**Algorithms – Part I – Lecture Notes**

**Week 1**

**Union-Find**

**Dynamic Connectivity**

Steps to Developing a Usable Algorithm

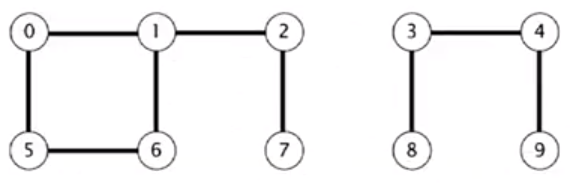
* Model the problem
* Find an algorithm to solve it
* Fast enough? Fits in memory?
* If not, address the issue
* Iterate until satisfied

Dynamic Connectivity

* Given a set of objects, we must support the following commands
  + Union command: connect two objects
  + Find/connected query: is there a path connecting the two objects



* After we call union(5, 0), union(7, 2), union(6, 1), union(1, 0), we get the following graph.



* Then, given the command connected(0, 7), this returns true.

Modeling the Objects

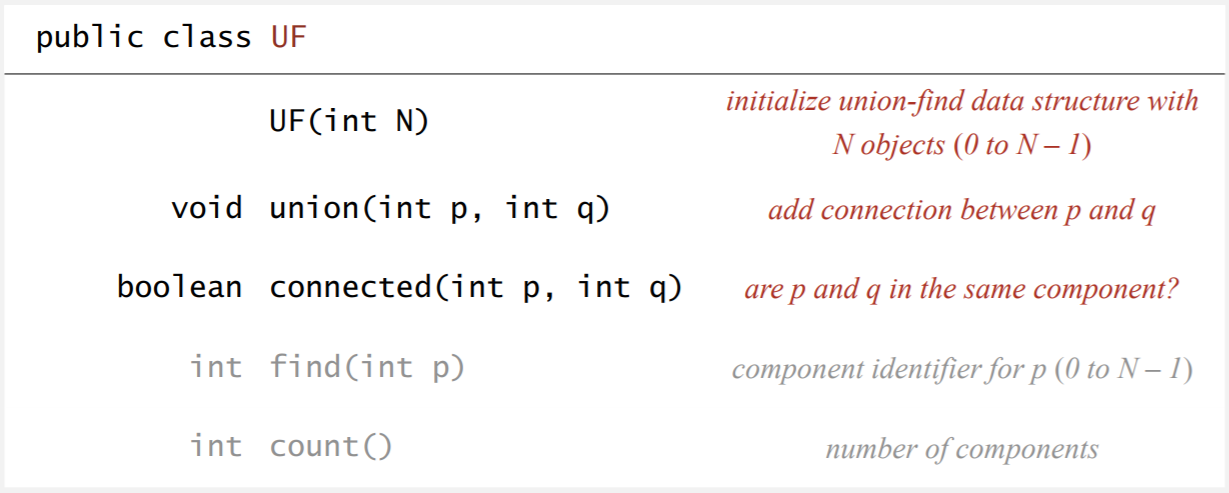
* Pixels in a digital photo
* Computers in a network
* Friends in a social network
* Transistors in a computer chip
* Elements in a mathematical set
* Variable names in FORTRAN
* Metallic sites in a composite system
* When programming, it is convenient to name objects from 0 to .

Modeling the Connections

* We assume “is connected to” is an equivalence relation:
  + Reflexive: is connected to
  + Symmetric: if is connected to , then is connected to
  + Transitive: if is connected to and is connected to , then is connected to
* **Connected components:** Maximal *set* of objects that are mutually connected.

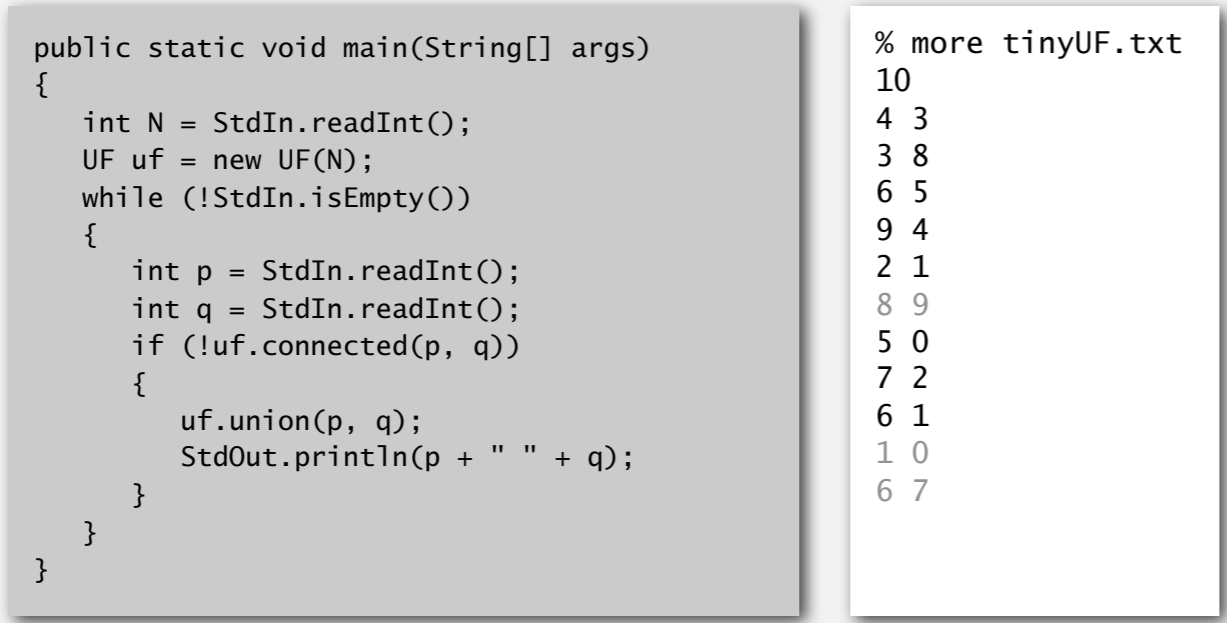
Implementing the Operations

* *Find query.* Check if two objects are in the same component.
* *Union command.* Replace components containing two objects with their union.



Dynamic-Connectivity Client

* Read in number of objects from standard input.
* Repeat:
  + read in pair of integers from standard input
  + if they are not yet connected, connect them and print out a pair

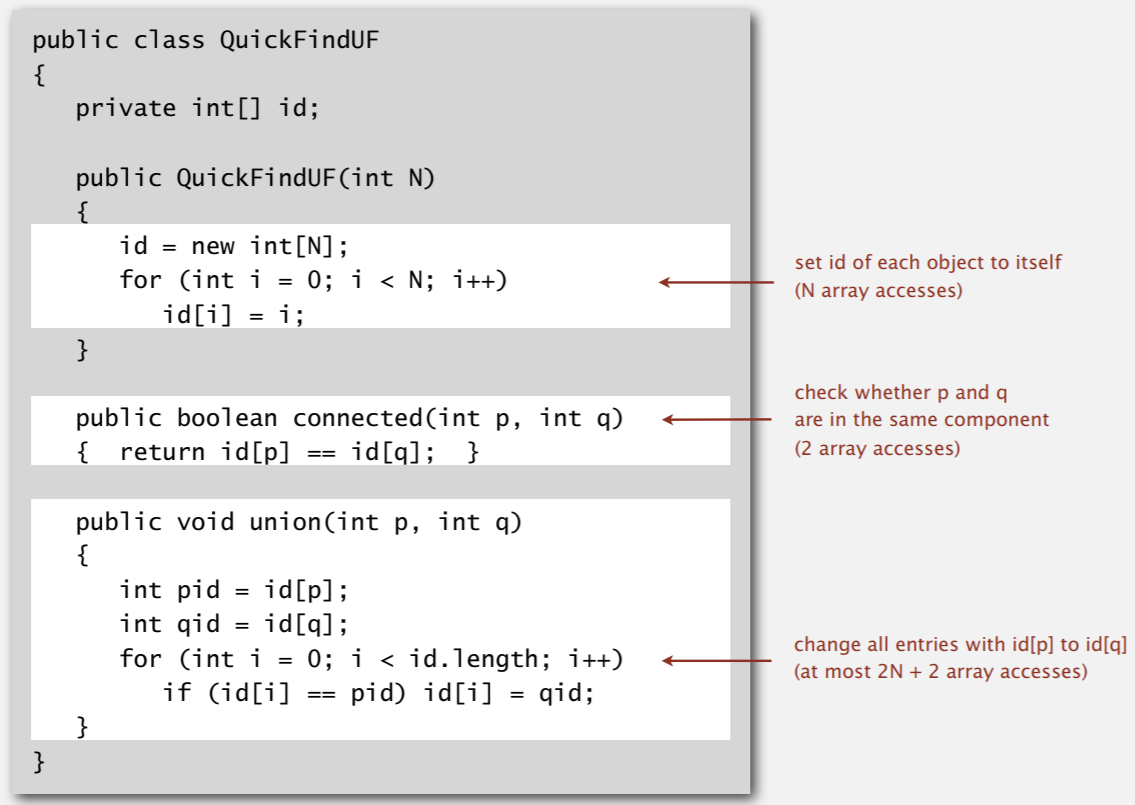


**Quick Find**

[Eager Approach]

* Data structure used:
  + Integer array id[] of size N
  + Interpretation: and are connected iff they have the same id
* *Find.* Check if and have the same
* *Union.* To merge components containing *p* and *q*, change all entries whose id equals id[p] to id[q].
* What this does is makes it very easy to see if two elements i and j are connected. We simply see if id[i] == id[j].

Quick-Find: Java Implementation



* The first function QuickFindUF(int N) simply initializes the id array with each element corresponding to its index (no unions yet).
* The function connected(int p, int q) produces id[p] == id[q] since this determines if two elements are connected.
* The function union(int p, int q) says if any of the ids in the id[i] == id[p], then we set that id to id[i] = id[q].

Quick-Find is Too Slow

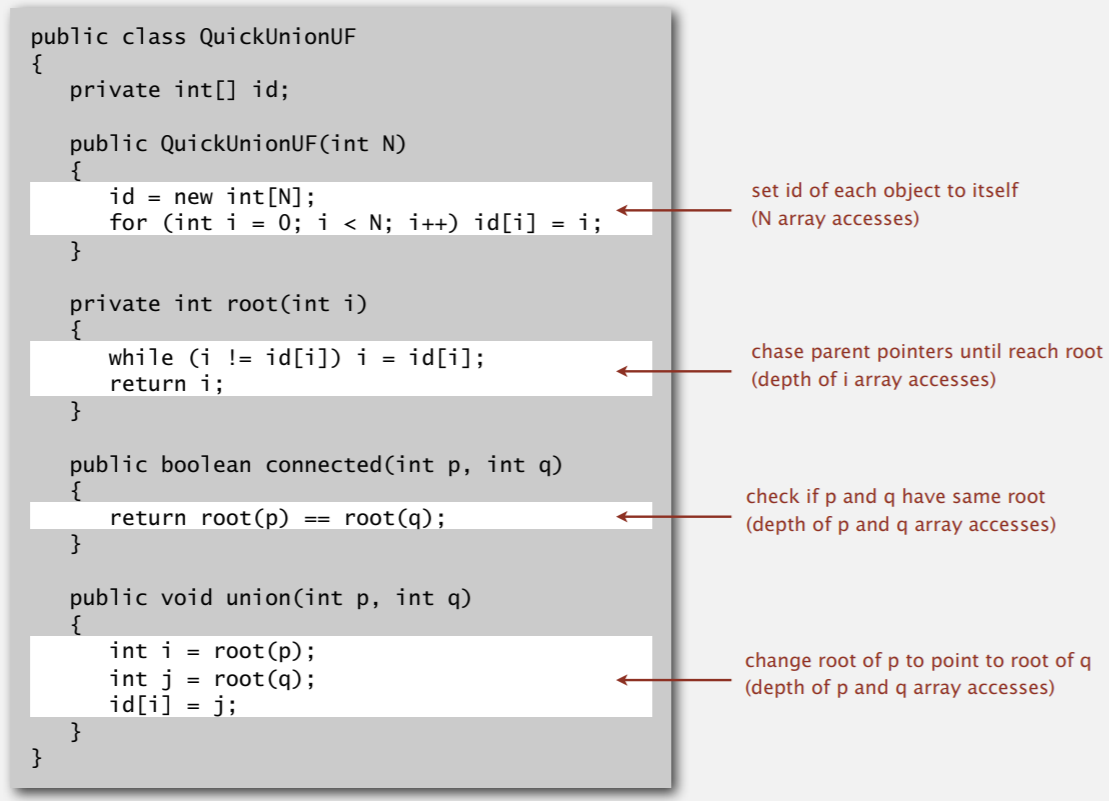
* The union function is too slow because it operates it goes through each of the elements in the array and checks for elements. However, this means our entire QuickFindUF operates in time, which is quite slow.
* The problem with quadratic algorithms is that they don’t scale with the technology. It starts to take much more time very quickly.

**Quick Union**

[Lazy Approach]

* The same data structure is used, but with a different interpretation.
  + Integer array id[] of size N
  + *Interpretation:* id[i] is parent of i
  + Root of i is id[id[…id[i]…]]
* *Find.* Check if *p* and *q* have the same root.
* *Union.* To merge components containing *p* and *q*, set the id of *p*’s root to the id of *q*’s root (only changes 1 entry).

Quick-Union: Java Implementation



Quick-Union is Too Slow

* Quick-Find defect (last video)
  + Union too expensive ( array accesses)
  + Trees are flat, but it is expensive to keep them flat
* Quick-Union defect (this video)
  + Trees can get tall
  + Find too expensive (could be up to array accesses)

**Quick-Union Improvements**

Improvement 1: Weighting