CS 251, Spring 2012

Data Structures and Algorithms

Prof. Xavier Tricoche

xmt@purdue.edu

PURDUE UNIVERSITY

Overview

- Course overview
- Introductory example: *Union find*

Course information

Time and location:

- T/R 12:00-1:15pm, ME 1061

Instructor:

- Prof. Xavier Tricoche, LWSN 3154P, xmt@purdue.edu

Office hours:

- See web page

Teaching Assistants:

- Mohammed Almeshekah
- Duc Nguyen Chi
- Shunrang Cao
- Pawan Hosakote Nagesh
- Rahul Nanda

Website:

- www.cs.purdue.edu/homes/cs251

Course goals

Program = Algorithm + Data Structures

- Algorithm: method for solving a problem.
- Data structure: method to store information.
- CS251: programming and problem solving, with applications.

Learning objectives:

1) learn how data representation in the computer impacts the performance of a program
2) different types of data structures and algorithms that utilize these data structures
3) improve your programming skills

Course content

- ✓ Simple proof techniques
- ✓ Program analysis
- ✓ Bags, stacks, and queues
- ✓ Trees
- ✓ Heaps and priority queues
- ✓ Sorting algorithms
- ✓ Search trees
- ✓ Hash tables
- ✓ Graphs
- ✓ Text processing
- ✓ Data compression

Course information

Textbook:

- Algorithms, Sedgewick and Wayne, 4th edition

Other course information:

- Syllabus, schedule of topics and slides available on course webpage
- Read chapter/section before class (see schedule page)
- Print /download slides and bring to class
- Take notes in class

Prerequisites

The class assumes that you have either (i) good Java background, or (ii) basic Java + OO programming background

- Data types
- Control statements
- Arrays, simple classes
- Inheritance and polymorphism
- Exceptions
- Interfaces and abstract classes

Basic background in C++ is also assumed

The algorithms will be presented in pseudocode or Java

- Programming projects must be completed in Java
- One project will require use of C++

Programming

Follow good programming style. See:

- http://www.jbonneau.com/style_guide.pdf
- http://192.220.96.201/essays/java-style/

Resources:

- Java API: http://download.oracle.com/javase/1.5.0/docs/api/index.html
- http://www.cs.princeton.edu/introcs/home/ (for Java basics)

Course resources

Schedule:

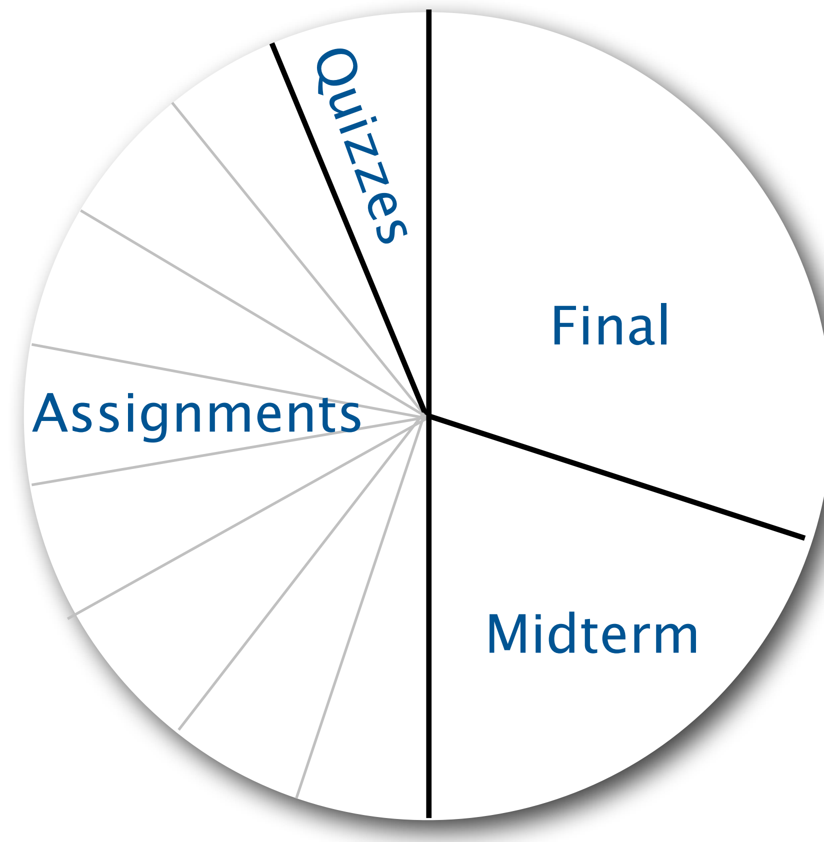
- See course web page:
  - No class on Oct 9 (October break), and Nov 22 (Thanksgiving)
  - No PSO on Oct 8 (October break), Nov 22-23 (Thanksgiving)
  - Midterm will be during class hours on October 11 (ME 1061)
  - Final exam: TBD

- Links to slides, assignments, and additional readings are on the schedule
  - Material is password protected, username: cs251-fall, password: NlogN
  - Links will be activated as material is posted
  - DO NOT distribute material

Coursework and grading

Assignments: 45%

- 4 written homeworks
- 5 programming projects (4 in Java, 1 in C++)
- Due at 11:59pm via electronic submission



Quizzes: 5%

- Given very regularly
- Showing up is half the battle!

Exams: 20% + 30%

- Closed-book, closed notes.
- Midterm (Oct 11)
- Final (scheduled by Registrar)
  - comprehensive but emphasis on latter topics

Grades will be reported on Blackboard

Course logistics

You will need a CS account

- You should already have one.
- If not, contact me!

Email

- We will use a mailing list for class announcements
- The mail alias cs251-ta@cs.purdue.edu is for contacting the TAs
- Feel free to email me as well (xmt@purdue.edu)
- Use appropriate language when sending messages

Course logistics

Course content:

- The course moves very fast
- Attend all lectures
  - Lectures will assume that you have read the material from the text. We will build on that.
- Attendance for PSOs is highly recommended
- Quizzes will check basic understanding of material and attendance

Lecture etiquette:

- Students are expected to focus their attention on the lecture (e.g., no Facebook)
- No talking among students
- Before class allow instructor to prepare before asking questions

Course policies

Missing exams:

- If you cannot make an exam, contact the instructor BEFORE the exam, otherwise you will receive 0 on the exam
- Exceptions: documented medical and family emergencies only

Late policy:

- Each person will be allowed **4 days** of extensions which can be applied to any combination of assignments during the semester without penalty
  - Use of a partial day will be counted as a full day
  - Use of extension must be stated explicitly in the submission header or by email to the TAs, otherwise late penalties will apply
  - Extensions cannot be applied after the final day of classes (ie., Dec 8)
  - Extensions cannot be rearranged after they are granted. Use them wisely!
- After that a late penalty of 20% per day will be assigned
- Assignments will not be accepted if they are more than five days late

Course policies

Campus emergencies:

- Course requirements, deadlines, and grading are subject to change
- Course website and email list will be used to notify you
  - Emergencies include: pandemics, weather extremes, hazardous spills, safety issues, etc
- H1N1 (or other contagious flu)
  - Do not attend lectures or PSOs
  - Contact instructor via email to make arrangements

Ethics

We encourage you to interact amongst yourselves:

- You may discuss and obtain help with basic concepts covered in lectures or the textbook, homework specification (but not solution), and program implementation (but not design)

However, this is NOT a team programming course:

- Work turned in should reflect your own efforts and knowledge.
- Sharing or copying solutions is unacceptable. It can result in failure for the course AND exclusion from Purdue (for repeated offenders).
- We use copy detection software, so do not copy code and make changes (either from the Web or from other students).
- You are expected to take reasonable precautions to prevent others from using your work.

Read the Academic Integrity Policy on the web page and SIGN IT

- Only those who have signed it will be allowed to take the midterm exam

Example: Union Find

- dynamic connectivity
- applications
- quick find

Take home message

Steps to developing a **usable** algorithm.

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

- dynamic connectivity
- applications
- quick find

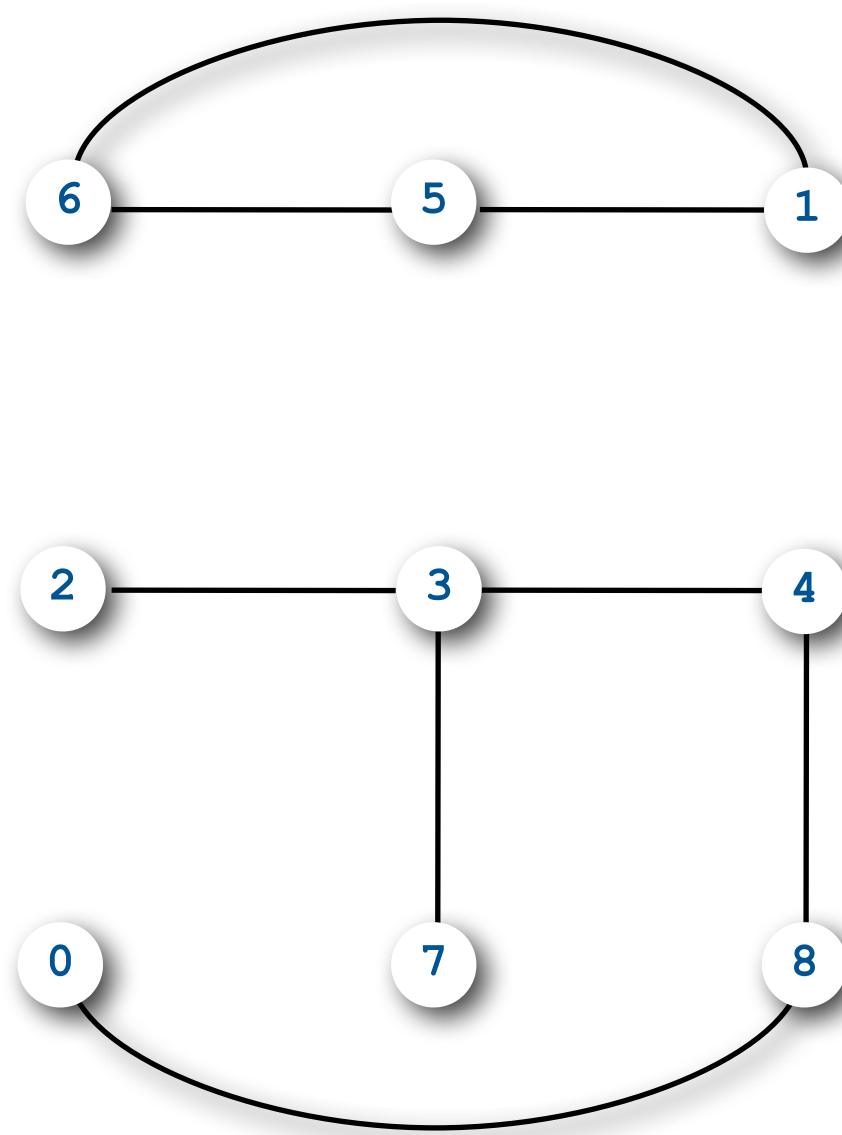
Dynamic connectivity

Given a set of objects

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

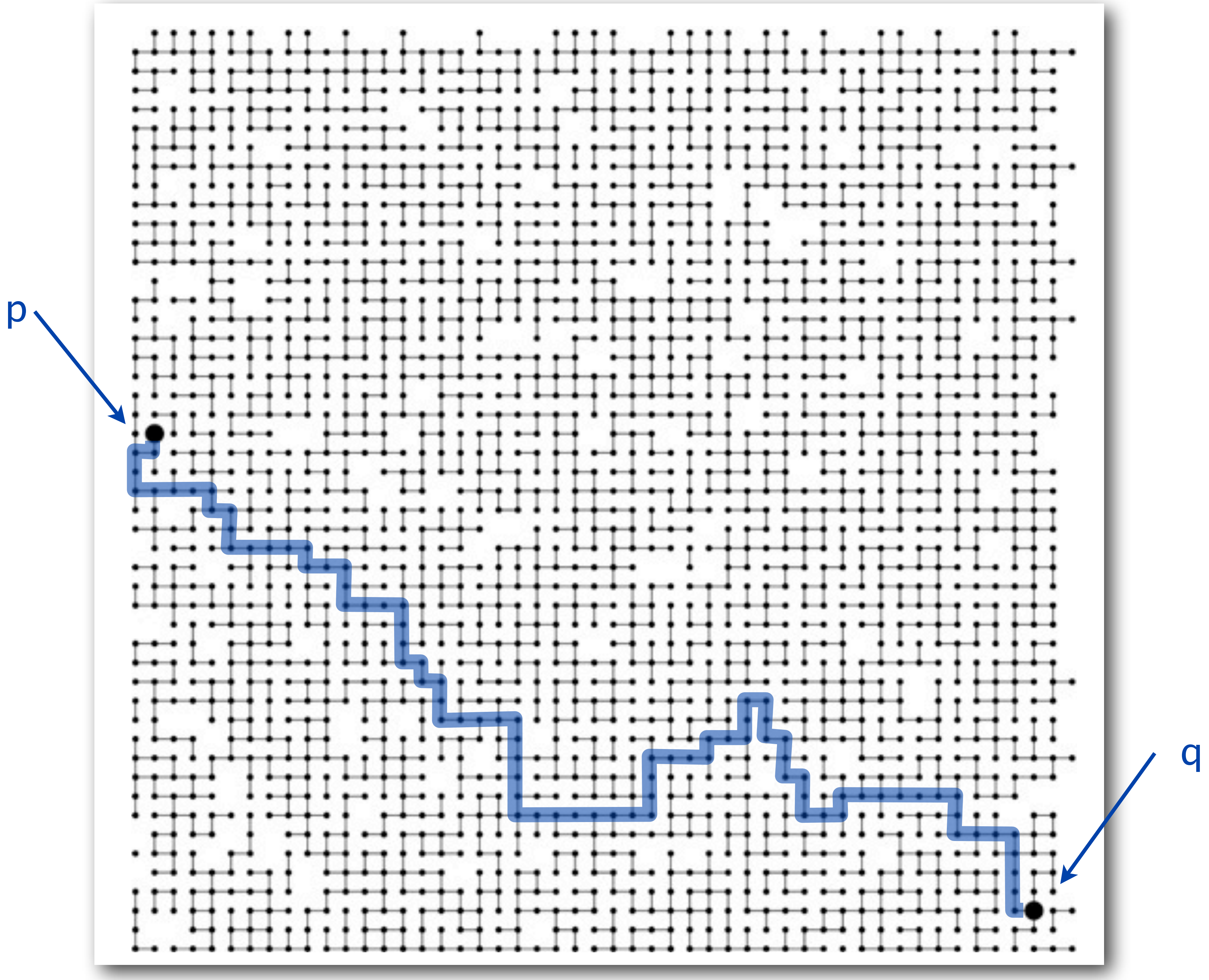
more difficult problem: find the path

```
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
```



Connectivity example

Q. Is there a path from p to q?



A. Yes.

Modeling the objects

Dynamic connectivity applications involve manipulating objects of all types.

- Pixels in a digital photo.
- Computers in a network.
- Variable names in Fortran.
- Friends in a social network.
- Transistors in a computer chip.
- Elements in a mathematical set.
- Metallic sites in a composite system.

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

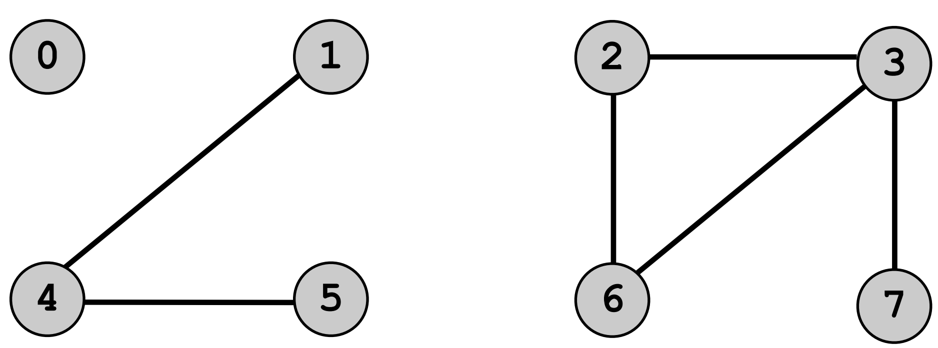
- Use integers as array index.
- Suppress details not relevant to union-find.

Modeling 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 set of objects that are mutually connected.



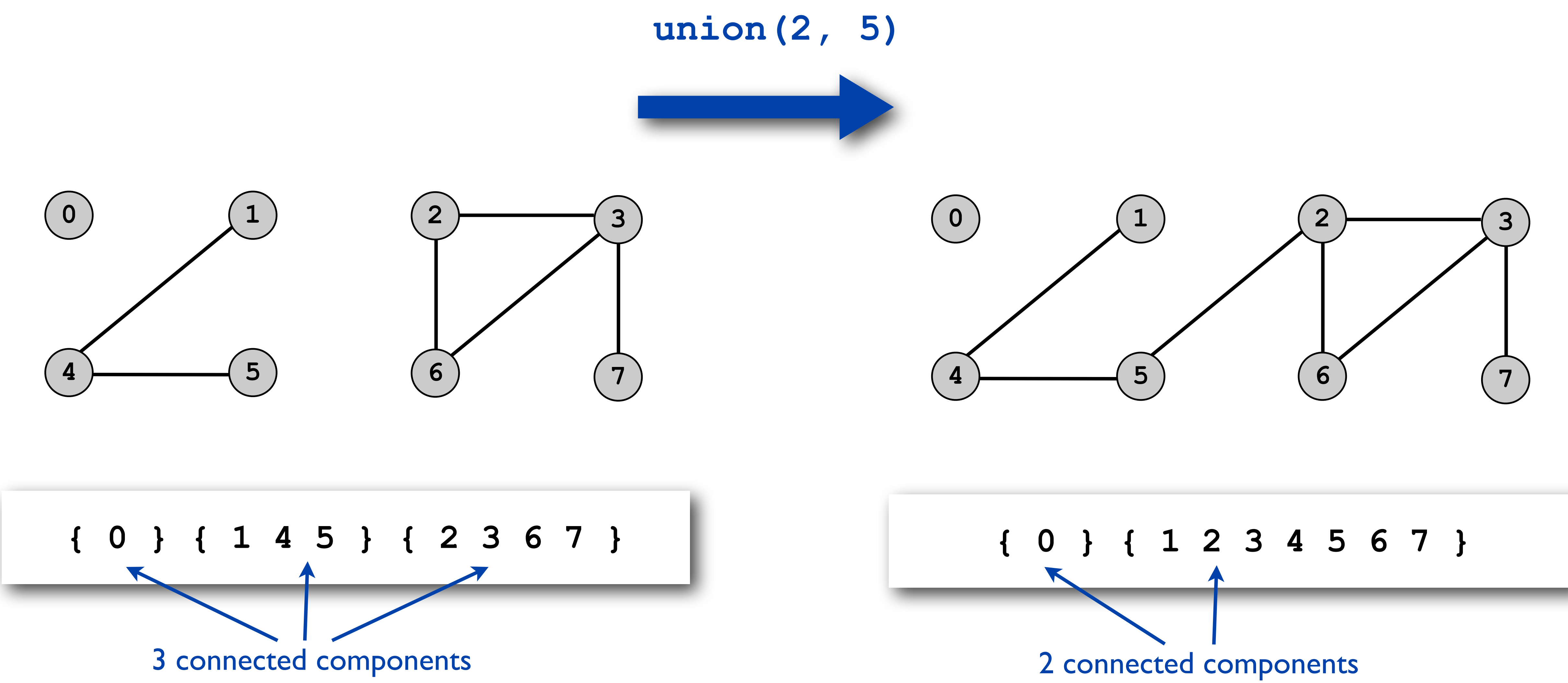
{ 0 } { 1 4 5 } { 2 3 6 7 }

3 connected components

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.

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

public class UF		
	`UF(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 component?*
`void`	`union(int p, int q)`	*add connection between p and q*
`int`	`count()`	*number of components*

Dynamic-connectivity client

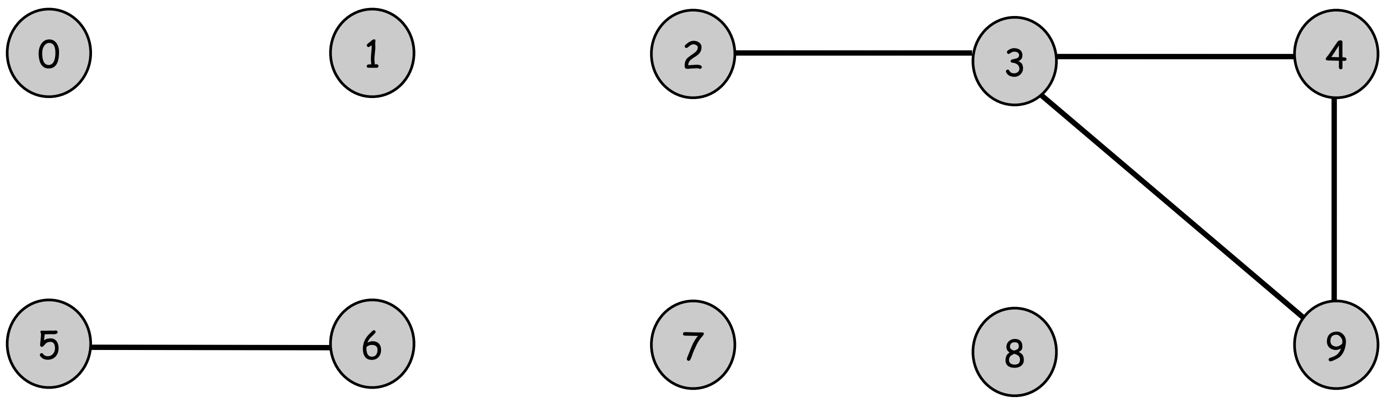
- Read in number of objects N from standard input.
- Repeat:
  - read in pair of integers from standard input
  - write out pair if they are not already connected

```
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.find(p, q)) continue;
      uf.union(p, q);
      StdOut.println(p + " " + q);
   }
}
```

```
% more tiny.txt
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
```

- dynamic connectivity
- applications
- quick find

Example



Quick-find [eager approach]

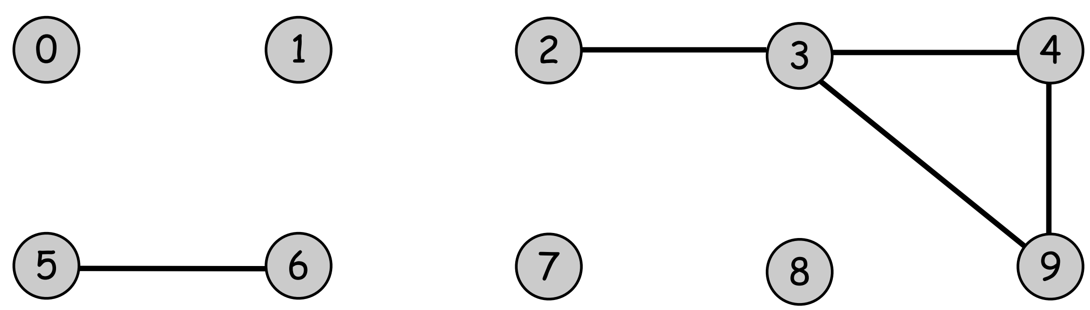
Data structure.

- Integer array **`id[]`** of size **`N`**.
- Interpretation: **`p`** and **`q`** in same component iff 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`** in same component iff 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 in different components

Quick-find [eager approach]

Data structure.

- Integer array **`id[]`** of size **`N`**.
- Interpretation: **`p`** and **`q`** in same component iff 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 in different components

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

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

`id[p]` *and* `id[q]` *differ, so* `union()` *changes entries equal to* `id[p]` *to* `id[q]` *(in red)*

`id[p]` *and* `id[q]` *match, so no change*

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 find(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;
   }
}
```

Quick-find is too slow

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

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 find(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]**
(linear number of array accesses)

Quick-find is too slow

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

algorithm	init	union	find
quick-find	N	N	1

Quick-find defect: **what is wrong with it?**

Quick-find is too slow

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

algorithm	init	union	find
quick-find	N	N	1

Quick-find defect.

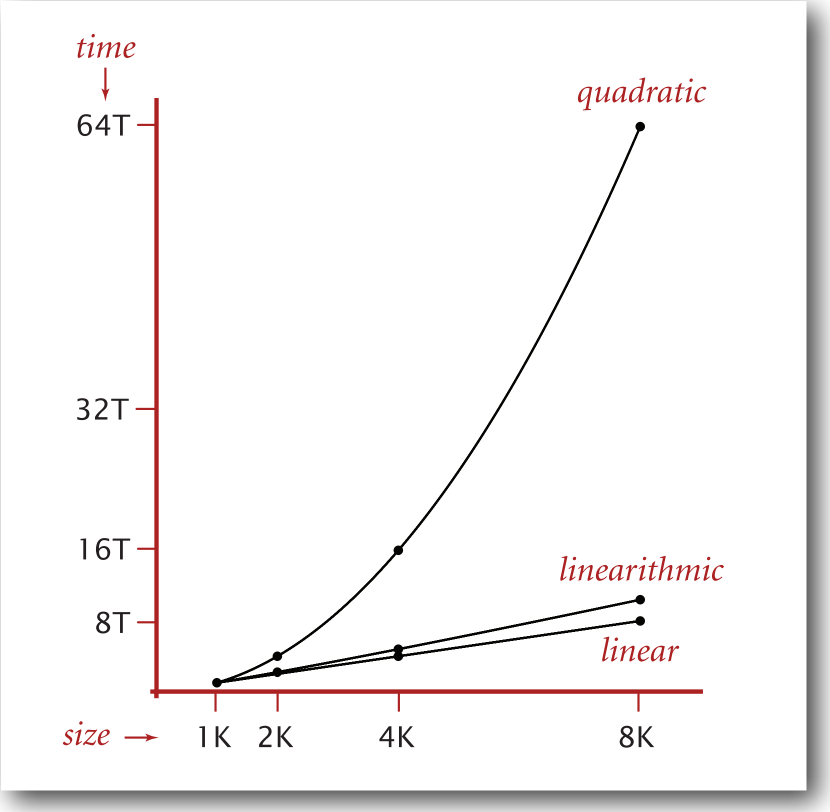
- Union too expensive.
- Ex. Takes N^2 array accesses to process sequence of N union commands on N objects. This is a **quadratic** algorithm.

Quadratic algorithms do not scale

Rough standard (for now).

true since 1950

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



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!

Take home message

Steps to developing a usable algorithm.

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

Take home message

Steps to developing a usable algorithm.

- ✓ Model the problem.
- ✓ Find an algorithm (and data structure) to solve it.
- ✓ Fast enough? NO.
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

Will continue this example after we cover analysis methods and additional data structures