1.5 Case Study

- dynamic connectivity
- quick find
- quick union
- improvements
- applications

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.

- **dynamic connectivity**
- quick find
- quick union
- improvements
- applications

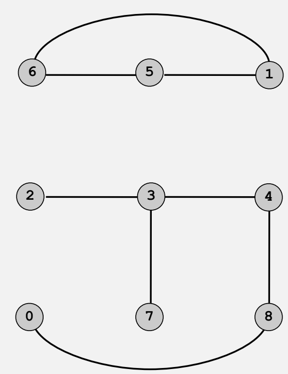
Dynamic connectivity

Given a set of objects

- Union: connect two objects.
- Find: is there a path connecting the two objects?

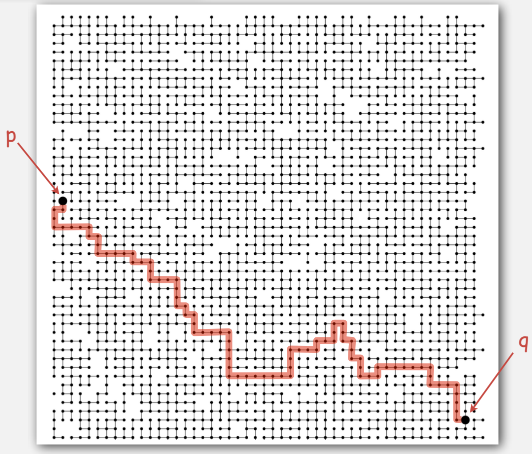
more difficult problem: find the path

```
union(3, 4)
union(8, 0)
union(2, 3)
union(5, 6)
 find(0, 2)      no
 find(2, 4)      yes
union(5, 1)
union(7, 3)
union(1, 6)
union(4, 8)
 find(0, 2)      yes
 find(2, 4)      yes
```



Network connectivity: larger example

Q. Is there a path from p to q?



A. Yes. ← but finding the path is more difficult: stay tuned (Chapter 4)

Modeling the objects

Dynamic connectivity applications involve manipulating objects of all types.

- Variable name aliases.
- Pixels in a digital photo.
- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Metallic sites in a composite system.

When programming, convenient to name objects 0 to N-1.

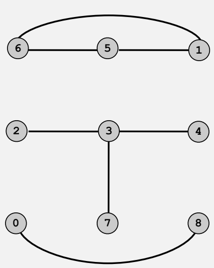
- Use integers as array index.
- Suppress details not relevant to union-find.

can use symbol table to translate from object names to integers (stay tuned)

Modeling the connections

Transitivity. If `p` is connected to `q` and `q` is connected to `r`, then `p` is connected to `r`.

Connected components. Maximal set of objects that are mutually connected.



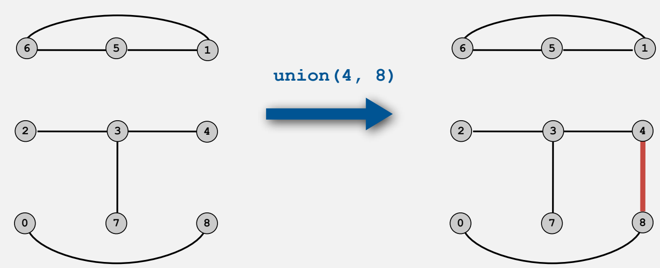
`{ 1 5 6 } { 2 3 4 7 } { 0 8 }`

connected components

Implementing the operations

Find query. Check if two objects are in the same set.

Union command. Replace sets containing two objects with their union.



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

connected components

Union-find data type (API)

Goal. Design efficient data structure for union-find.

- Number of objects N can be huge.
- Number of operations M can be huge.
- Find queries and union commands may be intermixed.

public class UnionFind		
	`UnionFind(int N)`	*create union-find data structure with N objects and no connections*
`boolean`	`find(int p, int q)`	*are p and q in the same set?*
`void`	`union(int p, int q)`	*replace sets containing p and q with their union*

- dynamic connectivity
- **quick find**
- quick union
- improvements
- applications

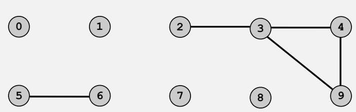
Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

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

5 and 6 are connected
2, 3, 4, and 9 are connected



Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

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

5 and 6 are connected
2, 3, 4, and 9 are connected

Find. Check if `p` and `q` have the same id.

`id[3] = 9`; `id[6] = 6`
3 and 6 not connected

Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

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

5 and 6 are connected
2, 3, 4, and 9 are connected

Find. Check if `p` and `q` have the same id.

`id[3] = 9; id[6] = 6`
3 and 6 not connected

Union. To merge sets containing `p` and `q`, change all entries with `id[p]` to `id[q]`.

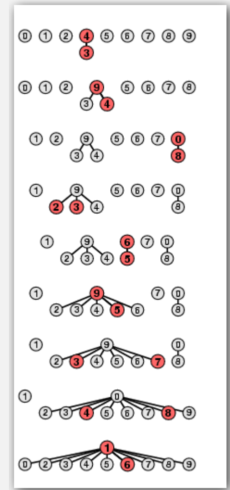
i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	6	6	6	6	6	7	8	6

union of 3 and 6
2, 3, 4, 5, 6, and 9 are connected

problem: many values can change

Quick-find example

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



problem: many values can change

Quick-find: Java implementation

```
public class QuickFind
{
   private int[] id;

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

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

   public boolean find(int p, int q)
   {  return find(p) == find(q);  }

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

set id of each object to itself
(N array accesses)

return component id for `p`
(1 array access)

check whether `p` and `q`
are in the same component

change all entries with `id[p]` to `id[q]`
(N array accesses)

Quick-find is too slow

Quick-find defect.

- Union too expensive (N array accesses).
- Trees are flat, but too expensive to keep them flat.

algorithm	union	find
quick-find	N	1

Ex. Takes N^2 array accesses to process sequence of N union commands on N objects.

Quadratic algorithms do not scale

Rough standard (for now).

- 10^9 operations per second.
- 10^9 words of main memory.
- Touch all words in approximately 1 second.

a truism (roughly) since 1950 !

Ex. Huge problem for quick-find.

- 10^9 union commands on 10^9 objects.
- Quick-find takes more than 10^{18} operations.
- 30+ years of computer time!

Paradoxically, quadratic algorithms get worse with newer equipment.

- New computer may be 10x as fast.
- But, has 10x as much memory so problem may be 10x bigger.
- With quadratic algorithm, takes 10x as long!

‣ dynamic connectivity
‣ quick find
‣ quick union
‣ improvements
‣ applications

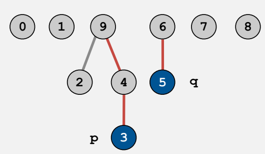
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

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



3's root is 9; 5's root is 6

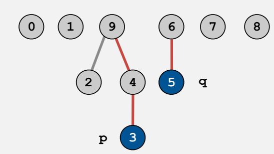
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

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



3's root is 9; 5's root is 6
3 and 5 are not connected

Find. Check if `p` and `q` have the same root.

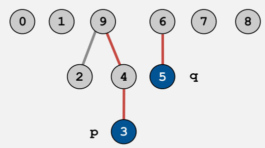
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

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



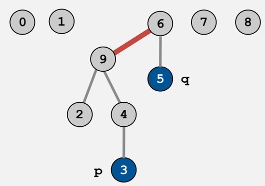
3's root is 9; 5's root is 6
3 and 5 are not connected

Find. Check if `p` and `q` have the same root.

Union. To merge sets containing `p` and `q`, set the id of `p`'s root to the id of `q`'s root.

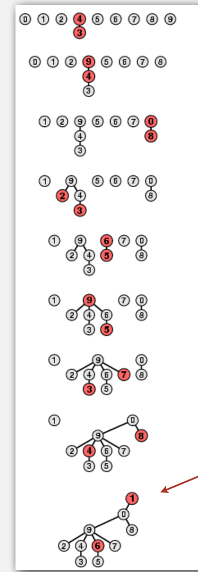
i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	6

only one value changes



Quick-union example

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



problem:
trees can get tall

Quick-union: Java implementation

```
public class QuickUnion
{
   private int[] id;

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

   private int find(int i)
   {
      while (i != id[i]) i = id[i];
      return i;
   }

   public boolean find(int p, int q)
   {
      return find(p) == find(q);
   }

   public void union(int p, int q)
   {
      int i = root(p), j = root(q);
      id[i] = j;
   }
}
```

set id of each object to itself
(N array accesses)

chase parent pointers until reach root
(depth of i array accesses)

check if p and q have same root
(depth of p and q array accesses)

change root of p to point to root of q
(depth of p and q array accesses)

Quick-union is also too slow

Quick-find defect.

- Union too expensive (N array accesses).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.

- Trees can get tall.
- Find too expensive (could be N array accesses).

algorithm	union	find
quick-find	N	1
quick-union	N †	N

worst case

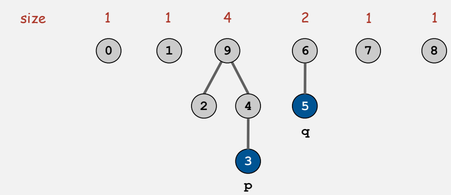
† includes cost of finding root

- dynamic connectivity
- quick find
- quick union
- **improvements**
- applications

Improvement 1: weighting

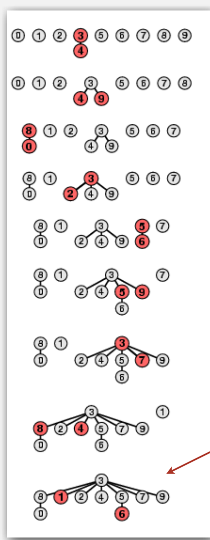
Weighted quick-union.
- Modify quick-union to avoid tall trees.
- Keep track of size of each set.
- Balance by linking small tree below large one.

Ex. Union of 3 and 5.
- Quick union: link 9 to 6.
- Weighted quick union: link 6 to 9.



Weighted quick-union example

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



no problem:
trees stay flat

Weighted quick-union: Java implementation

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

Find. Identical to quick-union.

```
return find(p) == find(q);
```

Union. Modify quick-union to:
- Merge smaller tree into larger tree.
- Update the `sz[]` array.

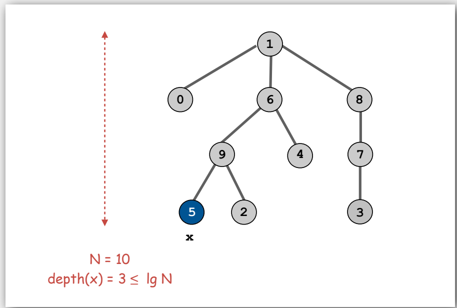
```
int i = find(p);
int j = find(q);
if  (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else                { id[j] = i; sz[i] += sz[j]; }
```

Weighted quick-union analysis

Analysis.

- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most lg N.



Weighted quick-union analysis

Analysis.

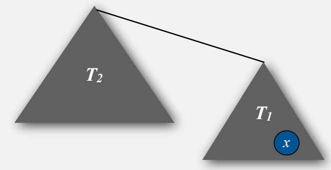
- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most lg N.

Pf. When does depth of x increase?

Increases by 1 when tree T_1 containing x is merged into another tree T_2.

- The size of the tree containing x at least doubles since $|T_2| \geq |T_1|$.
- Size of tree containing x can double at most lg N times. Why?



Weighted quick-union analysis

Analysis.

- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most lg N.

algorithm	union	find
quick-find	N	1
quick-union	N †	N
weighted QU	lg N †	lg N

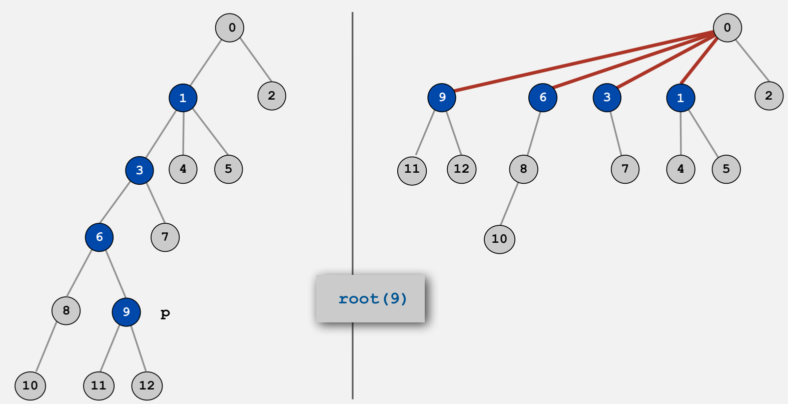
† includes cost of finding root

Q. Stop at guaranteed acceptable performance?

A. No, easy to improve further.

Improvement 2: path compression

Quick union with path compression. Just after computing the root of `p`, set the id of each examined node to point to that root.



Path compression: Java implementation

Standard implementation: add second loop to `find()` to set the `id[]` of each examined node to the root.

Simpler one-pass variant: halve the path length by making every other node in path point to its grandparent.

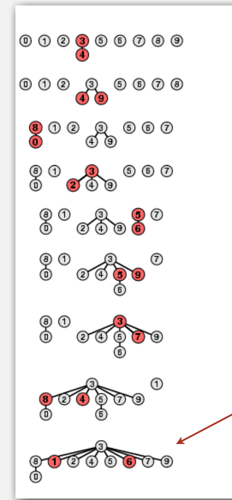
```
public int find(int i)
{
   while (i != id[i])
   {
      id[i] = id[id[i]];
      i = id[i];
   }
   return i;
}
```

only one extra line of code !

In practice. No reason not to! Keeps tree almost completely flat.

Weighted quick-union with path compression example

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



no problem:
trees stay VERY flat

WQUPC performance

Proposition. [Tarjan 1975] Starting from an empty data structure, any sequence of M union and find ops on N objects takes O(N + M lg* N) time.

- Proof is very difficult.
- But the algorithm is still simple!

actually O(N + M α(M, N))
see COS 423

Linear algorithm?

- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

because lg* N is a constant in this universe

N	lg* N
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

lg* function
number of times needed to take the lg of a number until reaching 1

Amazing fact. No linear-time algorithm exists.

Summary

Bottom line. WQUPC 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 QU	N + M log N
QU + path compression	N + M log N
weighted QU + path compression	N + M lg* N

M union-find operations on a set of N objects

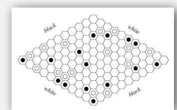
Ex. [10^9 unions and finds with 10^9 objects]

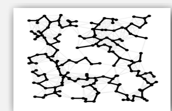
- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.

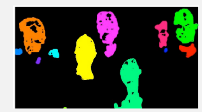
- dynamic connectivity
- quick find
- quick union
- improvements
- **applications**

Union-find applications

- Percolation.
- Games (Go, Hex).
- ✓ Network connectivity.
- Least common ancestor.
- Equivalence of finite state automata.
- Hoshen-Kopelman algorithm in physics.
- Hinley-Milner polymorphic type inference.
- Kruskal's minimum spanning tree algorithm.
- Compiling equivalence statements in Fortran.
- Morphological attribute openings and closings.
- Matlab's `bwlabel()` function in image processing.



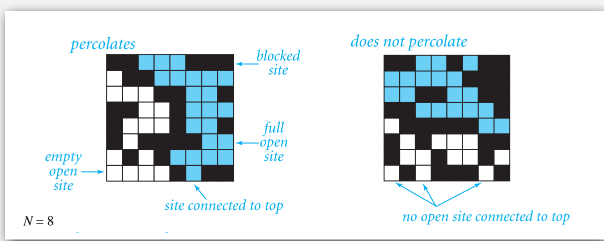




Percolation

A model for many physical systems:

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



Percolation

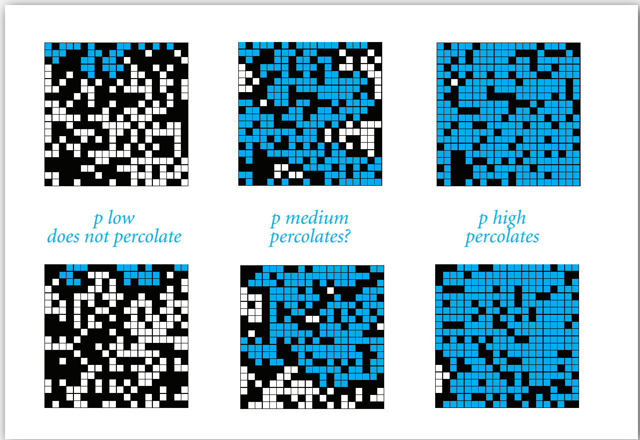
A model for many physical systems:

- N-by-N grid of sites.
- Each site is open with probability p (or blocked with probability 1-p).
- System percolates if 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 site vacancy probability p.



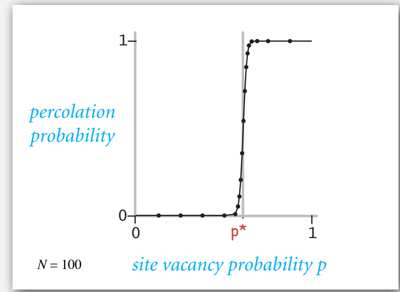
$N = 20$

Percolation phase transition

When N is large, theory guarantees a sharp threshold p*.

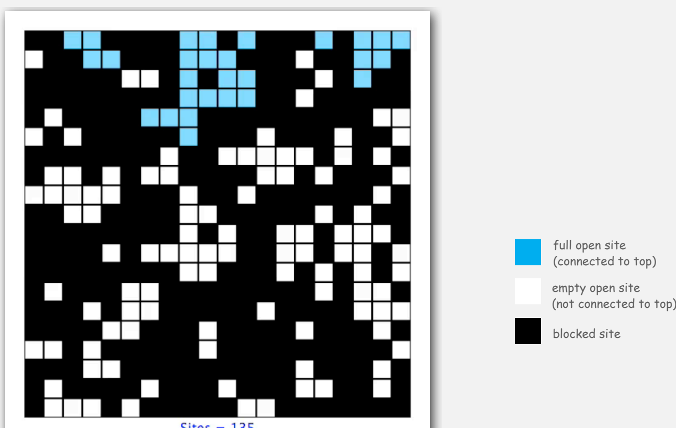
- p > p*: almost certainly percolates.
- p < p*: almost certainly does not percolate.

Q. What is the value of p* ?



Monte Carlo simulation

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



UF solution to find percolation threshold

How to check whether system percolates?

- Create an object for each site.
- Sites are in same set if connected by open sites.
- Percolates if any site in top row is in same set as any site in bottom row.

brute force algorithm needs to check N^2 pairs

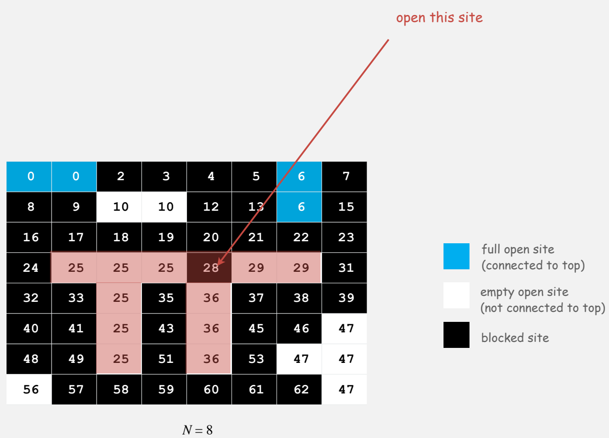
0	0	2	3	4	5	6	7
8	9	10	10	12	13	6	15
16	17	18	19	20	21	22	23
24	25	25	25	28	29	29	31
32	33	25	35	36	37	38	39
40	41	25	43	36	45	46	47
48	49	25	51	36	53	47	47
56	57	58	59	60	61	62	47

full open site (connected to top)
empty open site (not connected to top)
blocked site

$N = 8$

UF solution to find percolation threshold

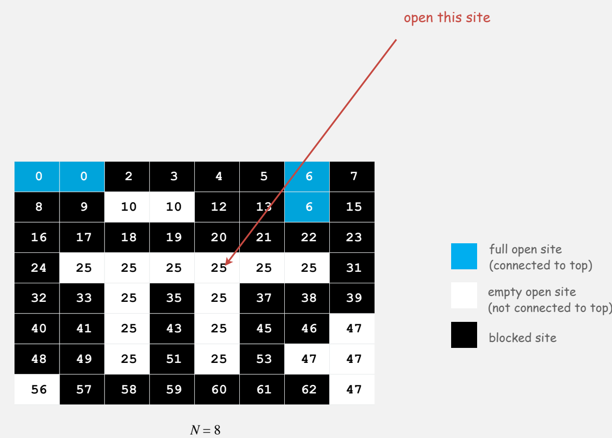
Q. How to declare a new site open?



UF solution to find percolation threshold

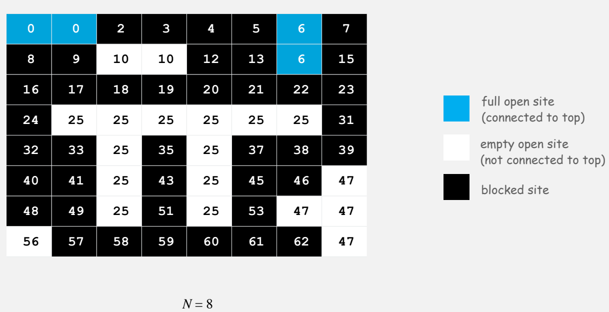
Q. How to declare a new site open?

A. Take union of new site and all adjacent open sites.



UF solution: a critical optimization

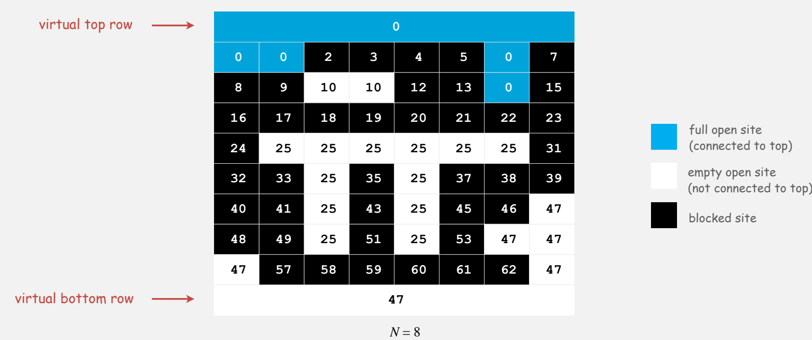
Q. How to avoid checking all pairs of top and bottom sites?



UF solution: a critical optimization

Q. How to avoid checking all pairs of top and bottom sites?

A. Create a virtual top and bottom objects;
system percolates when virtual top and bottom objects are in same set.

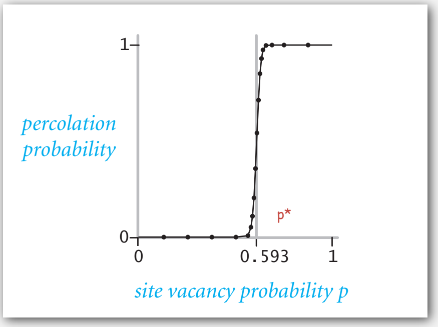


Percolation threshold

Q. What is percolation threshold p* ?

A. About 0.592746 for large square lattices.

percolation 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.