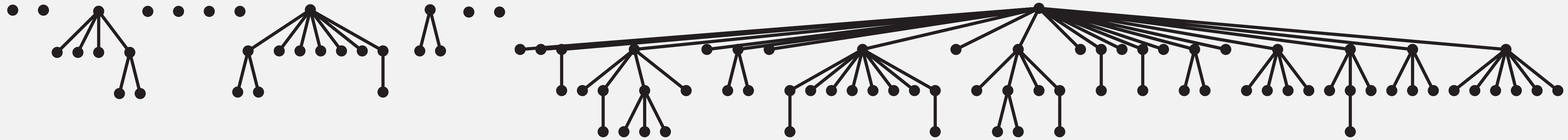
https://algs4.cs.princeton.edu/15uf/WeightedQuickUnionUF.java.html

Quick-union vs. weighted quick-union: larger example

quick-union

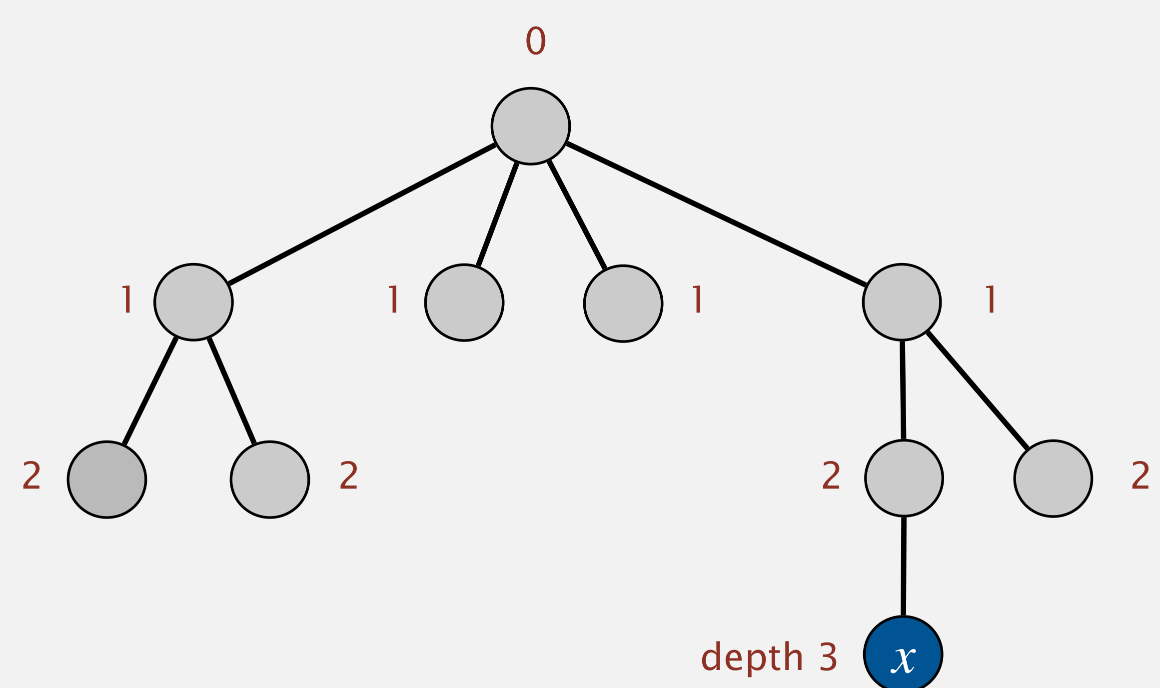


weighted



Weighted quick-union analysis

Proposition. Depth of any node x in tree is at most $\log_2 n$.



n = 10
depth(x) = 3 ≤ $\log_2$ n

Weighted quick-union analysis

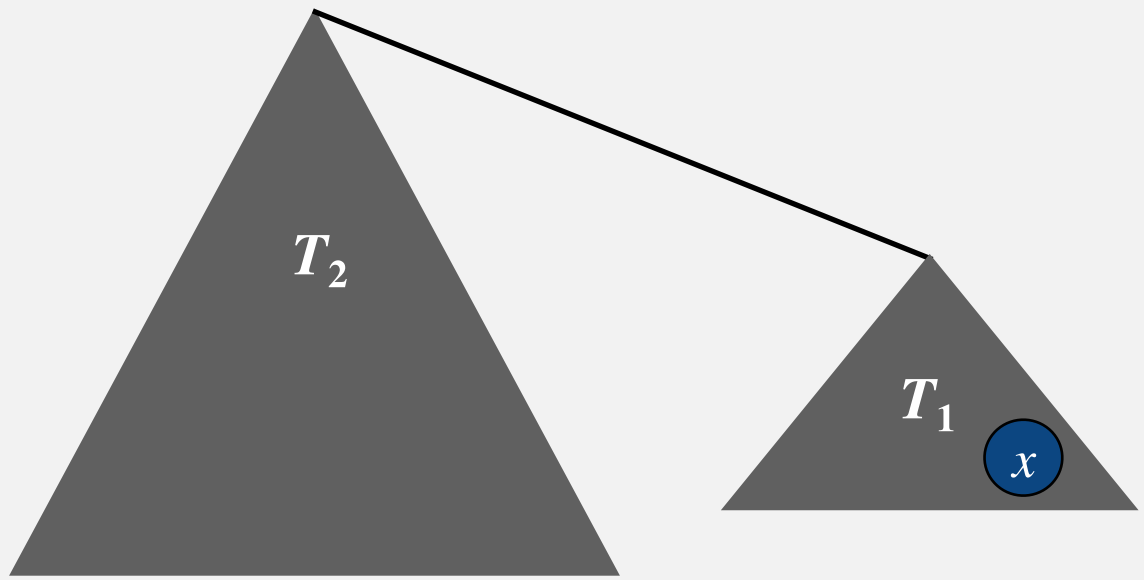
Proposition. Depth of any node x in tree is at most $\log_2 n$.

Pf.

- Depth of x does not change unless root of tree T_1 containing x is linked to root of a larger tree T_2, forming new tree T_3.
- In this case:
  - depth of x increases by exactly 1
  - size of tree containing x at least doubles because $\text{size}(T_3) = \text{size}(T_1) + \text{size}(T_2) \geq 2 \times \text{size}(T_1)$.

can happen at most $\log_2 n$ times. Why?

$$\underbrace{1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow \cdots \rightarrow n}_{\log_2 n}$$



Weighted quick-union analysis

Proposition. Depth of any node x in tree is at most $\log_2 n$.

Running time.

- Union: takes constant time, given two roots.
- Find: takes time proportional to depth of node in tree.

algorithm	initialize	union	find
quick-find	n	n	1
quick-union	n	n	n
weighted quick-union	n	$\boxed{\log n}$	$\boxed{\log n}$

worst-case number of array accesses (ignoring leading constant)

log mean logarithm,
for some constant base

Summary

Key point. Weighted quick-union makes it possible to solve problems that could not otherwise be addressed.

algorithm	worst-case time
quick-find	$m\,n$
quick-union	$m\,n$
weighted quick-union	$m \log n$
QU + path compression	$m \log n$
weighted QU + path compression	$m\,\alpha(n)$

order of growth for m ≥ n union-find operations on a set of n elements

fastest for percolation?

inverse Ackermann function (ask Tarjan!)

Ex. [10^9 unions and finds with 10^9 elements]

- Weighted quick-union reduces run time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.

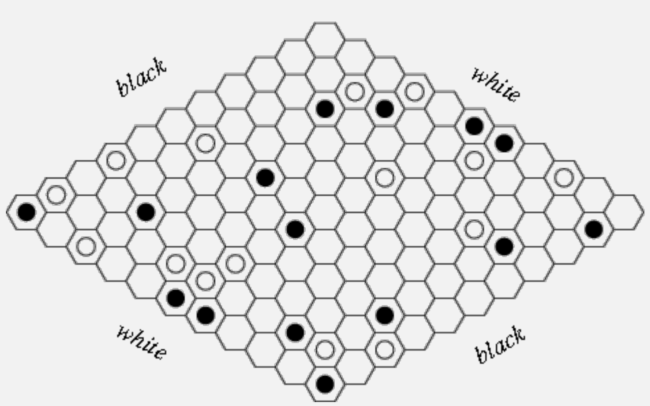
https://algs4.cs.princeton.edu

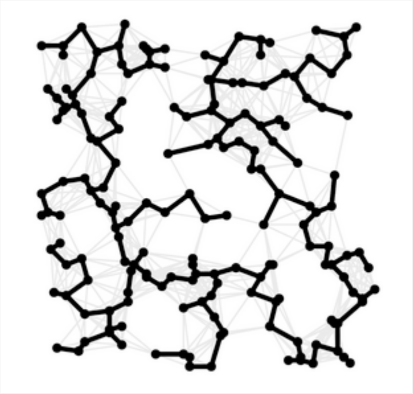
1.5 Union–Find

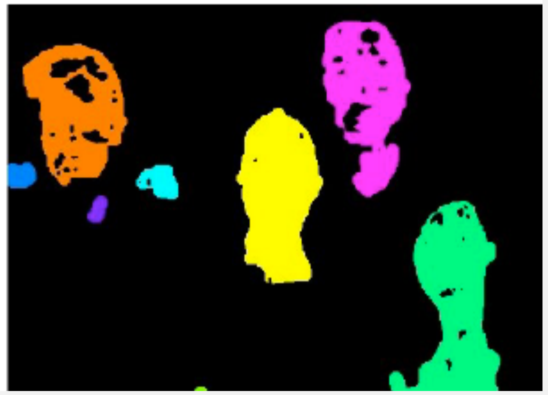
- *union–find data type*
- *quick-find*
- *quick-union*
- *weighted quick-union*
- *applications*

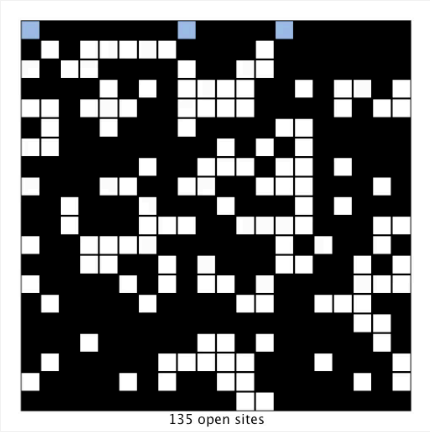
Union-find applications

- Percolation. ⟵ first programming assignment
- Terrain analysis.
- Dynamic-connectivity problem. ⟵ see textbook
- Least common ancestors in trees.
- Games (Go, Hex, maze generation).
- Minimum spanning tree algorithms.
- Equivalence of finite state automata.
- Hoshen–Kopelman algorithm in physics.
- Hindley–Milner polymorphic type inference.
- Compiling equivalence statements in Fortran.
- Matlab's `bwlabel()` function in image processing.







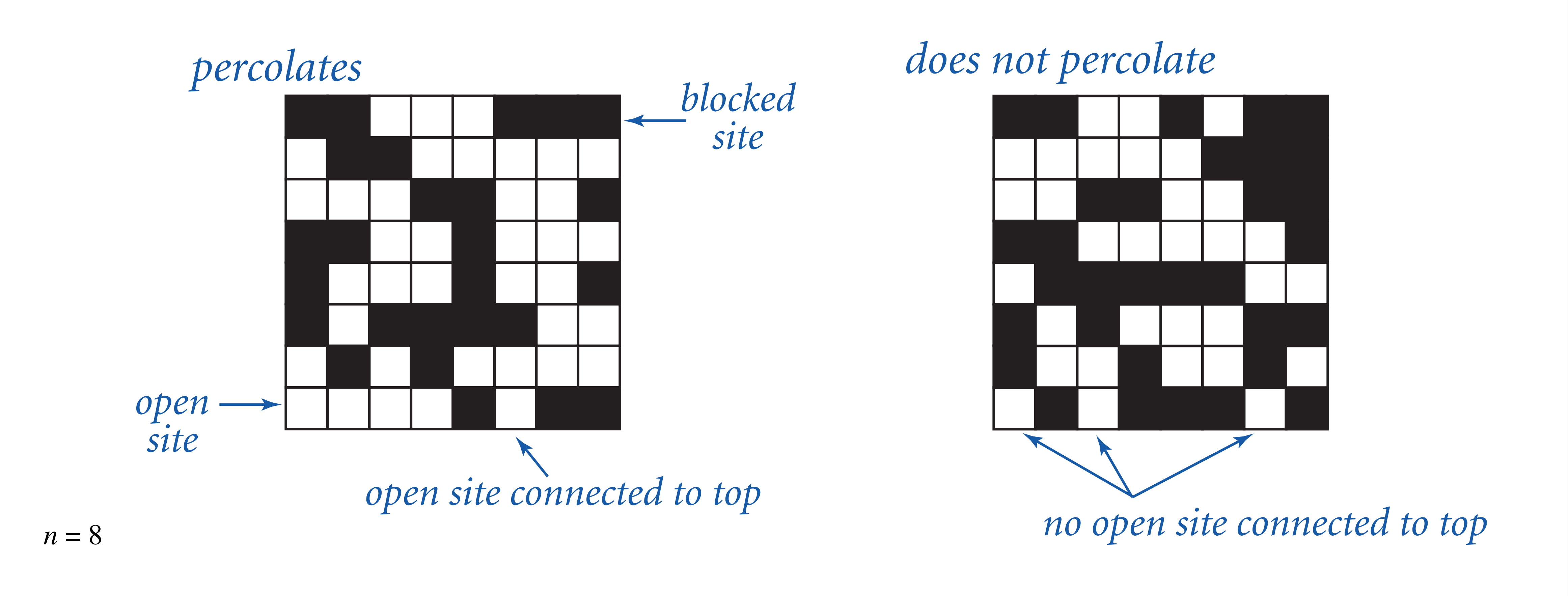


Percolation

An abstract model for many physical systems:

- n-by-n grid of sites.
- Each site is open with probability p (and blocked with probability $1 - p$).
- System percolates iff top and bottom are connected by open sites.

if and only if



Percolation

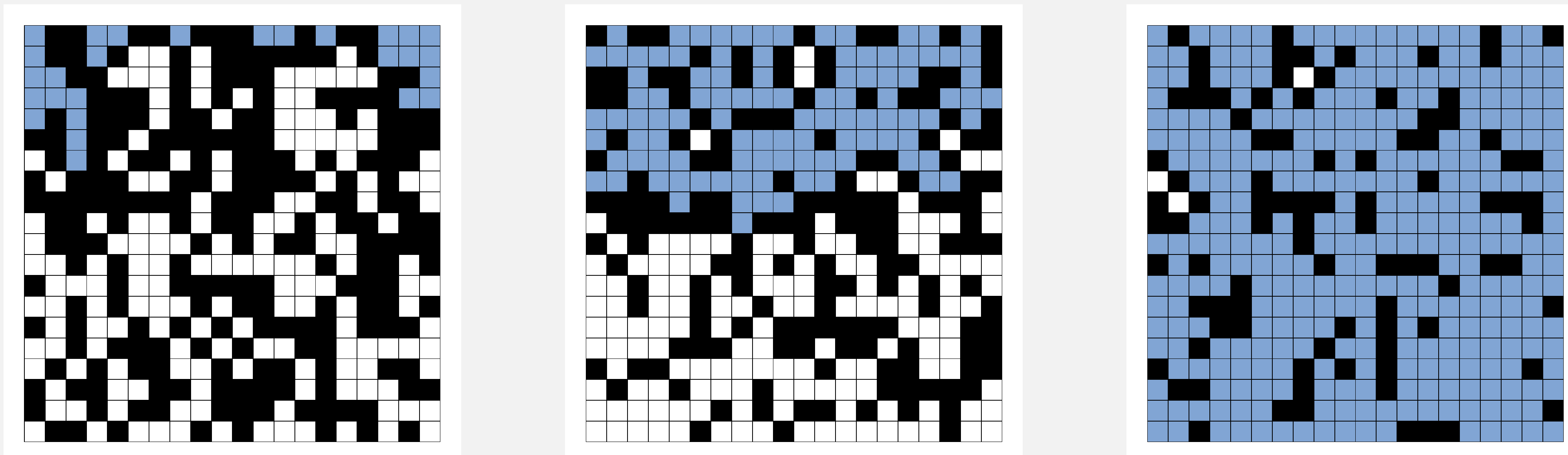
An abstract model for many physical systems:

- n-by-n grid of sites.
- Each site is open with probability p (and blocked with probability $1 - p$).
- System **percolates** iff top and bottom are connected by open sites.

model	system	vacant site	occupied site	percolates
electricity	material	conductor	insulated	conducts
fluid flow	material	empty	blocked	porous
social interaction	population	person	empty	communicates

Likelihood of percolation

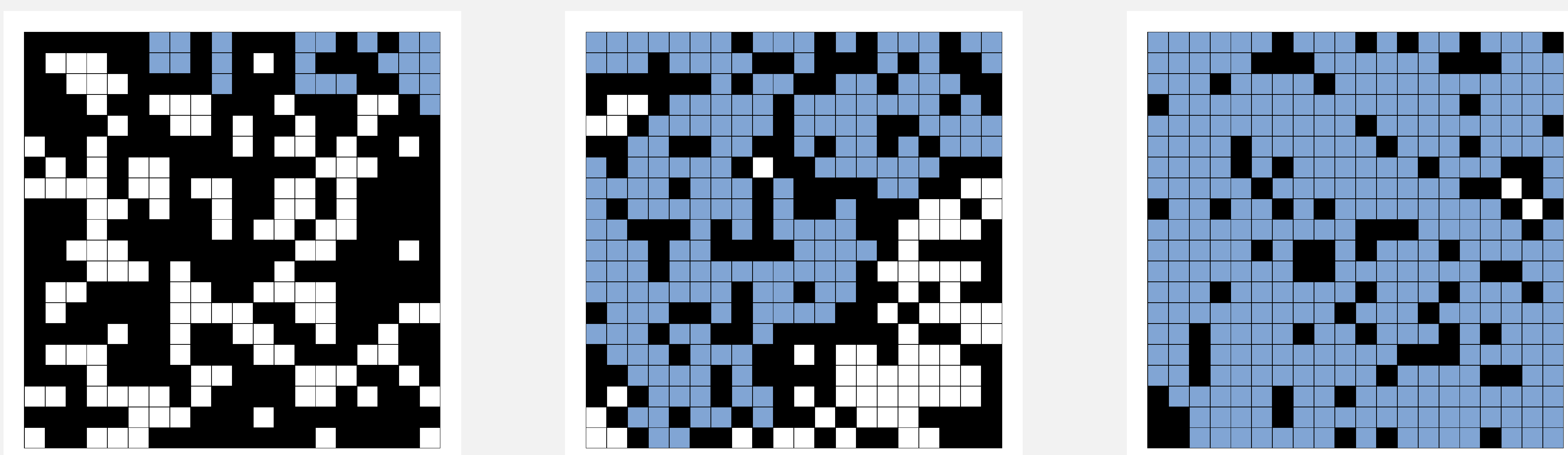
Depends on grid size n and site vacancy probability p.



p low (0.4)
does not percolate

p medium (0.6)
percolates?

p high (0.8)
percolates



empty open site (not connected to top)

full open site (connected to top)

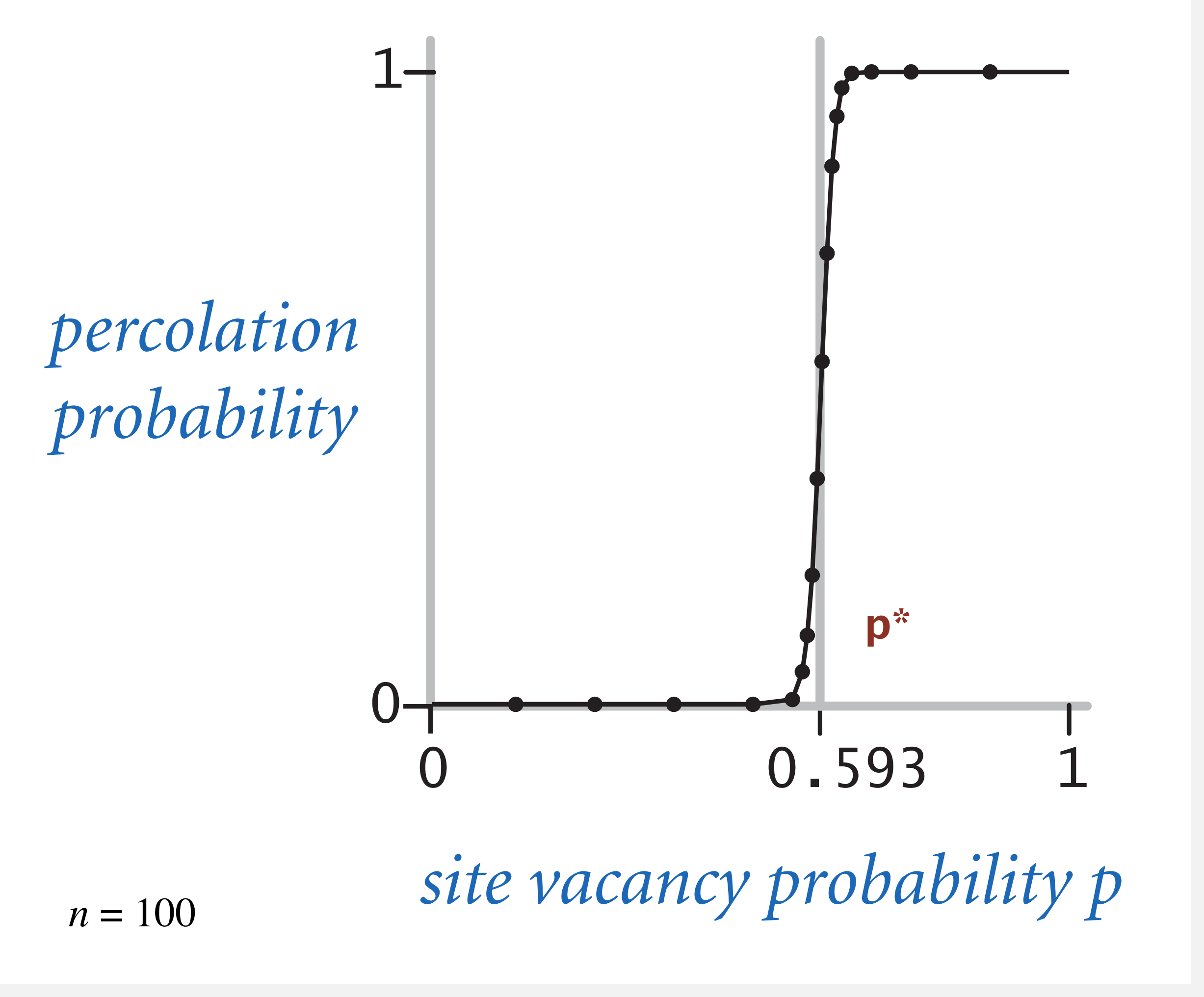
blocked site

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^* ?



Monte Carlo simulation

Barrier. Determining the exact threshold p^* is beyond mathematical reach.

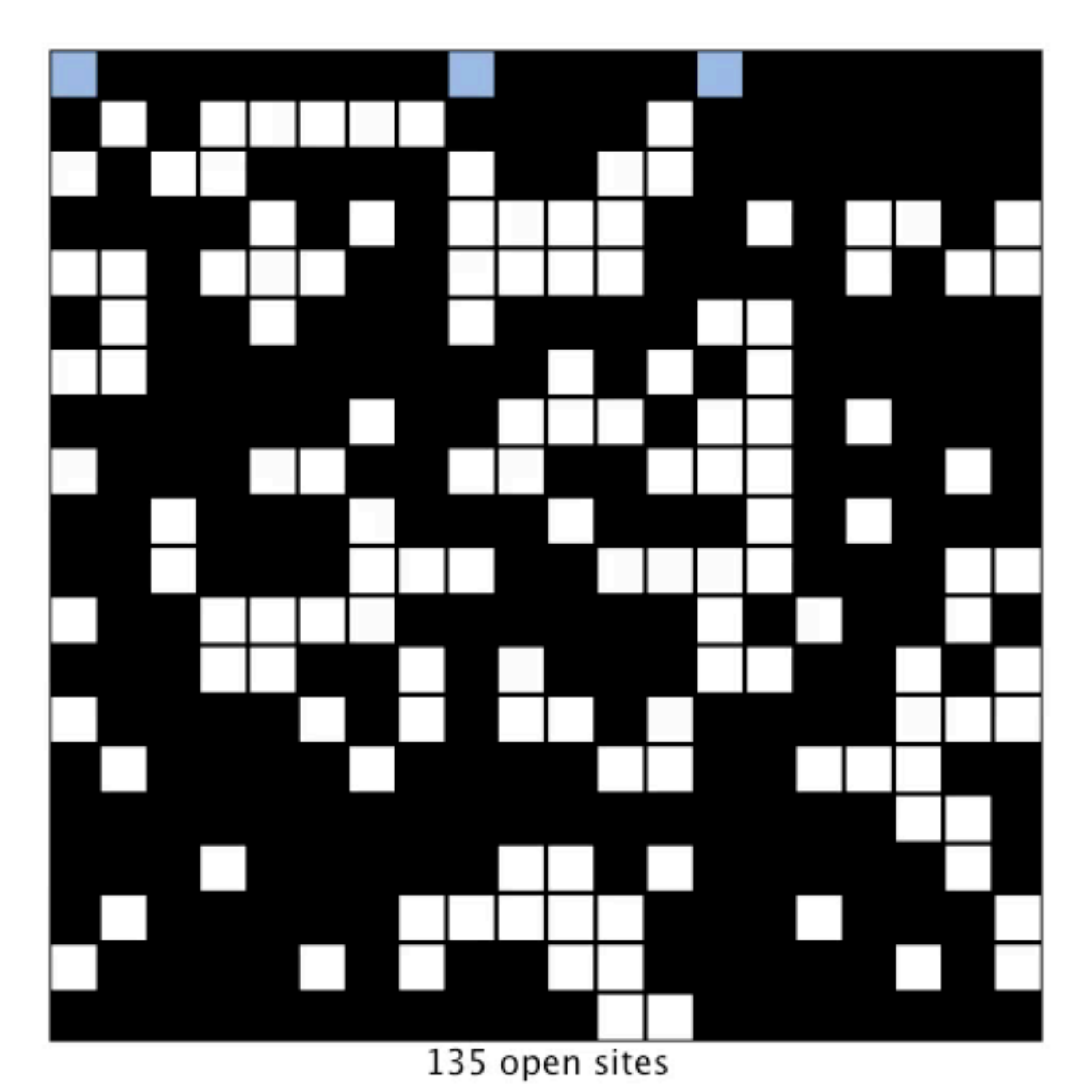
Computational approach.

- Conduct many random experiments.
- Compute statistics.
- Obtain estimate of p^*.

Casino de Monte-Carlo

Monte Carlo simulation

- Initialize all sites in an n-by-n grid to be blocked.
- Declare random sites open until top connected to bottom.
- Vacancy percentage estimates p^*.
- Repeat many times to get more accurate estimate.



full open site (connected to top)

empty open site (not connected to top)

blocked site

$$\hat{p} = \frac{204}{400} = 0.51$$

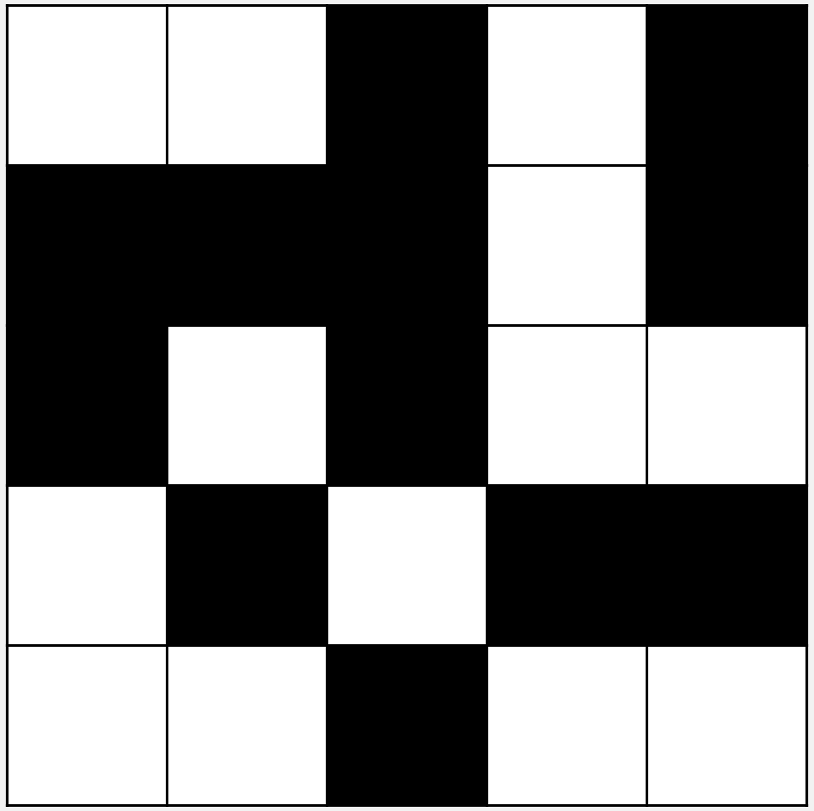
$n = 20$

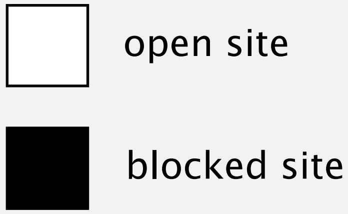
Dynamic-connectivity solution to estimate percolation threshold

Q. How to check whether an n-by-n system percolates?

A. Model as a dynamic-connectivity problem problem and use union–find.

$n = 5$

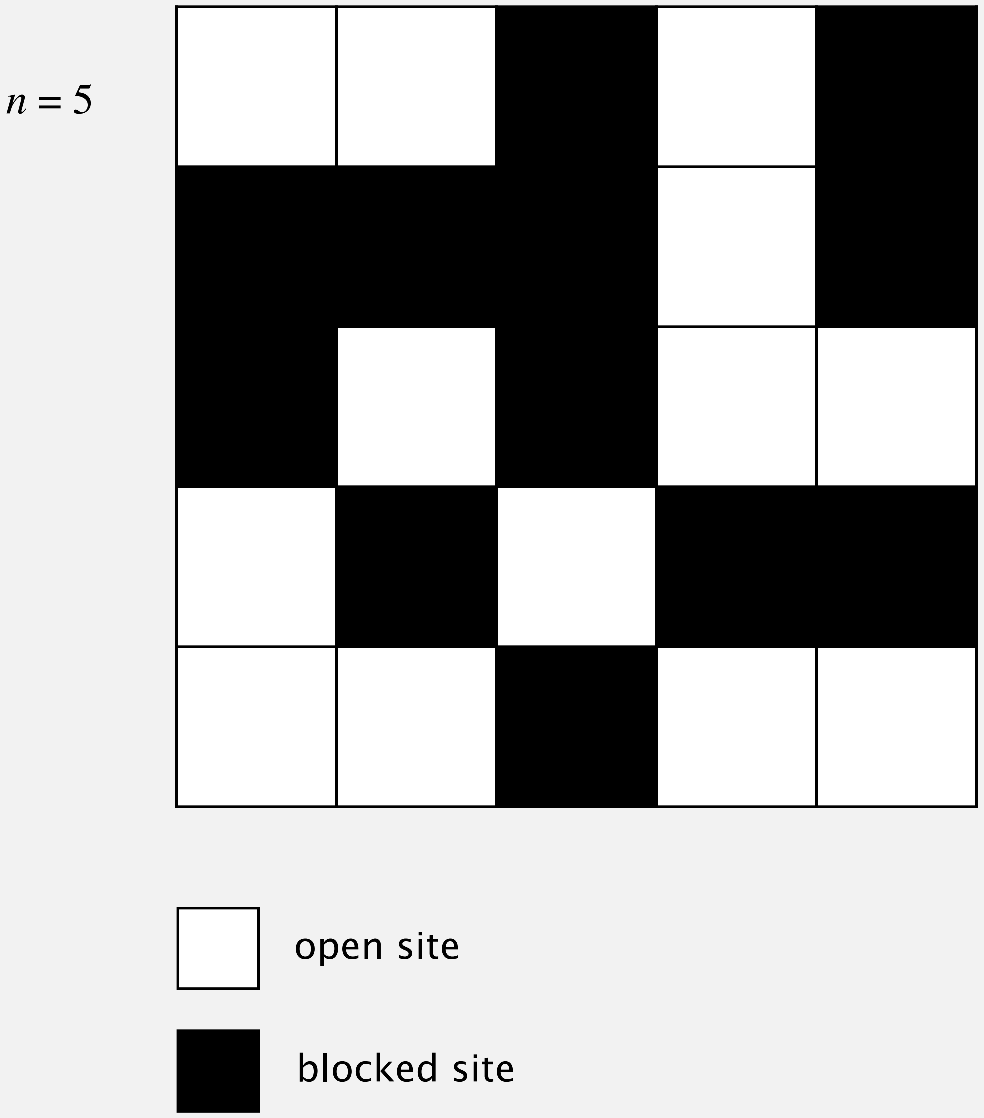


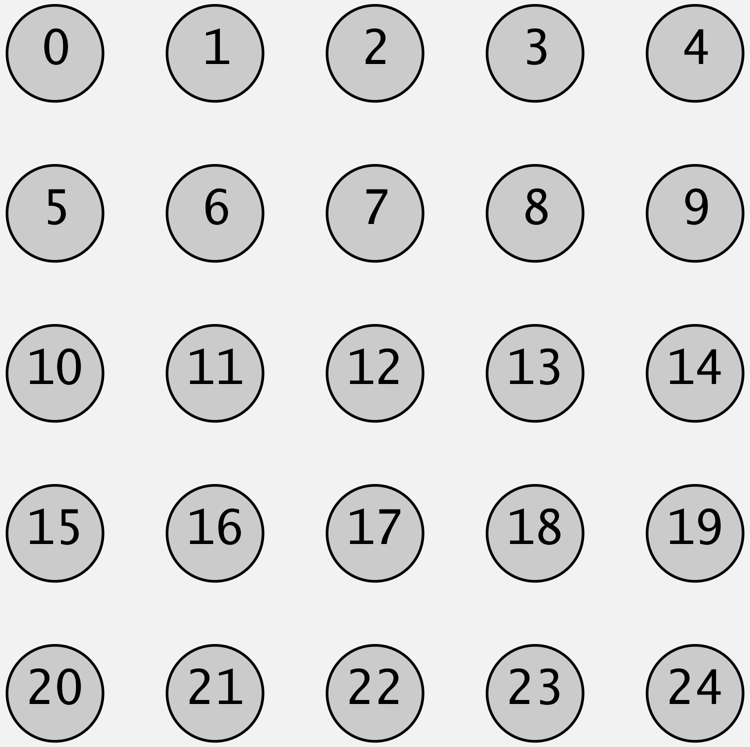


Dynamic-connectivity solution to estimate percolation threshold

Q. How to check whether an n-by-n system percolates?

- Create an element for each site, named 0 to $n^2 - 1$.



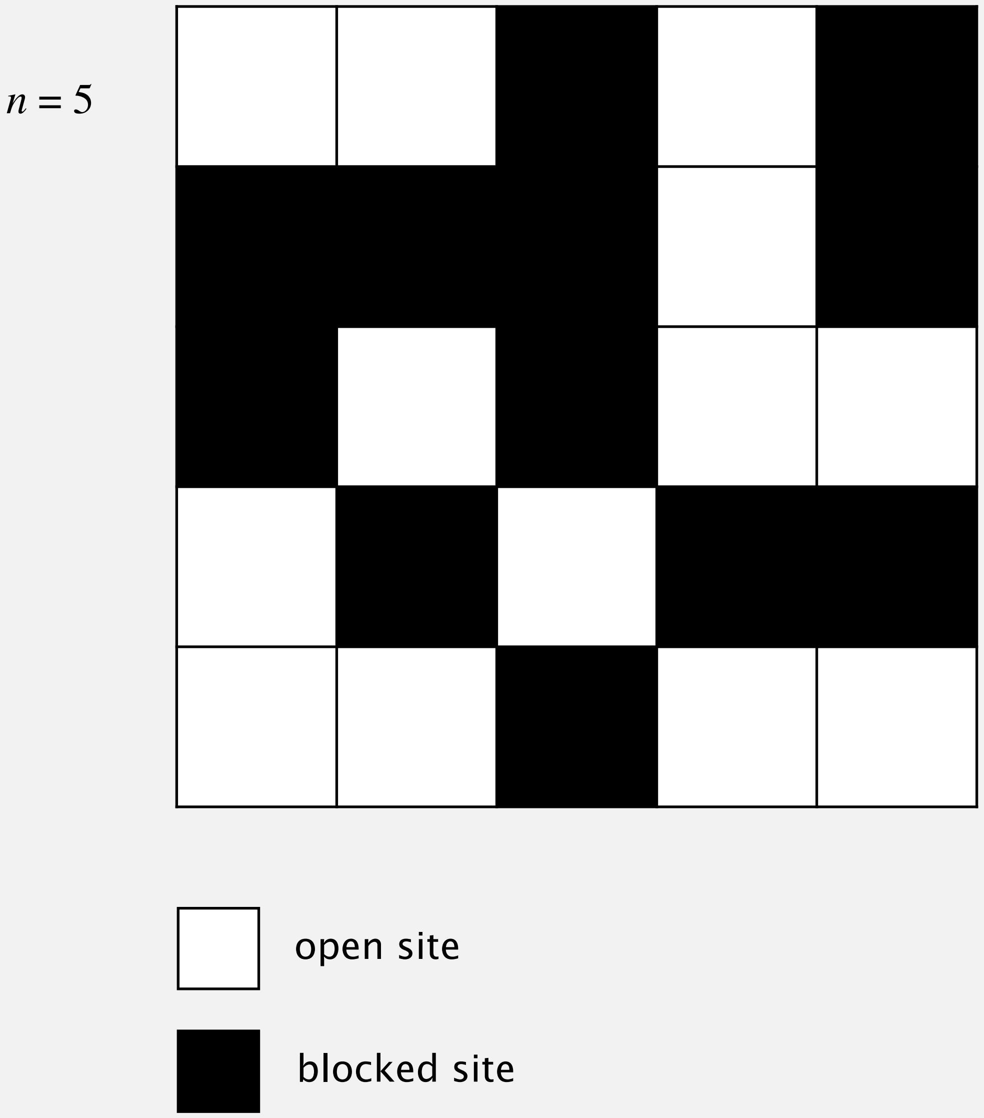


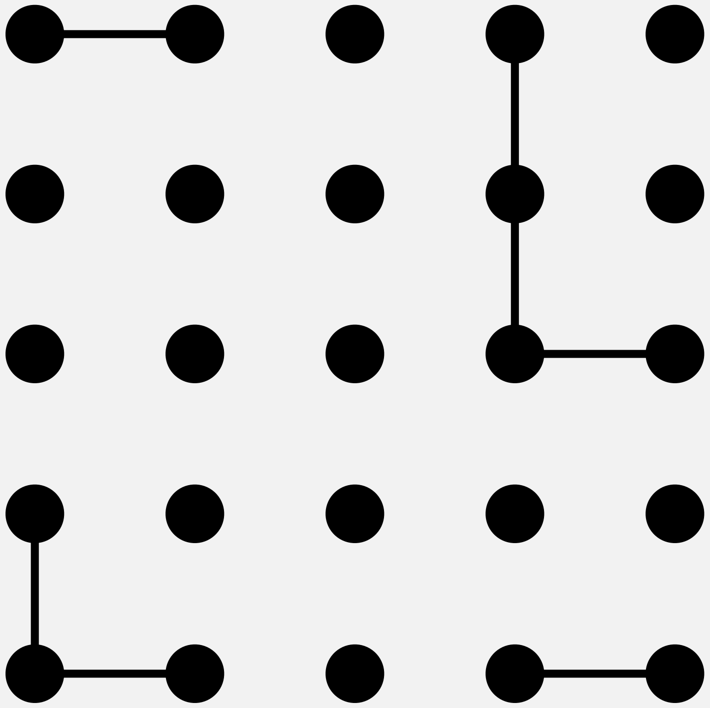
Dynamic-connectivity solution to estimate percolation threshold

Q. How to check whether an n-by-n system percolates?

- Create an element for each site, named 0 to $n^2 - 1$.
- Add edge between two adjacent sites if they both open.

4 possible neighbors: left, right, top, bottom





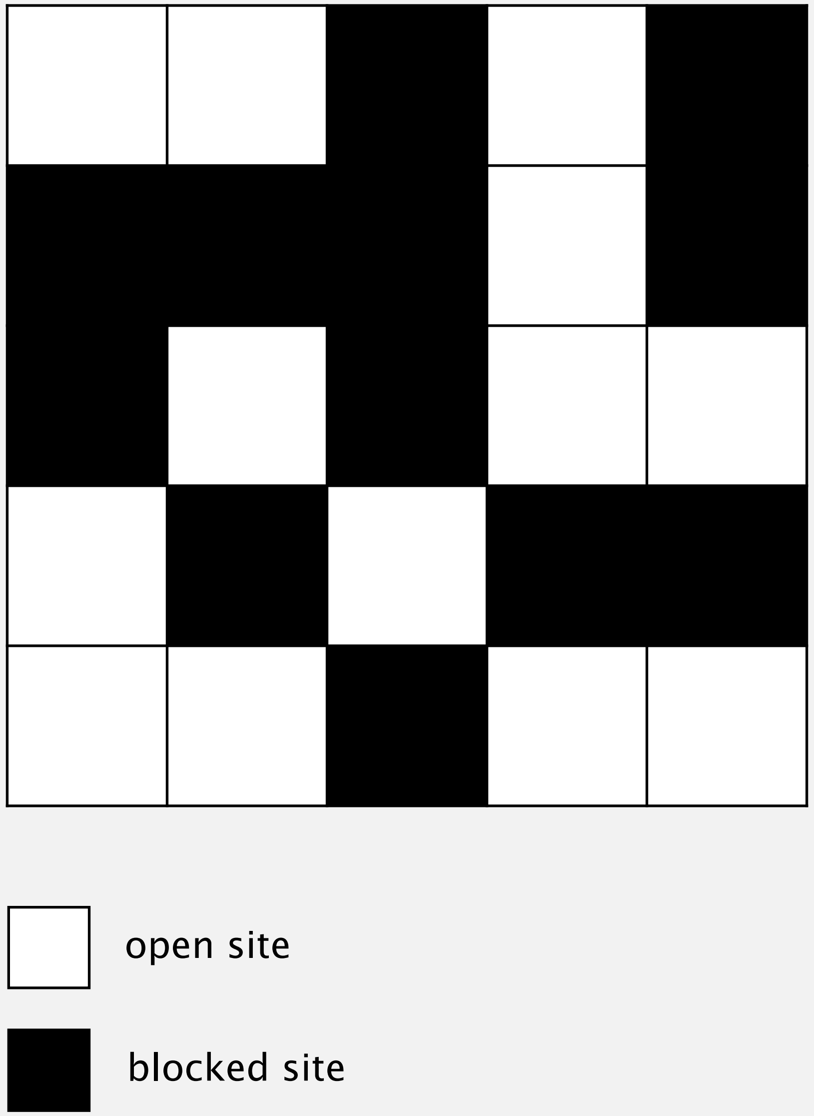
Dynamic-connectivity solution to estimate percolation threshold

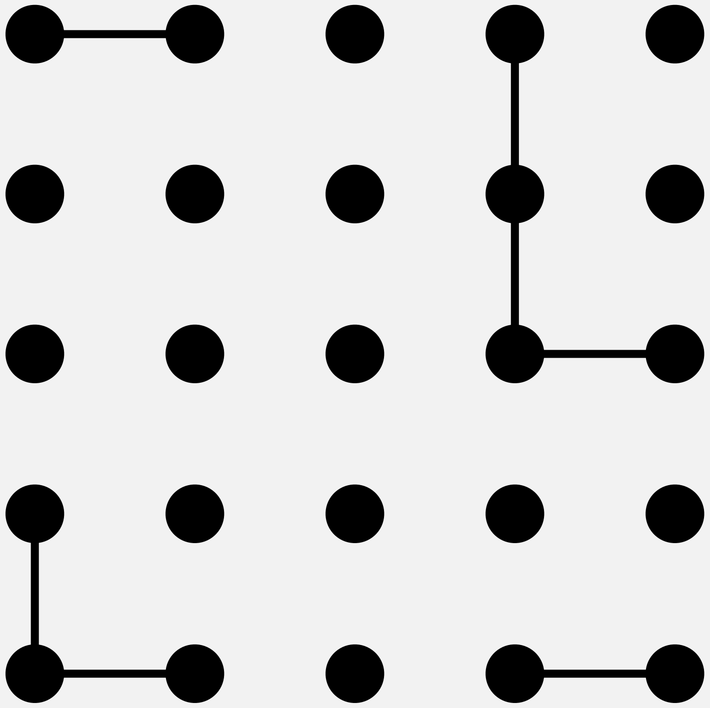
Q. How to check whether an n-by-n system percolates?

- Create an element for each site, named 0 to $n^2 - 1$.
- Add edge between two adjacent sites if they both open.
- Percolates iff any site on bottom row is connected to any site on top row.

brute-force algorithm: n^2 connected queries

$n = 5$





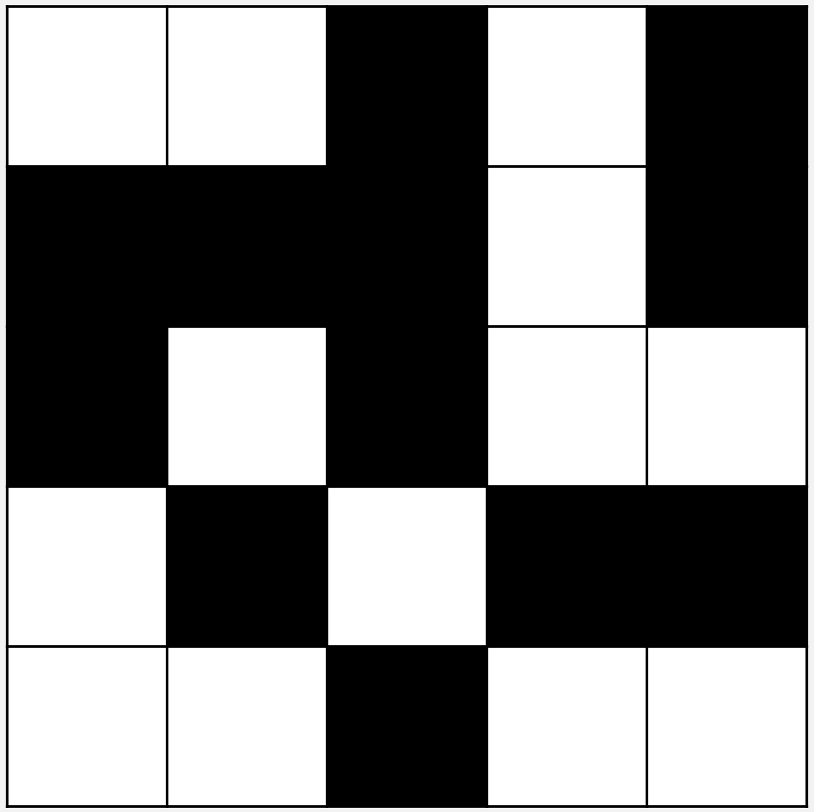
Dynamic-connectivity solution to estimate percolation threshold

Clever trick. Introduce 2 virtual sites (and edges to top and bottom).

- Percolates iff virtual top site is connected to virtual bottom site.

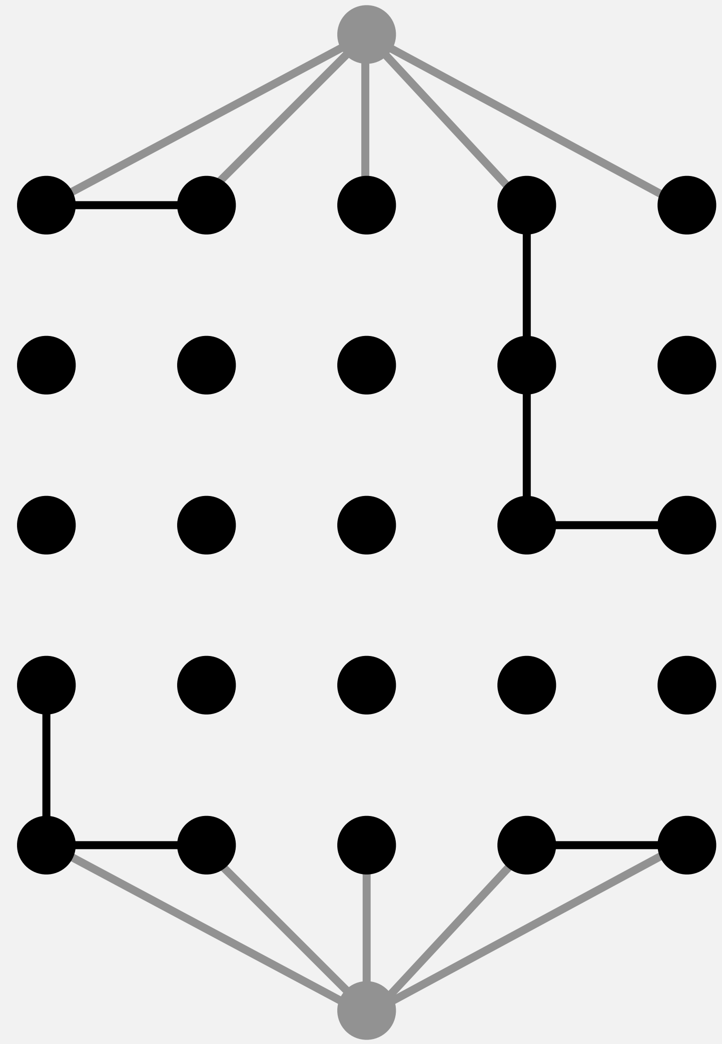
more efficient algorithm: only 1 connected query

$n = 5$



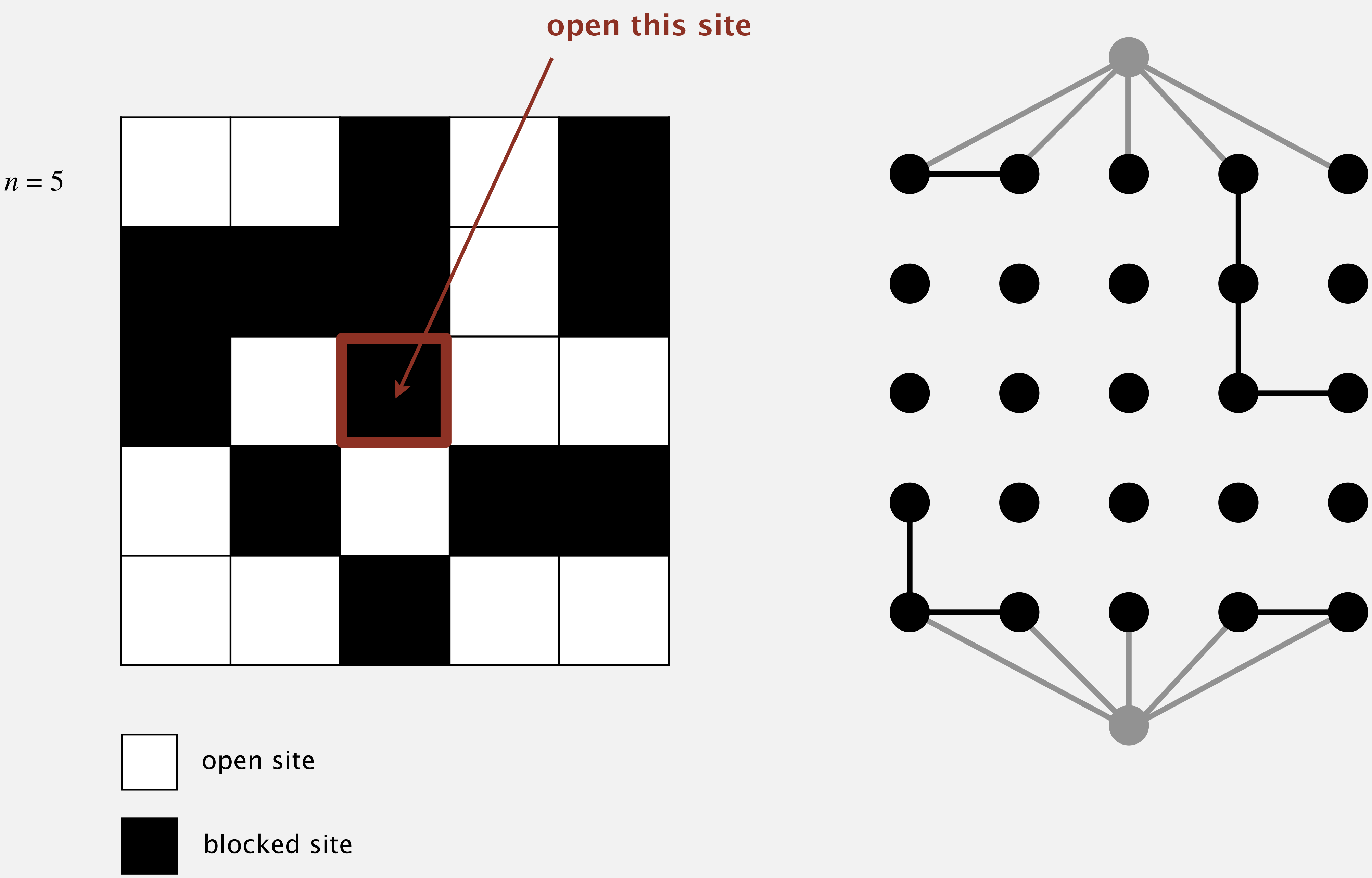
open site

blocked site



Dynamic-connectivity solution to estimate percolation threshold

Q. How to model opening a new site?

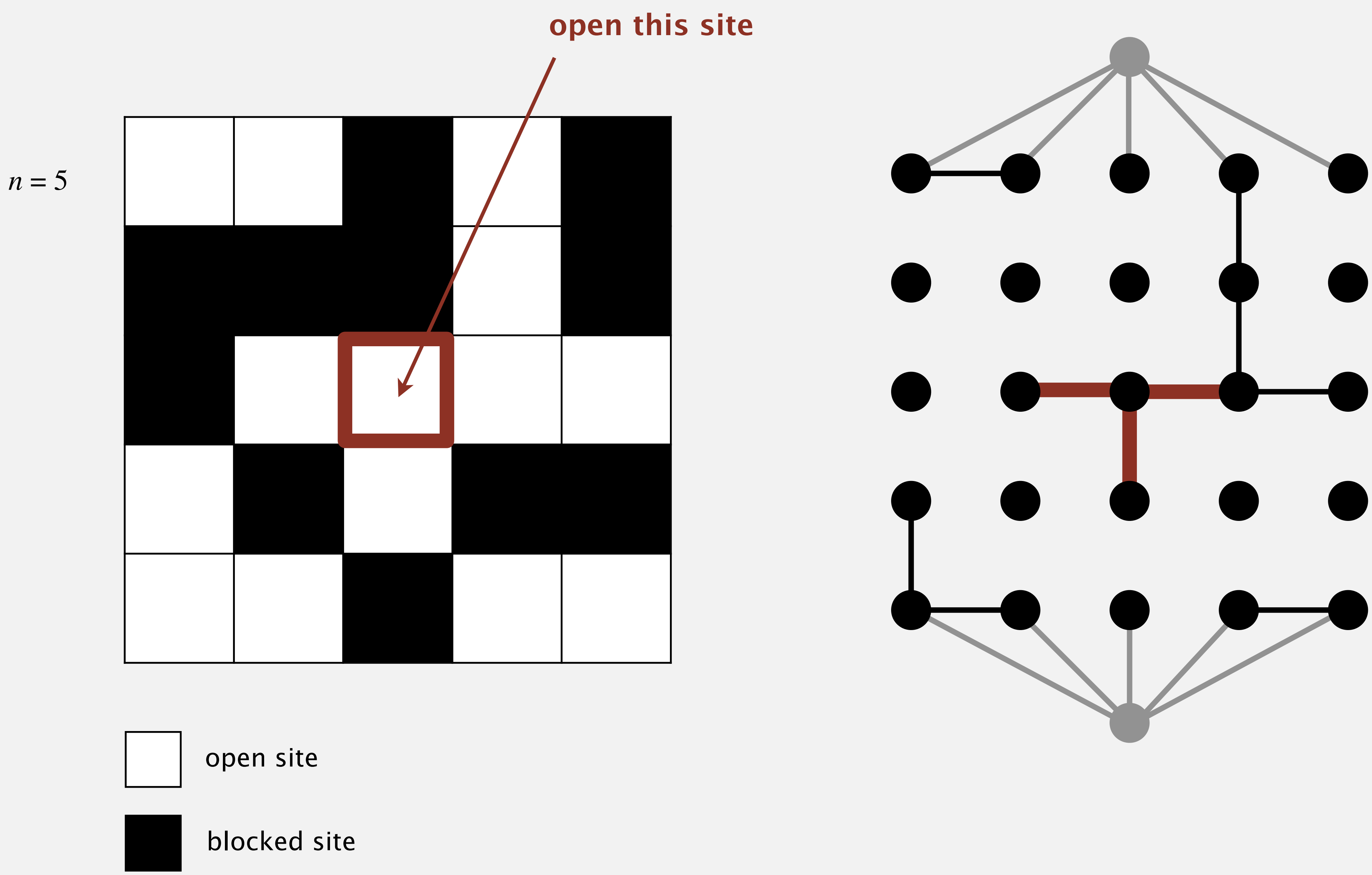


Dynamic-connectivity solution to estimate percolation threshold

Q. How to model opening a new site?

A. Mark new site as open; add edge to any adjacent site that is open.

adds up to 4 edges

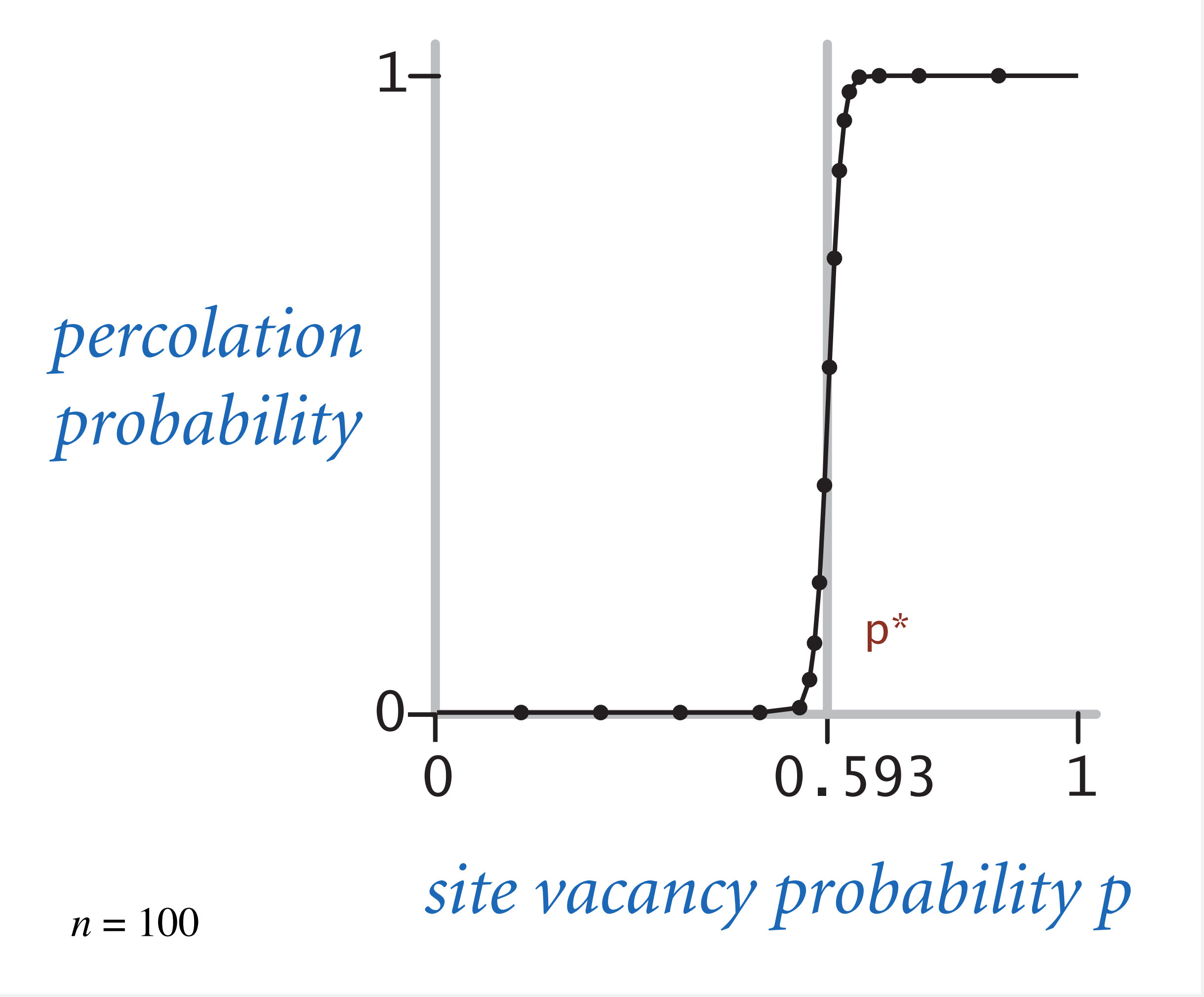


Percolation threshold

Q. What is percolation threshold p^* ?

A. About 0.592746 for large square lattices.

constant known only via simulation



Fast algorithm enables accurate answer to scientific question.

Subtext of today's lecture (and this course)

Steps to developing a usable algorithm.

- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.