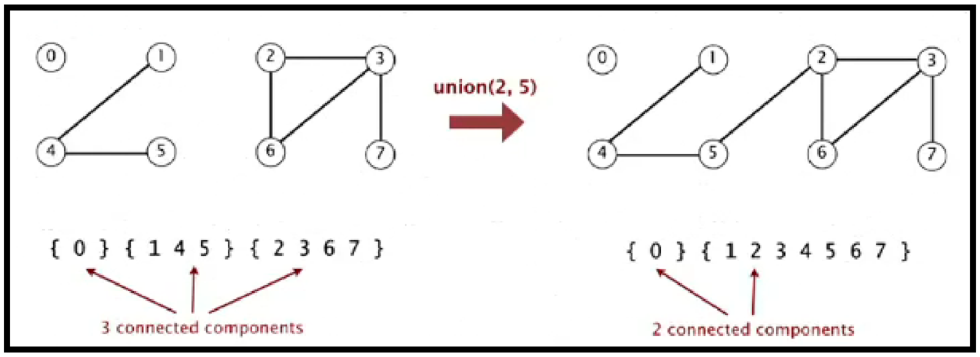
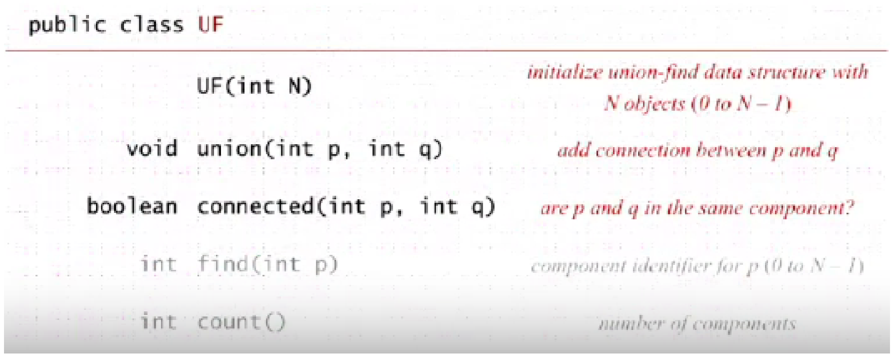
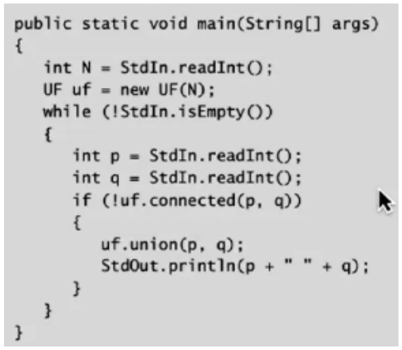
Union-Find (Week 1):

- Dynamic Connectivity
  - Given a set of N objects.
    - Union command
      - Connect two objects (given two objects, provide a connection between them).
    - Find/connected query
      - Is there a path connecting the two objects?
  - Modelling the objects
    - Applications of union find involve manipulating objects of all types:
      - Pixels in a digital photo.
      - Computers in a network.
      - Friends in a social network.
      - Transistors in a computer chip.
    - In programming, it is convenient to name objects 0 to N - 1.
      - Uses integers as array indices.
      - Suppresses details not relevant to union-find.
  - Modelling the connections
    - We assume "is connected to" is an equivalence relation:
      - Reflexive
        - P is connected to P.
      - Symmetric
        - If P is connected to Q, then Q is connected to P.
      - Transitive
        - If P is connected to Q and Q is connected to R, then P is connected to R.
    - Connected components
      - Maximal <u>set</u> of objects that are mutually connected.
  - Implementing the operations
    - Find query
      - Check if two objects are in the same component.
    - Union command
      - Replace components containing two objects with their union:
      - 

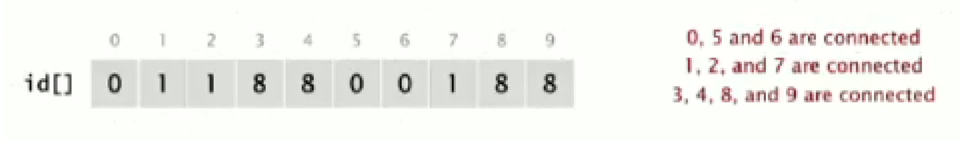
  - Union-find data type (API)
    - Goal

        - Design efficient data structure for union-find
    - Information
        - Number of objects N can be huge.
        - Number of operations M can be huge.
        - “Find queries” and “union commands” may be intermixed.
    - Class Overview
        - 

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

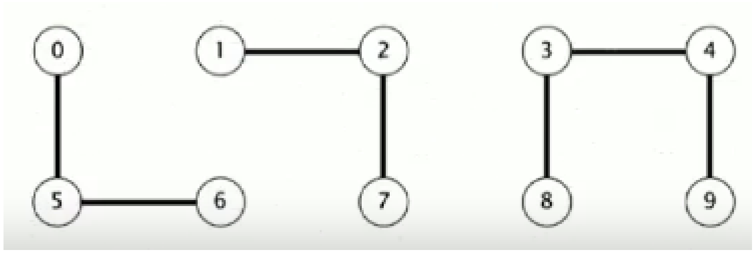
- Dynamic-connectivity client
    - Test client to test your code
        - Reads in the number of total objects N from standard input.
        - Then repeats:
            - Reads in a pair of integers from standard input.
            - If they are not yet connected, connects them and prints out the pair.
    - 

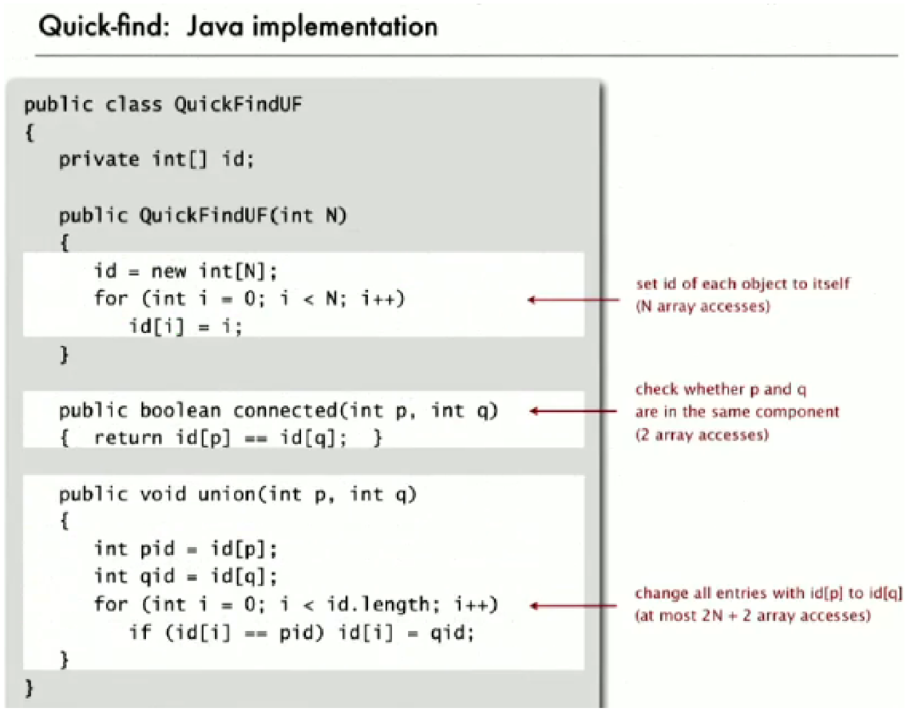
```
public static void main(String[] args)
{
   int N = StdIn.readInt();
   UF uf = new UF(N);
   while (!StdIn.isEmpty())
   {
      int p = StdIn.readInt();
      int q = StdIn.readInt();
      if (!uf.connected(p, q))
      {
         uf.union(p, q);
         StdOut.println(p + " " + q);
      }
   }
}
```

- Quick Find (eager approach)
    - Quick Find is our first implementation of an algorithm for solving the dynamic connectivity problem. It is an “eager” algorithm/approach.
    - Data Structure
        - Integer array id[] of size N.
            - Starts with index matching id. That is, id[2] would be 2 and so on.
        - Interpretation:
            - P and Q are connected if and only if they have the same id.
            - 

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

0, 5 and 6 are connected
1, 2, and 7 are connected
3, 4, 8, and 9 are connected

- 

    - Find/Connected query -> connected(P, Q)
        - Check if P and Q have the same id.
        - If id[6] = 0 and id[1] = 1, 6 and 1 would not be connected. Returns false.
    - Union command -> union(P, Q)
        - To merge components containing P and Q, change all entries whose id equals id[P] to id[Q].
    - Java Implementation
        - 

Quick-find: Java implementation

```
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 boolean connected(int p, int q)
   {  return id[p] == id[q];  }

   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;
   }
}
```

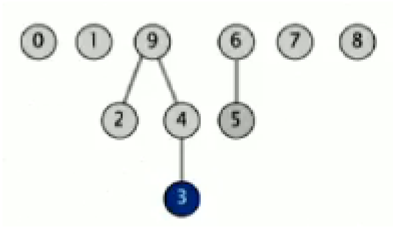
set id of each object to itself (N array accesses)

check whether p and q are in the same component (2 array accesses)

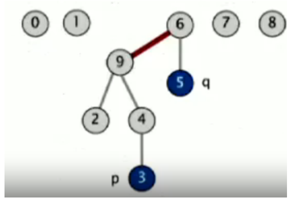
change all entries with id[p] to id[q] (at most 2N + 2 array accesses)

    - Issues
        - Quick-find is too slow
        - Cost model
            - Number of array accesses (for read or write)
                - Initializing the array takes N accesses
                - Using the union command takes N accesses (runs through the whole array and changes any that match id)
                - Find/connected query uses one access (checks if two ids are the same, constant time)
        - The union command is too expensive. If you were to have N unions on N number of objects, it would take N^2 time (quadratic) which is inefficient/too slow.
- Quick-union (lazy approach)

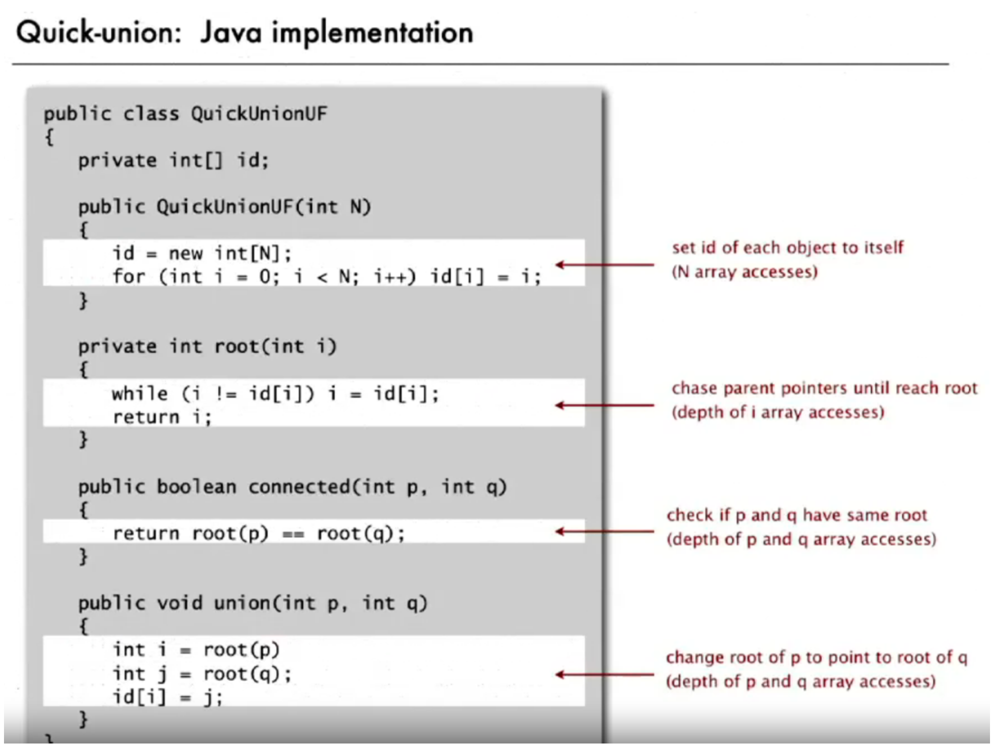
- Quick union is our second implementation of an algorithm for solving the dynamic connectivity problem. It is a "lazy" algorithm/approach.
    - We try to avoid doing work until we have to.
- Data Structure
    - Same as quick-find, integer array id[] of size N.
    - Interpretation
        - id[i] is the parent of i.
            - Each entry in the array is going to contain a reference to each parent in the tree.
        - The <u>root</u> of i is id[id[id[...id[i]...]]].
            - Tree representation:



- Find/Connected query -> connected(P, Q)
    - Check if P and Q have the same root.
        - "Are they in the same connected component?"
- Union command -> union(P, Q)
    - To merge components containing P and Q, set the id of P's <u>root</u> to the id of Q's <u>root</u>.
        - Example, connected(3, 5)
          We would need to change the root of P to the root of Q (changing the array value)



    - We are only changing one value, and that is the root of P.
- Java Implementation



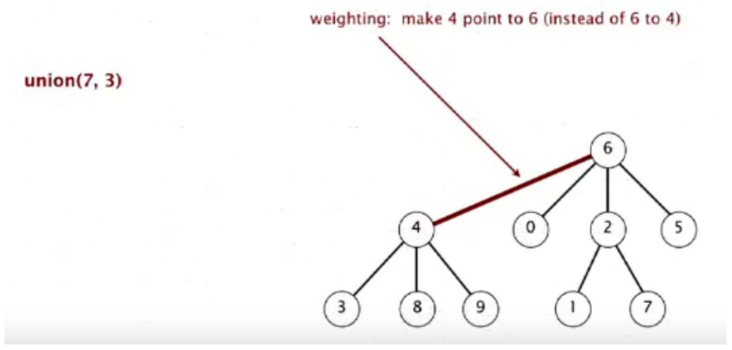
Quick-union: Java implementation

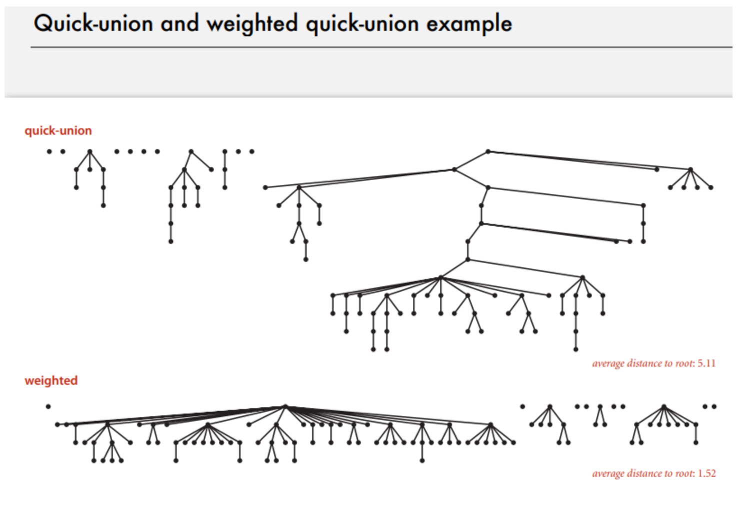
```
public class QuickUnionUF
{
   private int[] id;

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

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

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

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

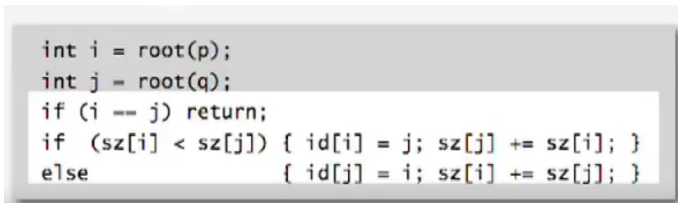
set id of each object to itself
(N array accesses)
chase parent pointers until reach root
(depth of i array accesses)
check if p and q have same root
(depth of p and q array accesses)
change root of p to point to root of q
(depth of p and q array accesses)

- Issues
    - Also too slow
    - Cost model
        - Initializing the array takes N accesses.
        - Using the union command takes N accesses (including the cost of finding roots).
        - Find/connected query uses N accesses (uses root method which is N at worst case).
    - Quick-find defect (eager approach)
        - Union is too expensive (N array accesses).
        - Trees are flat, but it's too expensive to keep them flat.
    - Quick-union defect (lazy approach)
        - Tress can get tall.
        - Find too expensive (could be N array accesses).

- Quick-Union Improvements
    - Improvement 1: Weighting
        - Weighted quick-union
            - Modify the quick-union algorithm to avoid tall trees.
            - Keep track of the size of each tree (number of objects).
            - Balance by linking the root of the smaller tree to the root of the larger tree.
            - Tall trees cause an expensive “find” operation (connected() method). If the tree is long and skinny, you would have to iterate through a lot to get to the root.

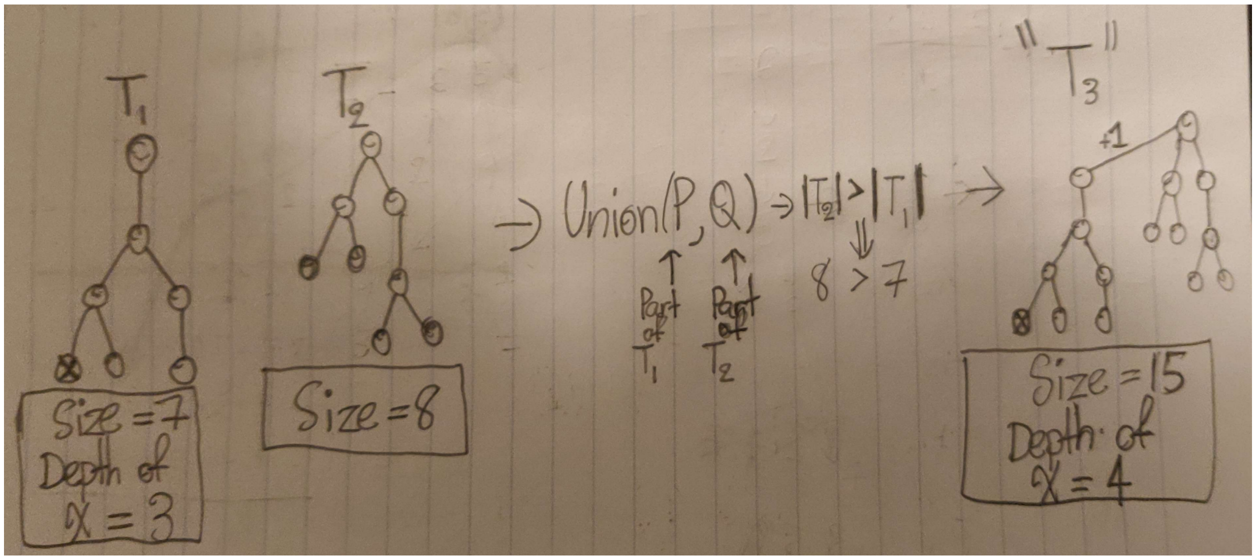
    - With weighted quick-union, we change the root of the smaller tree to the root of the larger tree. (Like the root of the smaller tree directly connects to the root of the larger tree).
    - By putting the smaller tree "lower" and linking/changing the root of the smaller tree to the root of the larger tree, all the trees would be flatter (not long skinny lines) and each object/tree would be closer to its root.
    - Can also think of "flatter" as less vertically long.
    - 

  - A visual example of tree difference between unweighted and weighted.
    - 
Quick-union and weighted quick-union example
quick-union
average distance to root: 5.11
weighted
average distance to root: 1.52
Quick-union and weighted quick-union (100 sites, 88 union() operations)
    - Since all the trees are flatter, the root method would be faster since all the trees are closer to the root and so iterating to the top would be faster.
- Data Structure
  - Same as quick-union, but have an extra array sz[i] to count the number of objects in the tree rooted at i.
- Find/Connected Query
  - Same as unweighted quick-union.
- Union Command
  - Modify quick-union to:
    - Link the root of the smaller tree to the root of the larger tree.

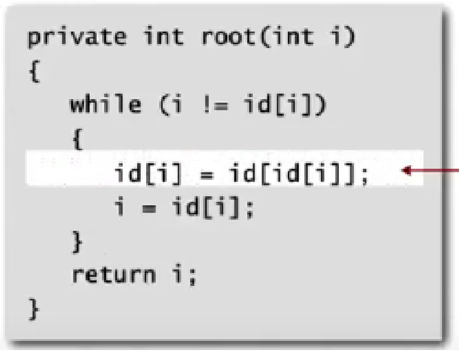
- Update the sz[] array.
- 

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

- Analysis
  - Running time
    - Find
      - Takes time proportional to the depth of P and Q.
    - Union
      - Takes constant time, given roots
  - Proposition
    - The depth of any node x is at most lgN (lg = base-2 logarithm).
    - Proof/Explanation
      - When does the depth of x increase?
        - It increases by 1 when tree T_1 containing x is merged into another tree T_2.
          - Because the weighting improvement causes the root of the smaller tree to link itself to the root of the larger tree adding 1 to the top of the smaller tree which adds 1 to the iteration from x to its root. I.e., increase the depth of x.
          - Visualization



        - The size of the tree containing x at least doubles since the size of T_2 has to be greater than or equal to the size of T_1.

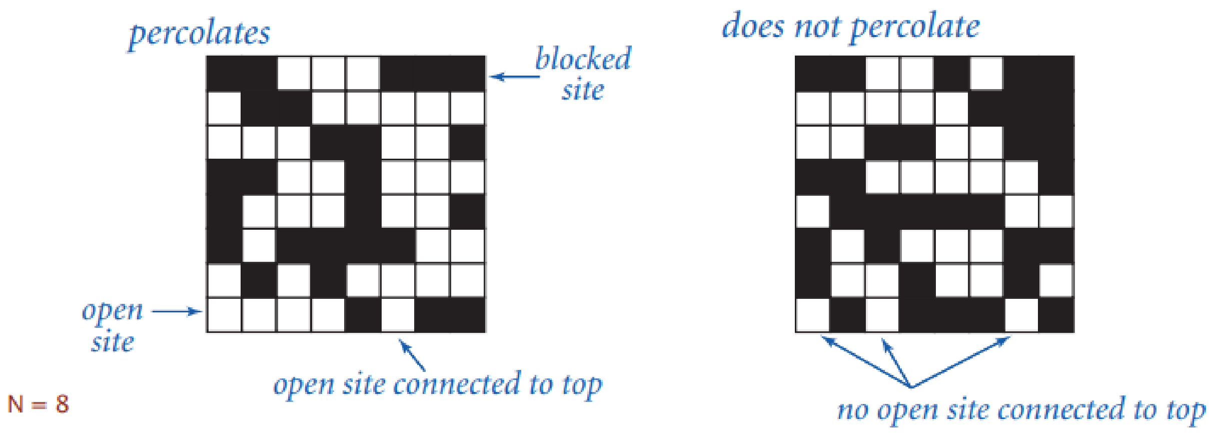
- If the size of the second tree is greater or equal to the size of T_1, then the new size (which is the combined size of both trees) would be at least double the size of T_1 since you are adding the same or more back into itself.
- The size of the tree containing x can only double at most lgN times. Why?
  - If you start with 1, and double it lgN times, you will end up with N, which is the max size of a tree since it is all of the objects existing.
  - Therefore the depth of any node x is at most lgN (log base-2 of N).
- Cost model
  - Initializing the array takes N accesses.
  - Using the union command takes lgN accesses (because you need to find the roots of both trees which is lgN complexity).
  - Find/connected query uses lgN accesses (because any node is at most lgN depth).
- Already better than both previous approaches.
  - Can still 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.
    - Implementations
      - Simple one-pass variant
        - Make every other node in the path point to its grandparent (1 node above the parent node),
        - Halves the path length since it connects each node to its grandparent node.
        - 

```
private int root(int i)
{
   while (i != id[i])
   {
      id[i] = id[id[i]];
      i = id[i];
   }
   return i;
}
```
        - Note that this still works when only one away from the top node because the id of the top node is itself. So the second to top node's parent id and grandparent id would

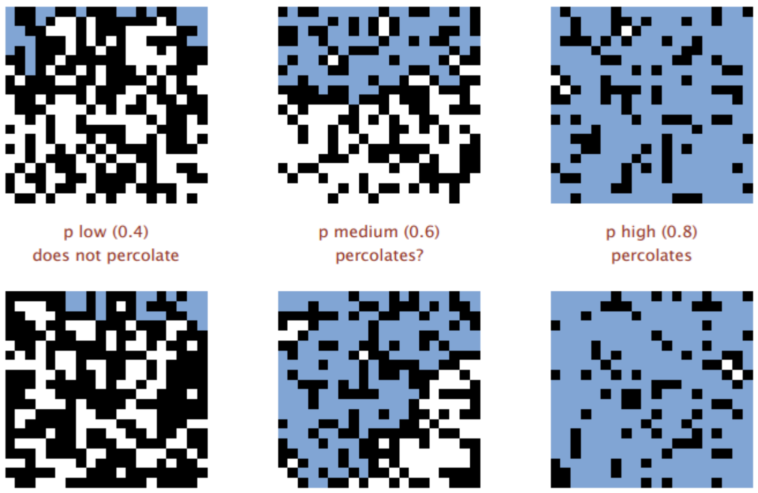
be the same since the top node's id is itself. If 1 and 2 are top and second top, 2's parent id is 1 and 1's parent id is 1 since it is the top node, this makes it so that 2's grandparent id is also 1.

        - Two-pass variant
            - Add a second loop to the root() method to set the id[] of each examined node to the root.
                - Flattens the trees by a lot as each node on the path to the top starting from x now directly connects to the root of the path.
    - Analysis
        - The combination of quick union with weighted union and path compression makes the algorithm so much faster it is considered almost linear.
- Summary
    - Weighted quick union with path compression makes it possible to solve problems that could not otherwise be addressed.
    - Reduces the time to perform 10^9 unions and finds with 10^9 objects from 30 years to 6 seconds.

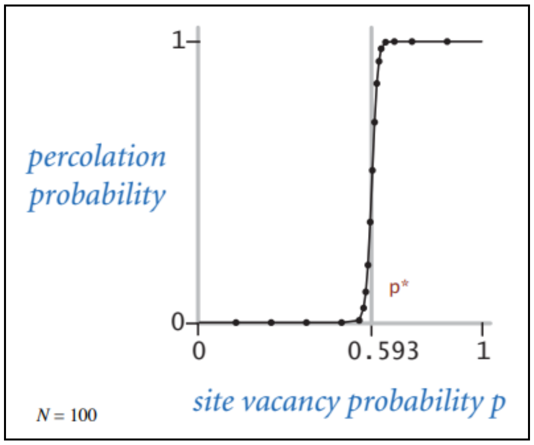
- Applications
    - Percolation
        - A model for many physical systems
            - N-by-N grid of "sites".
            - Each site is open with probability P (or blocked by probability 1 - P).
            - The system "percolates" if the top and bottom of the grid are connected by open sites. Open sites are white, and blocked sites are black.



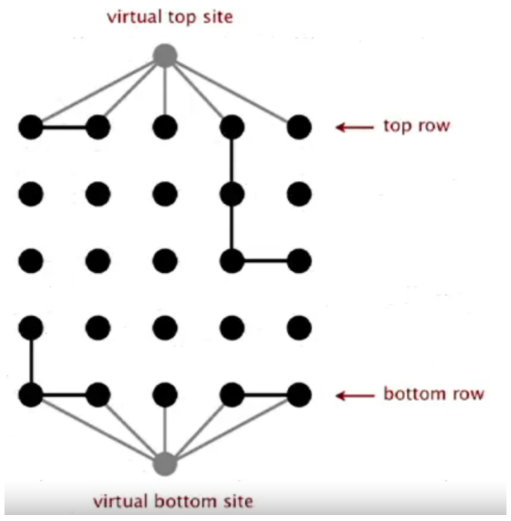
        - The likelihood of percolation depends on the site vacancy probability represented by P. Below are two examples of each various point of site vacancy probability.

- 

■ Percolation phase transition

- When N is large, theory guarantees a sharp threshold value P*.
    - P > P*
        - Almost certainly percolates.
    - P < P
        - Almost certainly does not percolate.
- The question is, what is the value of P*?
    - 

    - We don't actually know how to calculate the value so we use simulations and run them over and over to find a value through trial and error.
- Monte Carlo simulation
    - Initialize N-by-N whole grid to be blocked.
    - Declare random sites open until the top connects to the bottom.
    - The vacancy percentage (how many open sites out of the total sites are on the grid at the time the grid percolates?) estimates P*.
- Dynamic connectivity solution to estimate percolation threshold
    - 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 x n grid creates n^2 objects, with array indexing we get 0 to n^2-1)
            - Open sites are in the same component if they are connected to other open sites.
            - The system percolates if any open site on the bottom row is connected to any open site on the top row.
                - We can introduce two virtual sites, one that connects to the top row and the other connects to the bottom row. Then instead of checking each site in the top and bottom rows, you can just check if the bottom virtual site is connected to the top virtual site.
                - 

        - How to model opening a new site?
            - Connect a newly opened site to all of its adjacent open sites.
                - Up to 4 union() calls.
        - By simulating this on a large scale repeatedly, you'll end up with a threshold value of P* = 0.592746.
- Percolation assignment grade: 92%