Computer Science 226

Algorithms and Data Structures

Fall 2007

Instructors:
Bob Sedgewick
Kevin Wayne

Course Overview

- outline
- why study algorithms?
- usual suspects
- coursework
- resources (web)
- resources (books)

COS 226 course overview

What is COS 226?

- Intermediate-level survey course.
- Programming and problem solving with applications.
- Algorithm: method for solving a problem.
- Data structure: method to store information.

Topic	Data Structures and Algorithms
data types	stack, queue, list, union-find, priority queue
sorting	quicksort, mergesort, heapsort, radix sorts
searching	hash table, BST, red-black tree, B-tree
graphs	BFS, DFS, Prim, Kruskal, Dijkstra
strings	KMP, Rabin-Karp, TST, Huffman, LZW
geometry	Graham scan, k-d tree, Voronoi diagram

Why study algorithms?

Their impact is broad and far-reaching

Internet. Web search, packet routing, distributed file sharing.

Biology. Human genome project, protein folding.

Computers. Circuit layout, file system, compilers.

Computer graphics. Movies, video games, virtual reality.

Security. Cell phones, e-commerce, voting machines.

Multimedia. CD player, DVD, MP3, JPG, DivX, HDTV.

Transportation. Airline crew scheduling, map routing.

Physics. N-body simulation, particle collision simulation.

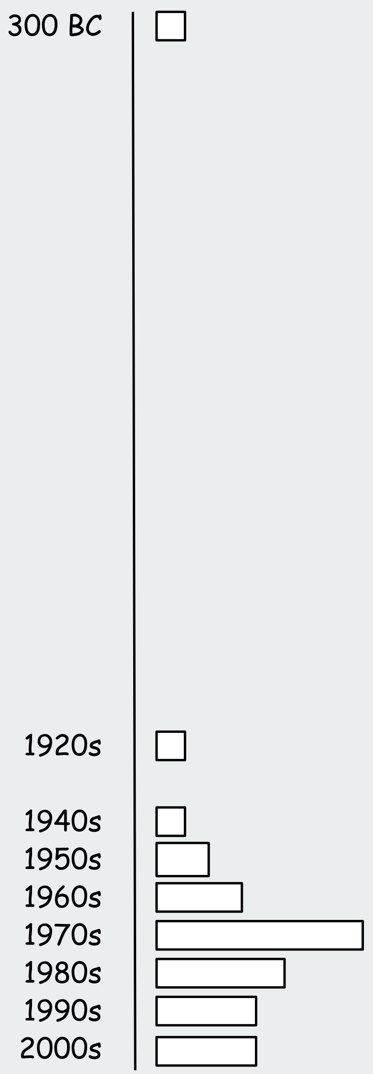
…

Why study algorithms?

Old roots, new opportunities

Study of algorithms dates at least to Euclid

Some important algorithms were discovered by undergraduates!

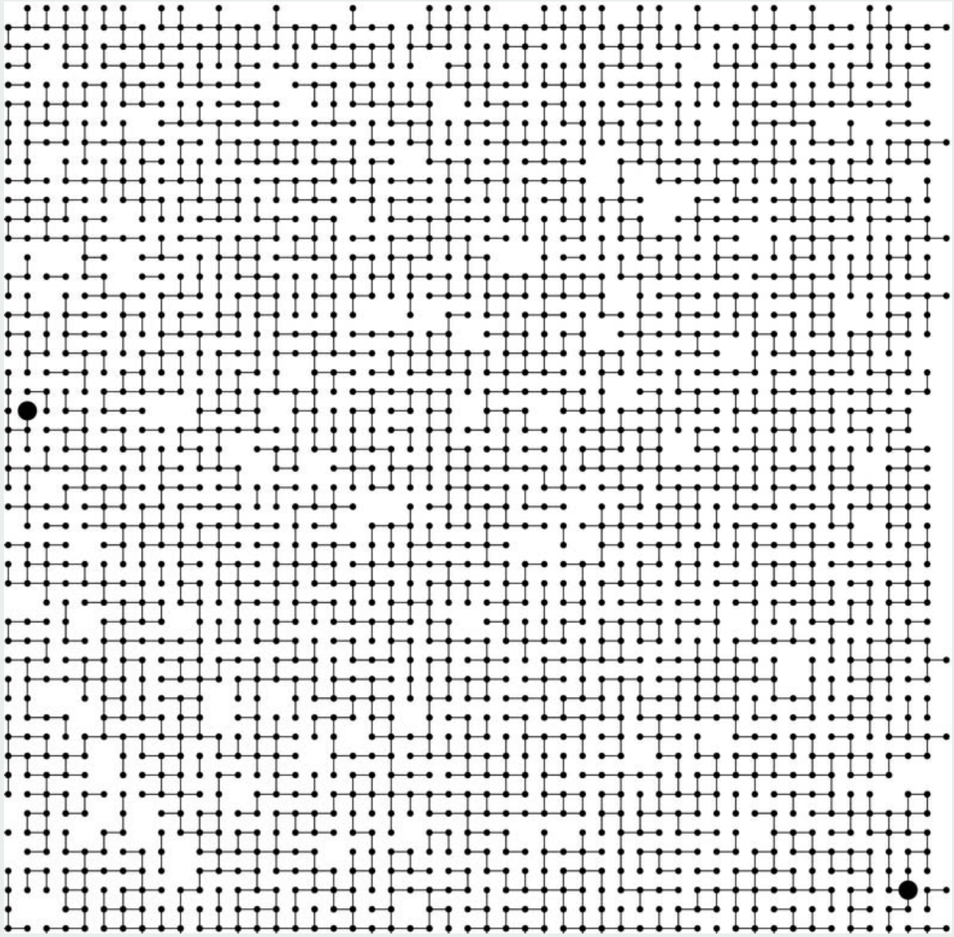


Why study algorithms?

To be able solve problems that could not otherwise be addressed

Example: Network connectivity

[stay tuned]



Why study algorithms?

For intellectual stimulation

> For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing. - Francis Sullivan

> An algorithm must be seen to be believed. - D. E. Knuth

Why study algorithms?

They may unlock the secrets of life and of the universe.

Computational models are replacing mathematical models in scientific enquiry

$$E = mc^2$$

$$F = ma$$

$$F = \frac{G m_1 m_2}{r^2}$$

$$\left[-\frac{\hbar^2}{2m} \nabla^2 + V(r) \right] \Psi(r) = E \, \Psi(r)$$

20th century science
(formula based)

```
for (double t = 0.0; true; t = t + dt)
   for (int i = 0; i < N; i++)
   {
      bodies[i].resetForce();
      for (int j = 0; j < N; j++)
         if (i != j)
            bodies[i].addForce(bodies[j]);
   }
```

21st century science
(algorithm based)

> Algorithms: a common language for nature, human, and computer.
> - Avi Wigderson

Why study algorithms?

For fun and profit

Nintendo

Adobe

Google™

DE Shaw & Co

Morgan Stanley

Microsoft

YAHOO!

PIXAR

Why study algorithms?

- Their impact is broad and far-reaching
- Old roots, new opportunities
- To be able to solve problems that could not otherwise be addressed
- For intellectual stimulation
- They may unlock the secrets of life and of the universe
- For fun and profit

Why study anything else?

The Usual Suspects

Lectures: Bob Sedgewick

- TTh 11-12:20, Bowen 222
- Office hours T 3-5 at Cafe Viv in Frist

Course management (everything else): Kevin Wayne

Precepts: Kevin Wayne

- Thursdays.
  - 1: 12:30 Friend 110
  - 2: 3:30 Friend 109
- Discuss programming assignments, exercises, lecture material.
- First precept meets Thursday 9/20
- Kevin's office hours TBA

Need a precept time? Need to change precepts?

- email Donna O'Leary (CS ugrad coordinator) doleary@cs.princeton.edu

Check course web page for up-to-date info

Coursework and Grading

7-8 programming assignments. 45%

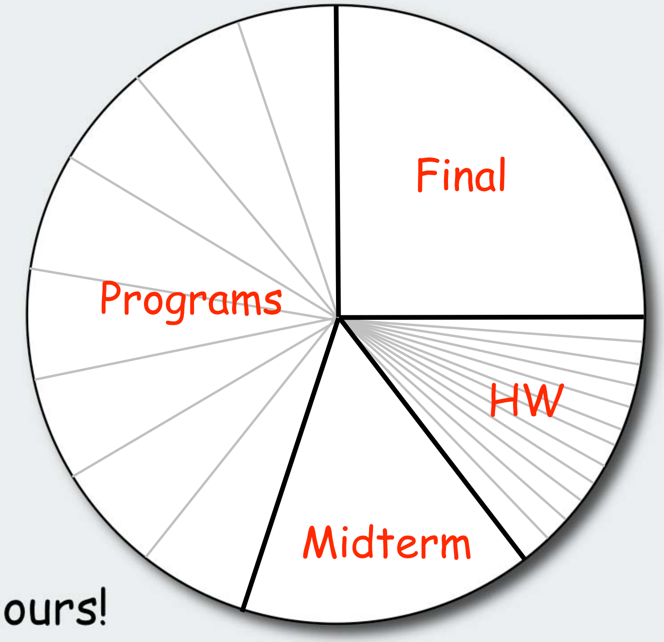
- Due 11:55pm, starting Monday 9/24.
- Available via course website.

Weekly written exercises. 15%

- Due at beginning of Wednesday lecture, starting 9/24.
- Available via course website.

Exams.

- Closed-book with cheatsheet.
- Midterm. 15%
- Final. 25%



Staff discretion. Adjust borderline cases.

- Participation in lecture and precepts
- Everyone needs to meet us both at office hours!

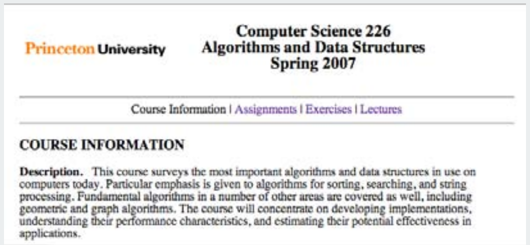
Challenge for the bored. Determine importance of 45-15-15-25 weights

Resources (web)

Course content.

`http://www.princeton.edu/~cos226`

- syllabus
- exercises
- lecture slides
- programming assignments (description, code, test data, checklists)



Princeton University

Computer Science 226
Algorithms and Data Structures
Spring 2007

Course Information | Assignments | Exercises | Lectures

COURSE INFORMATION

Description. This course surveys the most important algorithms and data structures in use on computers today. Particular emphasis is given to algorithms for sorting, searching, and string processing. Fundamental algorithms in a number of other areas are covered as well, including geometric and graph algorithms. The course will concentrate on developing implementations, understanding their performance characteristics, and estimating their potential effectiveness in applications.

Course administration.

`https://moodle.cs.princeton.edu/course/view.php?id=24`

- programming assignment submission.
- grades.

COS226, Spring 2007: Algorithms and Data Structures

You are currently using guest access (Login)

CS » COS226-S07

Content

Course information
Assignments
Exercises
Lectures
Exams
Intro to CS booksite
Intro to Algs booksite

Social Activities

Forums
Announcements
Assignment Forum
General Questions
Queues
Shellsort
Patterns
Bins
Markov
Burrows
Location
Wordnet
Exercises 1
Exercises 2
Exercises 3

Announcements — Subscribe to this forum

precepts & office hours
by David Walker - Saturday, 3 February 2007, 02:28 PM

Here are the precepts & office hours. You can always contact one of the TAs or professors to set up additional appointments.

Mohammad Mahmoody Ghidary,
Precept 03: Tuesday 2:30, CS 102
Office Hours:
Wednesday 12:20-12:50 (After Class)
Thursday 2:30-3

Jimin Song
Precept 01: Tuesday 12:30, Friend 109
Office Hours:
Wednesday 10:30-11:00 (Before Class)
Thursday 2:00-2:30

David Walker
Precept 01A: Tuesday 12:30, CS 102
Precept 02: Tuesday 1:30, CS 102
Monday 12:20-12:50 (After Class)
Thursday 11:30-12

Calendar

< February 2007 >

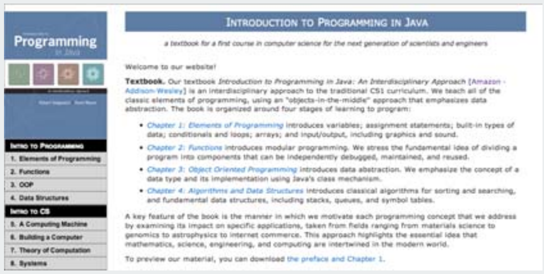
Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28			

Global events — Course events

Booksites.

```
http://www.cs.princeton.edu/IntroCS
http://www.cs.princeton.edu/IntroAlgsDS
```

- brief summary of content.
- code.
- links to web content.



INTRO TO PROGRAMMING
1. Elements of Programming
2. Functions
3. OOP
4. Data Structures
INTRO TO CS
5. A Computing Machine
6. Building a Computer
7. Theory of Computation
8. Systems

INTRODUCTION TO PROGRAMMING IN JAVA

a textbook for a first course in computer science for the next generation of scientists and engineers

Welcome to our website!

Textbook. Our textbook *Introduction to Programming in Java: An Interdisciplinary Approach* [Amazon · Addison-Wesley] is an interdisciplinary approach to the traditional CS1 curriculum. We teach all of the classic elements of programming, using an "objects-in-the-middle" approach that emphasizes data abstraction. The book is organized around four stages of learning to program:

- *Chapter 1: Elements of Programming* introduces variables; assignment statements; built-in types of data; conditionals and loops; arrays; and input/output, including graphics and sound.
- *Chapter 2: Functions* introduces modular programming. We stress the fundamental idea of dividing a program into components that can be independently debugged, maintained, and reused.
- *Chapter 3: Object Oriented Programming* introduces data abstraction. We emphasize the concept of a data type and its implementation using Java's class mechanism.
- *Chapter 4: Algorithms and Data Structures* introduces classical algorithms for sorting and searching, and fundamental data structures, including stacks, queues, and symbol tables.

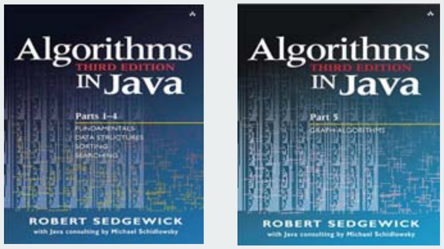
A key feature of the book is the manner in which we motivate each programming concept that we address by examining its impact on specific applications, taken from fields ranging from materials science to genomics to astrophysics to internet commerce. This approach highlights the essential idea that mathematics, science, engineering, and computing are intertwined in the modern world.

To preview our material, you can download the preface and Chapter 1.

Resources (books)

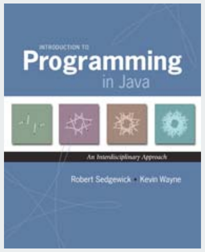
Algorithms in Java, 3rd edition

- Parts 1-4. [sorting, searching]
- Part 5. [graph algorithms]



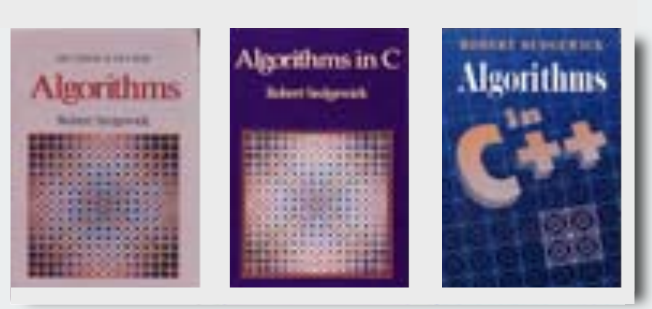
Introduction to Programming in Java

- basic programming model
- elementary AofA and data structures



Algorithms in Pascal(!)/C/C++, 2nd edition

- strings
- elementary geometric algorithms



Algorithms, 4th edition

(in preparation)

Union-Find

Union-Find Algorithms

- network connectivity
- quick find
- quick union
- improvements
- applications

Subtext of today's lecture (and this course)

Steps to developing a usable algorithm.

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

The scientific method

Mathematical models and computational complexity

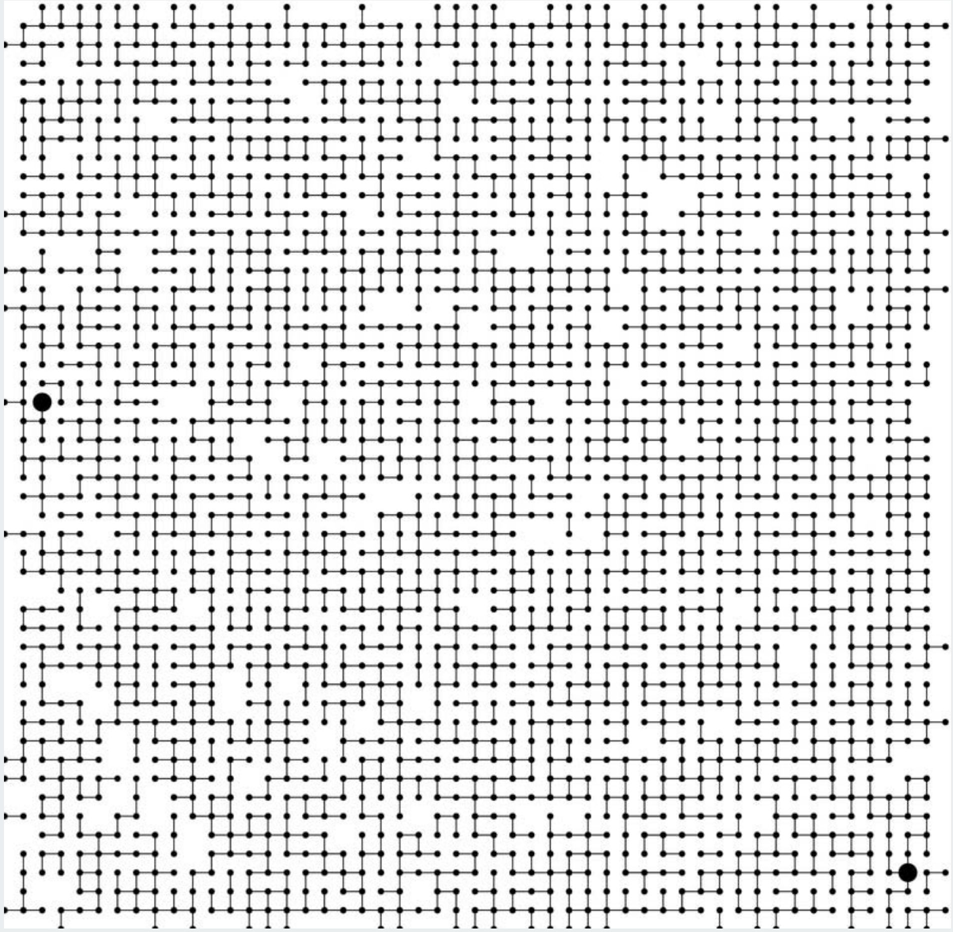
READ Chapter One of Algs in Java

▸ network connectivity
▸ quick find
▸ quick union
▸ improvements
▸ applications

Network connectivity

Basic abstractions

- set of objects
- union command: connect two objects
- find query: is there a path connecting one object to another?



Objects

Union-find applications involve manipulating objects of all types.

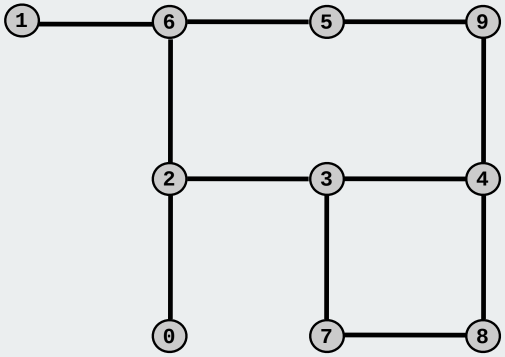
- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Variable name aliases.
- Pixels in a digital photo.
- Metallic sites in a composite system.

stay tuned

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

- Hide details not relevant to union-find.
- Integers allow quick access to object-related info.
- Could use symbol table to translate from object names

use as array index



Union-find abstractions

Simple model captures the essential nature of connectivity.

- Objects.

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

grid points

- Disjoint sets of objects.

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

subsets of connected grid points

- Find query: are objects `2` and `9` in the same set?

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

are two grid points connected?

- Union command: merge sets containing `3` and `8`.

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

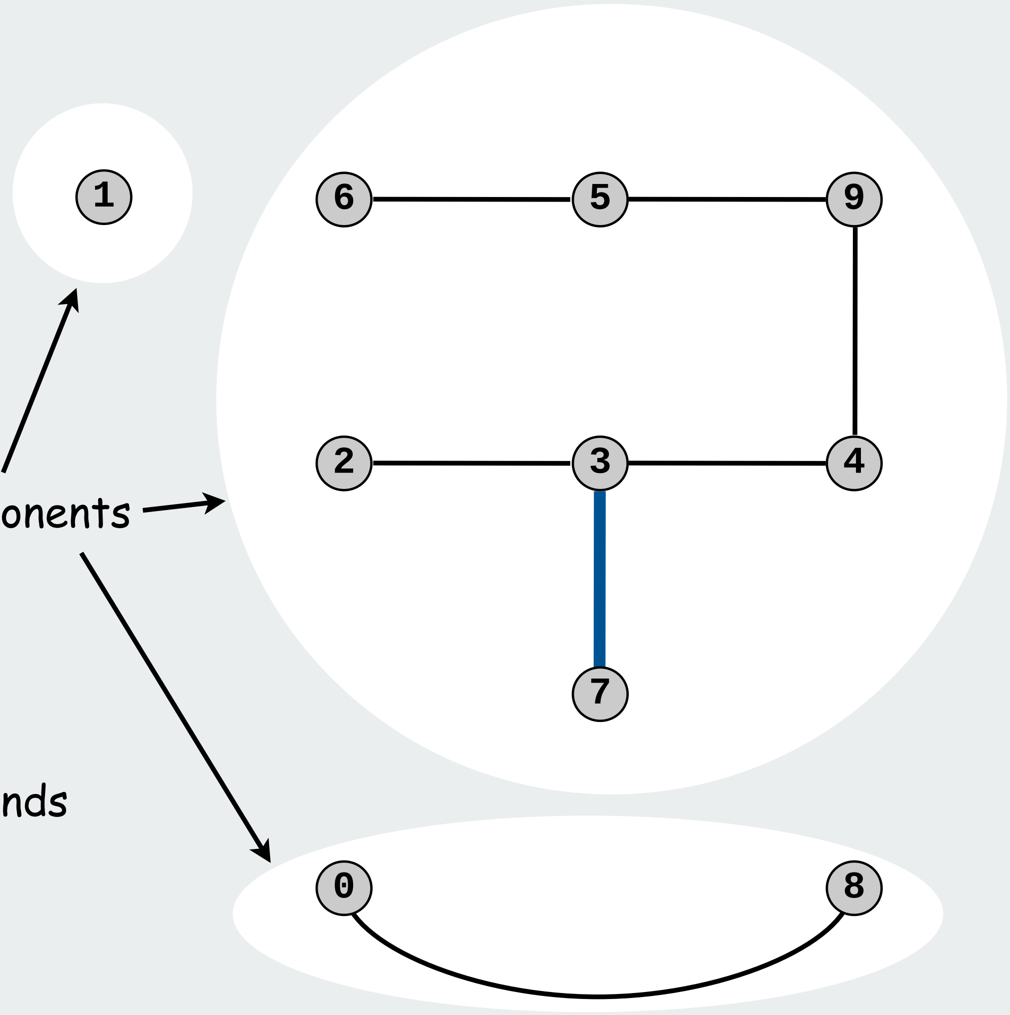
add a connection between
two grid points

Connected components

Connected component: set of mutually connected vertices

Each union command reduces by 1 the number of components

```
in      out

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

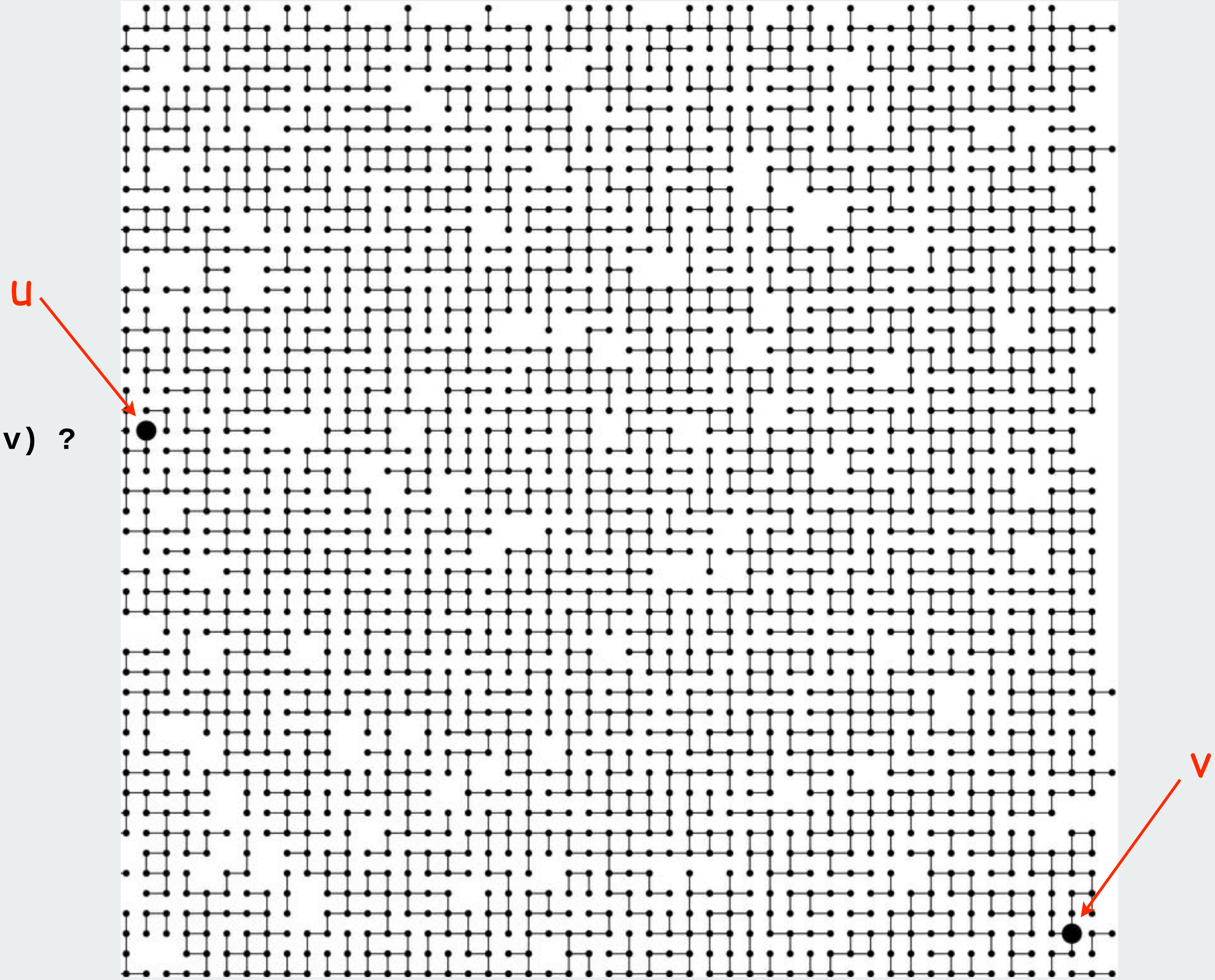


3 = 10-7 components

7 union commands

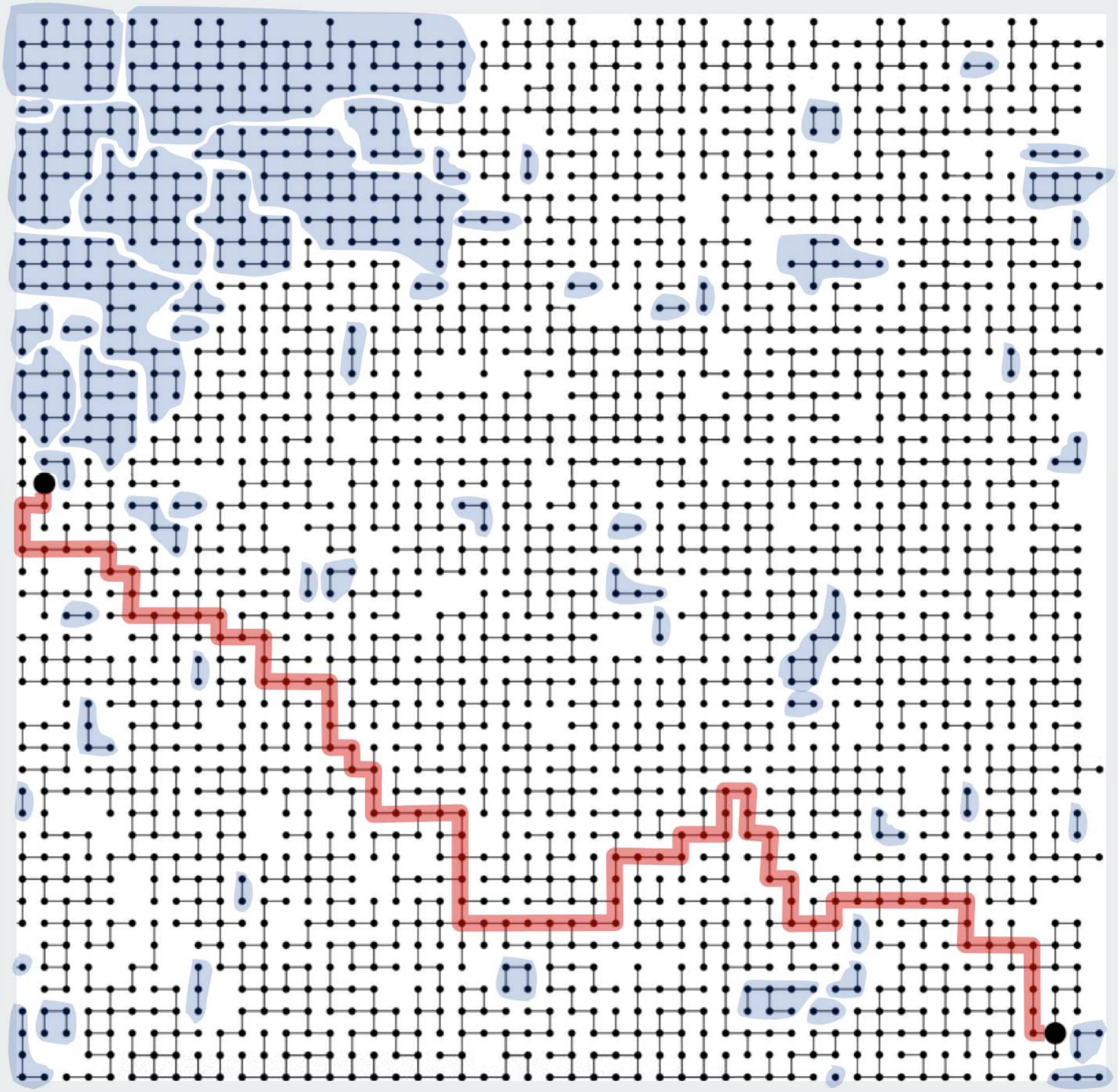
Network connectivity: larger example

```
find(u, v) ?
```



Network connectivity: larger example

```
find(u, v) ?
```



Union-find abstractions

- Objects.
- Disjoint sets of objects.
- Find queries: are two objects in the same set?
- Union commands: replace sets containing two items by their union

Goal. Design efficient data structure for union-find.

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

- network 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

Union. To merge components 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

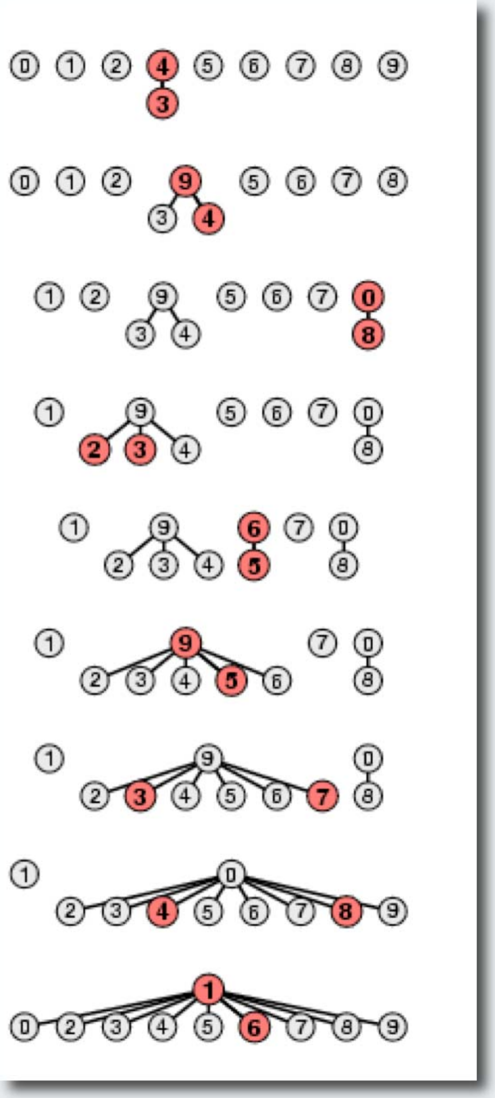
union of 3 and 6
2, 3, 4, 5, 6, and 9 are connected

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;
   }

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

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

set id of each
object to itself

1 operation

N operations

Quick-find is too slow

Quick-find algorithm may take ~MN steps
to process M union commands on N objects

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^{10} edges connecting 10^9 nodes.
- Quick-find takes more than 10^{19} operations.
- 300+ 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!

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

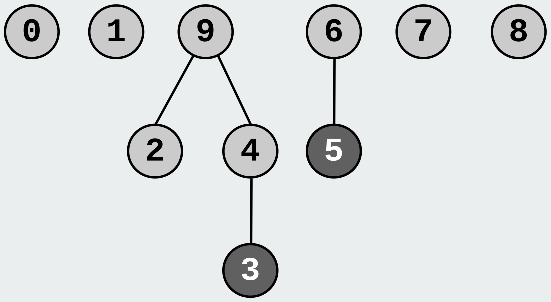
Quick-union [lazy approach]

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

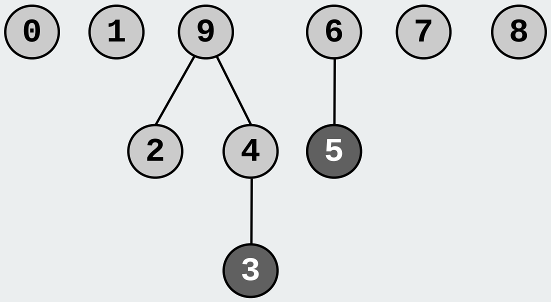
Quick-union [lazy approach]

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



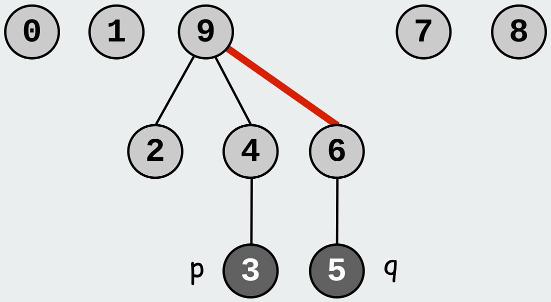
3's root is 9; 5's root is 6
3 and 5 are not connected

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

Union. Set the id of q's root to the id of p's root.

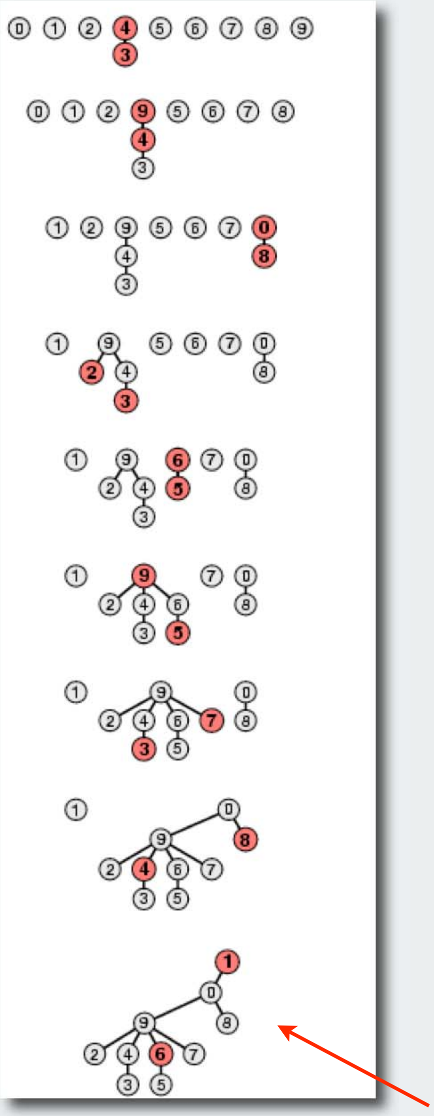
i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	9	7	8	9

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


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

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

time proportional
to depth of i

time proportional
to depth of p and q

time proportional
to depth of p and q

Quick-union is also too slow

Quick-find defect.

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

Quick-union defect.

- Trees can get tall.
- Find too expensive (could be N steps)
- Need to do find to do union

algorithm	union	find
Quick-find	N	1
Quick-union	N*	N

← worst case

* includes cost of find

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

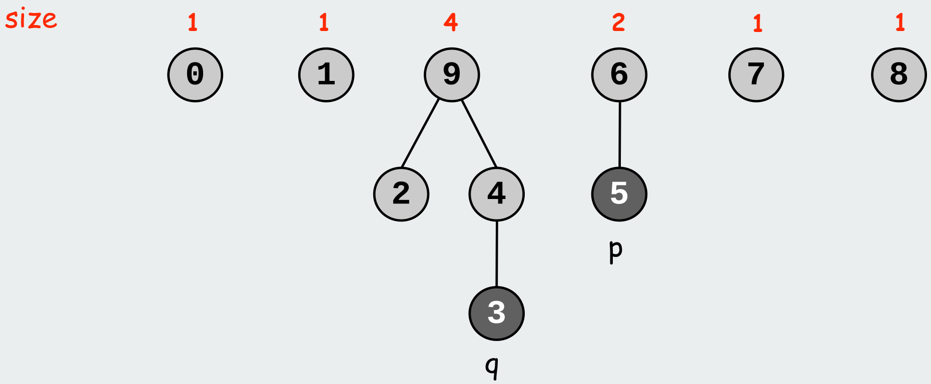
Improvement 1: Weighting

Weighted quick-union.

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

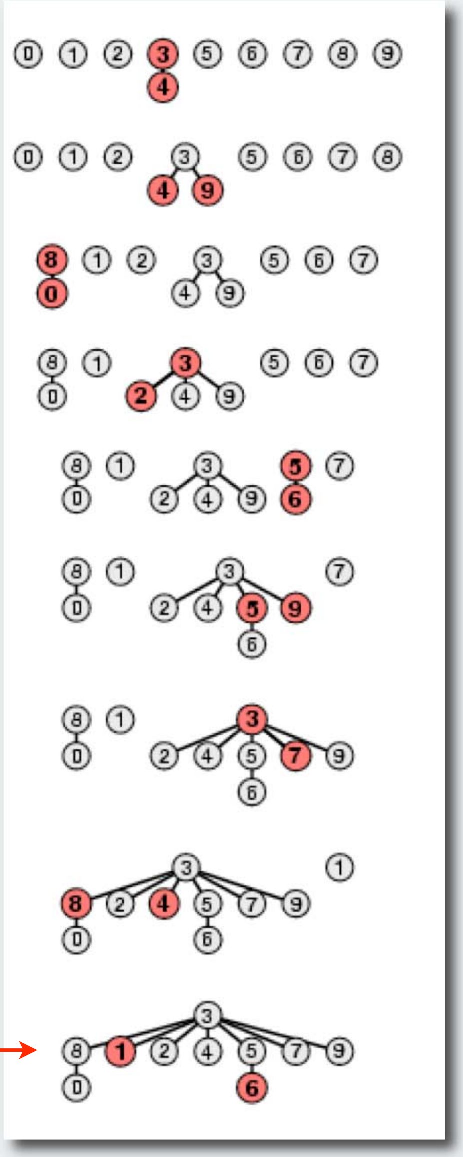
Ex. Union of 5 and 3.

- 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

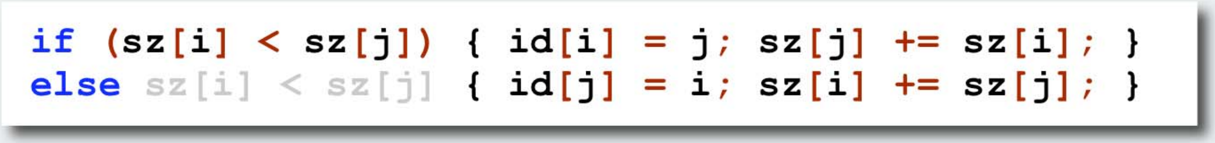
Java implementation.

- Almost identical to quick-union.
- Maintain extra array sz[] to count number of elements in the tree rooted at i.

Find. Identical to quick-union.

Union. Modify quick-union to

- merge smaller tree into larger tree
- update the sz[] array.



```
if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else sz[i] < sz[j] { 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.
- Fact: depth is at most lg N. [needs proof]

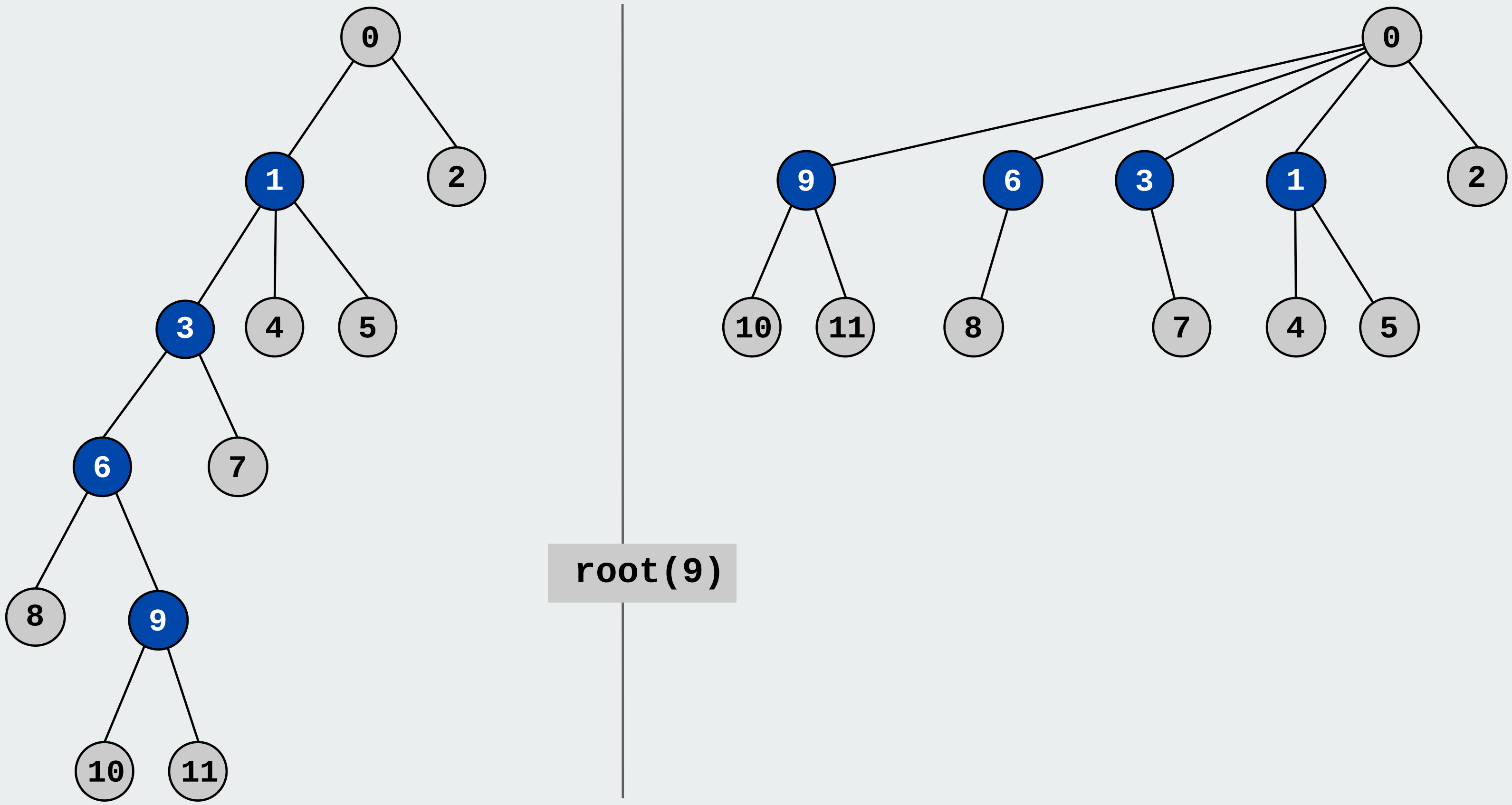
Data Structure	Union	Find
Quick-find	N	1
Quick-union	N *	N
Weighted QU	lg N *	lg N

* includes cost of find

Stop at guaranteed acceptable performance? No, easy to improve further.

Improvement 2: Path compression

Path compression. Just after computing the root of `i`, set the `id` of each examined node to `root(i)`.



Weighted quick-union with path compression

Path compression.

- Standard implementation: add second loop to `root()` to set the id of each examined node to the root.
- Simpler one-pass variant: make every other node in path point to its grandparent.

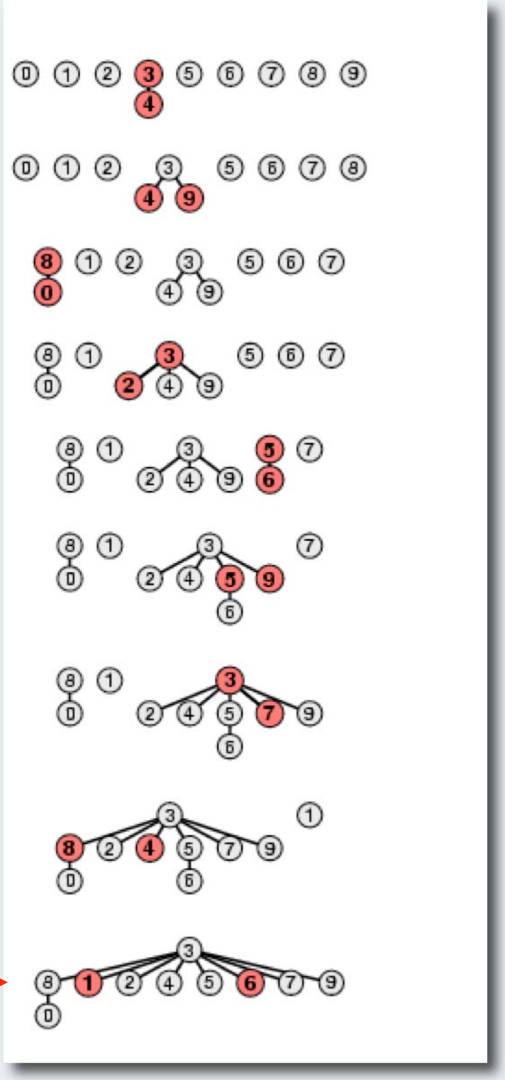
```
public int root(int i)
{
   while (i != id[i])
   {
      id[i] = id[id[i]];
      i = id[i];
   }
   return i;
}
```

only one extra line of code !

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

Weighted quick-union with path compression

```
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

Theorem. Starting from an empty data structure, any sequence of M union and find operations on N objects takes O(N + M lg* N) time.

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

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

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
265536	5

Amazing fact:

- In theory, no linear linking strategy exists

Summary

Algorithm	Worst-case time
Quick-find	M N
Quick-union	M N
Weighted QU	N + M log N
Path compression	N + M log N
Weighted + path	(M + N) lg* N

M union-find ops on a set of N objects

Ex. Huge practical problem.

- 10^{10} edges connecting 10^9 nodes.
- WQUPC reduces time from 3,000 years to 1 minute.
- Supercomputer won't help much.
- Good algorithm makes solution possible.

WQUPC on Java cell phone beats QF on supercomputer!

Bottom line.

WQUPC makes it possible to solve problems that could not otherwise be addressed

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

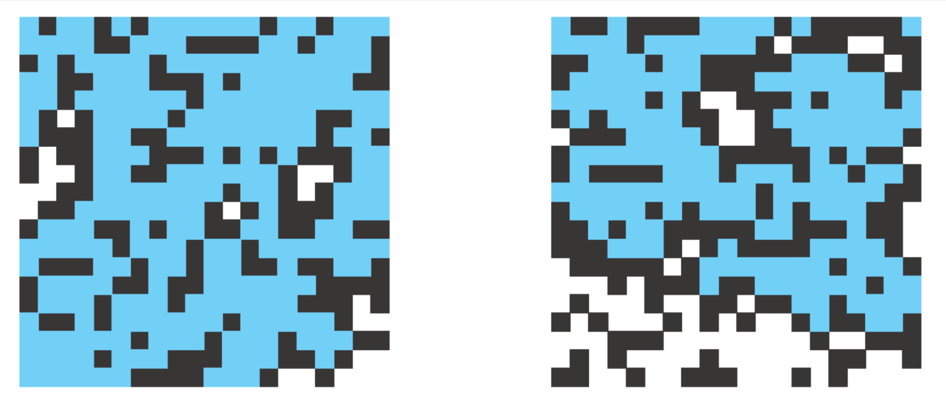
Union-find applications

✓ Network connectivity.
- Percolation.
- Image processing.
- Least common ancestor.
- Equivalence of finite state automata.
- Hinley-Milner polymorphic type inference.
- Kruskal's minimum spanning tree algorithm.
- Games (Go, Hex)
- Compiling equivalence statements in Fortran.

Percolation

A model for many physical systems

- N-by-N grid.
- Each square is vacant or occupied.
- Grid percolates if top and bottom are connected by vacant squares.



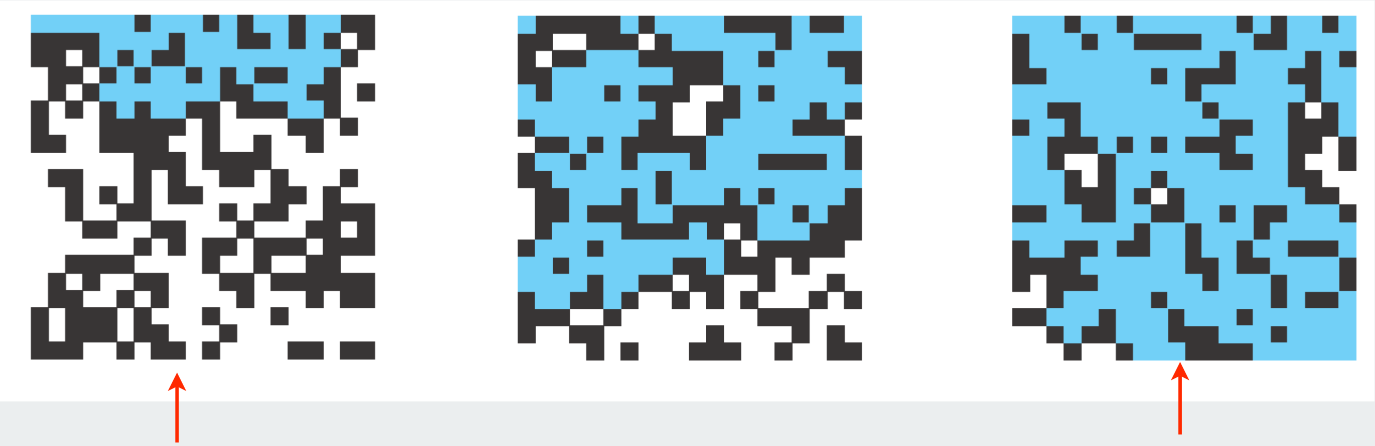
percolates

does not percolate

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

Percolation phase transition

Likelihood of percolation depends on site vacancy probability p



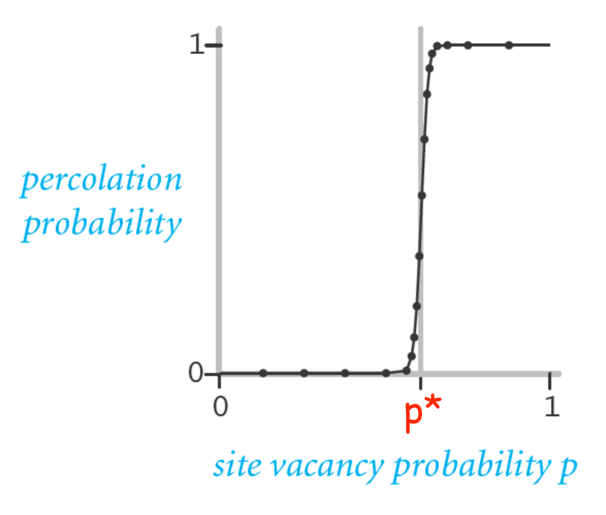
p low: does not percolate

p high: percolates

Experiments show a threshold p*

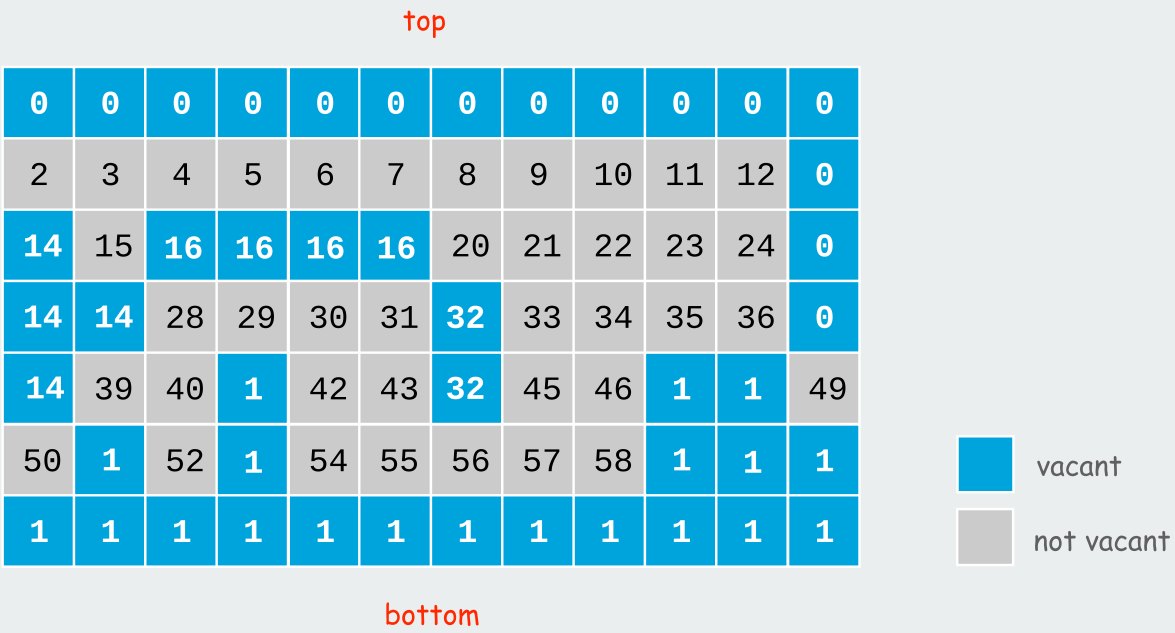
- p > p*: almost certainly percolates
- p < p*: almost certainly does not percolate

Q. What is the value of p* ?



UF solution to find percolation threshold

- Initialize whole grid to be "not vacant"
- Implement "make site vacant" operation
  that does **`union()`** with adjacent sites
- Make all sites on top and bottom rows vacant
- Make random sites vacant until **`find(top, bottom)`**
- Vacancy percentage estimates p*



top

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
2	3	4	5	6	7	8	9	10	11	12	**0**
14	15	**16**	**16**	**16**	**16**	20	21	22	23	24	**0**
14	**14**	28	29	30	31	**32**	33	34	35	36	**0**
14	39	40	**1**	42	43	**32**	45	46	**1**	**1**	49
50	**1**	52	**1**	54	55	56	57	58	**1**	**1**	**1**
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

bottom

🟦 vacant

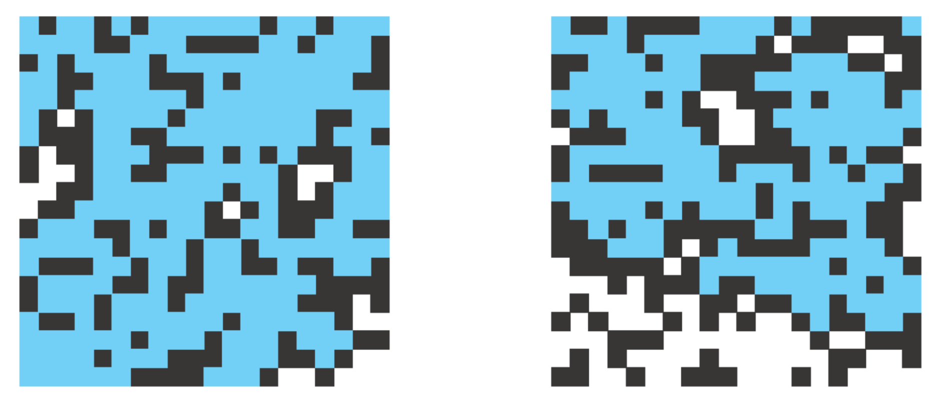
⬜ not vacant

Percolation

Q. What is percolation threshold p* ?

A. about 0.592746 for large square lattices.

percolation constant known only via simulation



percolates

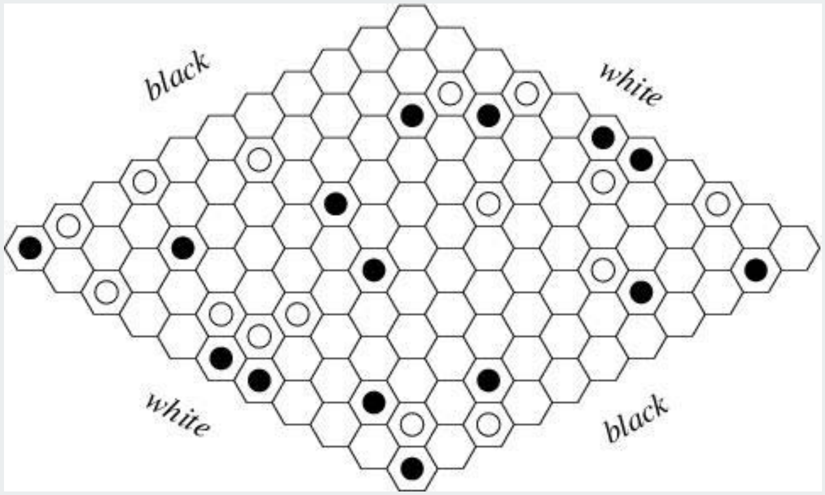
does not percolate

Q. Why is UF solution better than solution in IntroProgramming 2.4?

Hex

Hex. [Piet Hein 1942, John Nash 1948, Parker Brothers 1962]

- Two players alternate in picking a cell in a hex grid.
- Black: make a black path from upper left to lower right.
- White: make a white path from lower left to upper right.



Reference: http://mathworld.wolfram.com/GameofHex.html

Union-find application. Algorithm to detect when a player has won.

Subtext of today's lecture (and this course)

Steps to developing an usable algorithm.

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

The scientific method

Mathematical models and computational complexity

READ Chapter One of Algs in Java

Collaboration policy

Programs: Do not use someone else's code unless specifically authorized

Exceptions

- Code from course materials OK [cite source]
- Coding with partner OK after first assignment [stay tuned]

Where to get help

- Email (but no code in email)
- Office hours
- Lab TAs in Friend 008/009
- Bounce ideas (but not code) off classmates

Note: Programming in groups except as above is a serious violation.

Exercises: Write up your own solutions (no copying)

- working with classmates is encouraged
- checking solutions is OK

Stacks and Queues

- stacks
- dynamic resizing
- queues
- generics
- applications

Stacks and Queues

Fundamental data types.

- Values: sets of objects
- Operations: insert, remove, test if empty.
- Intent is clear when we insert.
- Which item do we remove?

LIFO = "last in first out"

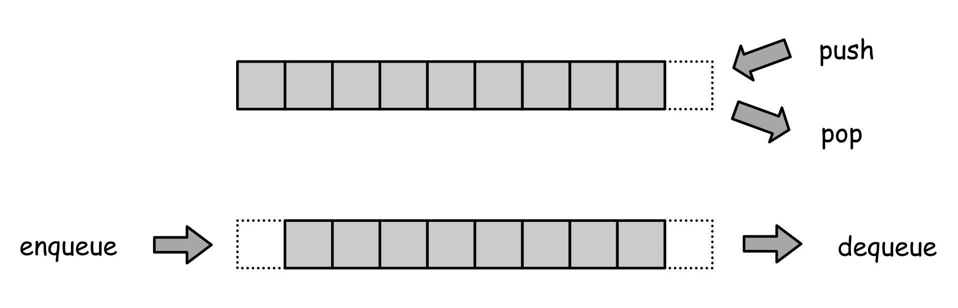
Stack.

- Remove the item most recently added.
- Analogy: cafeteria trays, Web surfing.

FIFO = "first in first out"

Queue.

- Remove the item least recently added.
- Analogy: Registrar's line.



Client, Implementation, Interface

Separate interface and implementation so as to:

- Build layers of abstraction.
- Reuse software.
- Ex: stack, queue, symbol table.

Interface: description of data type, basic operations.

Client: program using operations defined in interface.

Implementation: actual code implementing operations.

Client, Implementation, Interface

Benefits.

- Client can't know details of implementation ⇒ client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒ many clients can re-use the same implementation.
- Design: creates modular, re-usable libraries.
- Performance: use optimized implementation where it matters.

Interface: description of data type, basic operations.

Client: program using operations defined in interface.

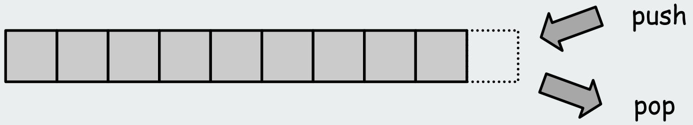
Implementation: actual code implementing operations.

- **stacks**
- dynamic resizing
- queues
- generics
- applications

Stacks

Stack operations.

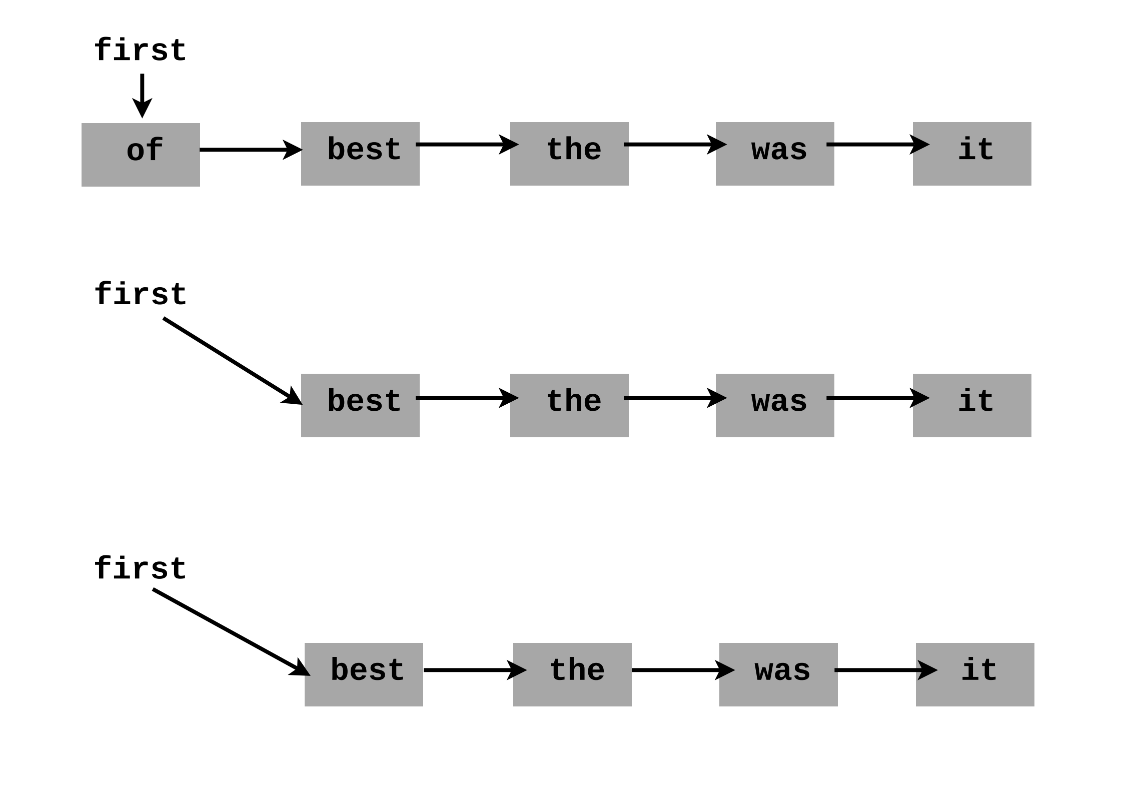
- `push()` Insert a new item onto stack.
- `pop()` Remove and return the item most recently added.
- `isEmpty()` Is the stack empty?



```
public static void main(String[] args)
{
   StackOfStrings stack = new StackOfStrings();
   while(!StdIn.isEmpty())
   {
      String s = StdIn.readString();
      stack.push(s);
   }
   while(!stack.isEmpty())
   {
      String s = stack.pop();
      StdOut.println(s);
   }
}
```

a sample stack client

Stack pop: Linked-list implementation

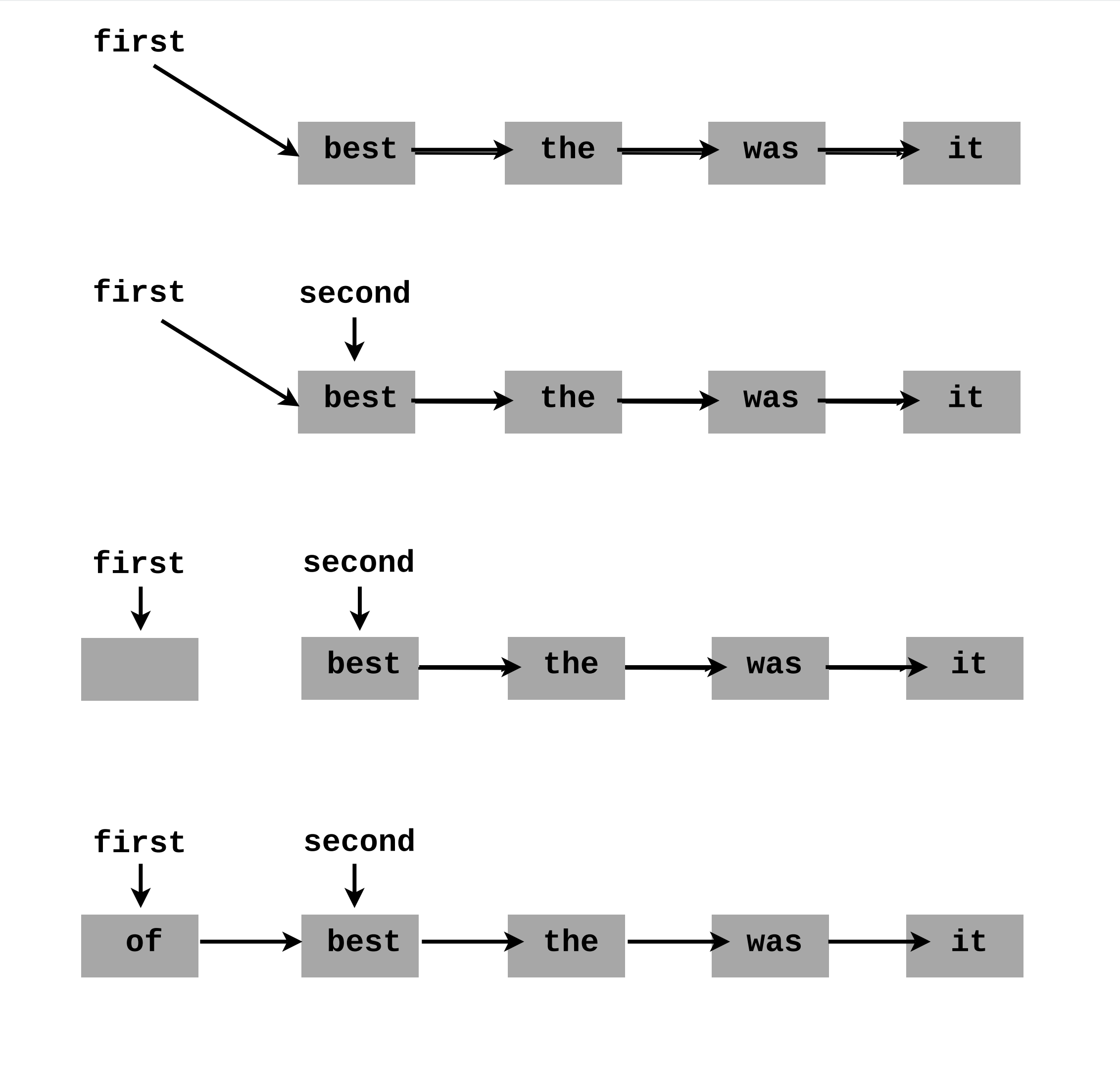


```
item = first.item;
```

```
first = first.next;
```

```
return item;
```

Stack push: Linked-list implementation



```
second = first;
```

```
first = new Node();
```

```
first.item = item;
first.next = second;
```

Stack: Linked-list implementation

```
public class StackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

"inner class"

Error conditions?

Example: pop() an empty stack

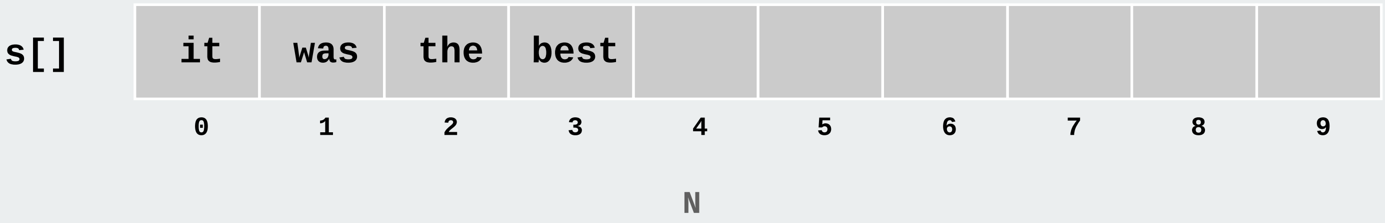
COS 217: bulletproof the code

COS 226: first find the code we want to use

Stack: Array implementation

Array implementation of a stack.

- Use array **`s[]`** to store **`N`** items on stack.
- **`push()`** add new item at **`s[N]`**.
- **`pop()`** remove item from **`s[N-1]`**.



s[]	it	was	the	best						
	0	1	2	3	4	5	6	7	8	9
					N					

Stack: Array implementation

```
public class StackOfStrings
{
   private String[] s;
   private int N = 0;

   public StringStack(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   { return N == 0; }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {
      String item = s[N-1];
      s[N-1] = null;
      N--;
      return item;
   }

}
```

avoid loitering
(garbage collector only reclaims memory
if no outstanding references)

- stacks
- dynamic resizing
- queues
- generics
- applications

Stack array implementation: Dynamic resizing

Q. How to grow array when capacity reached?
Q. How to shrink array (else it stays big even when stack is small)?

First try:

- **push()**: increase size of `s[]` by 1
- **pop()** : decrease size of `s[]` by 1

Too expensive

- Need to copy all of the elements to a new array.
- Inserting N elements: time proportional to $1 + 2 + \ldots + N \approx N^2/2$.

infeasible for large N

Need to guarantee that array resizing happens infrequently

Stack array implementation: Dynamic resizing

Q. How to grow array?

A. Use repeated doubling:

if array is full, create a new array of twice the size, and copy items

no-argument constructor

```
public StackOfStrings()
{ this(8); }

public void push(String item)
{
   if (N >= s.length) resize();
   s[N++] = item;
}

private void resize(int max)
{
   String[] dup = new String[max];
   for (int i = 0; i < N; i++)
      dup[i] = s[i];
   s = dup;
}
```

create new array copy items to it

Consequence. Inserting N items takes time proportional to N (not N^2).

$8 + 16 + \ldots + N/4 + N/2 + N \approx 2N$

Stack array implementation: Dynamic resizing

Q. How (and when) to shrink array?

How: create a new array of half the size, and copy items.

When (first try): array is half full?

No, causes thrashing

(push-pop-push-pop-... sequence: time proportional to N for each op)

When (solution): array is 1/4 full (then new array is half full).

```
public String pop(String item)
{
   String item = s[--N];
   sa[N] = null;
   if (N == s.length/4)
      resize(s.length/2);
   return item;
}
```

Not `a.length/2` to avoid thrashing

Consequences.

- any sequence of N ops takes time proportional to N
- array is always between 25% and 100% full

Stack Implementations: Array vs. Linked List

Stack implementation tradeoffs. Can implement with either array or linked list, and client can use interchangeably. Which is better?

Array.

- Most operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of N operations (starting from empty stack) takes time proportional to N.

"amortized" bound

Linked list.

- Grows and shrinks gracefully.
- Every operation takes constant time.
- Every operation uses extra space and time to deal with references.

Bottom line: tossup for stacks
but differences are significant when other operations are added

Stack implementations: Array vs. Linked list

Which implementation is more convenient?

	array?	linked list?
return count of elements in stack		
remove the kth most recently added		
sample a random element		

- stacks
- dynamic resizing
- **queues**
- generics
- applications

Queues

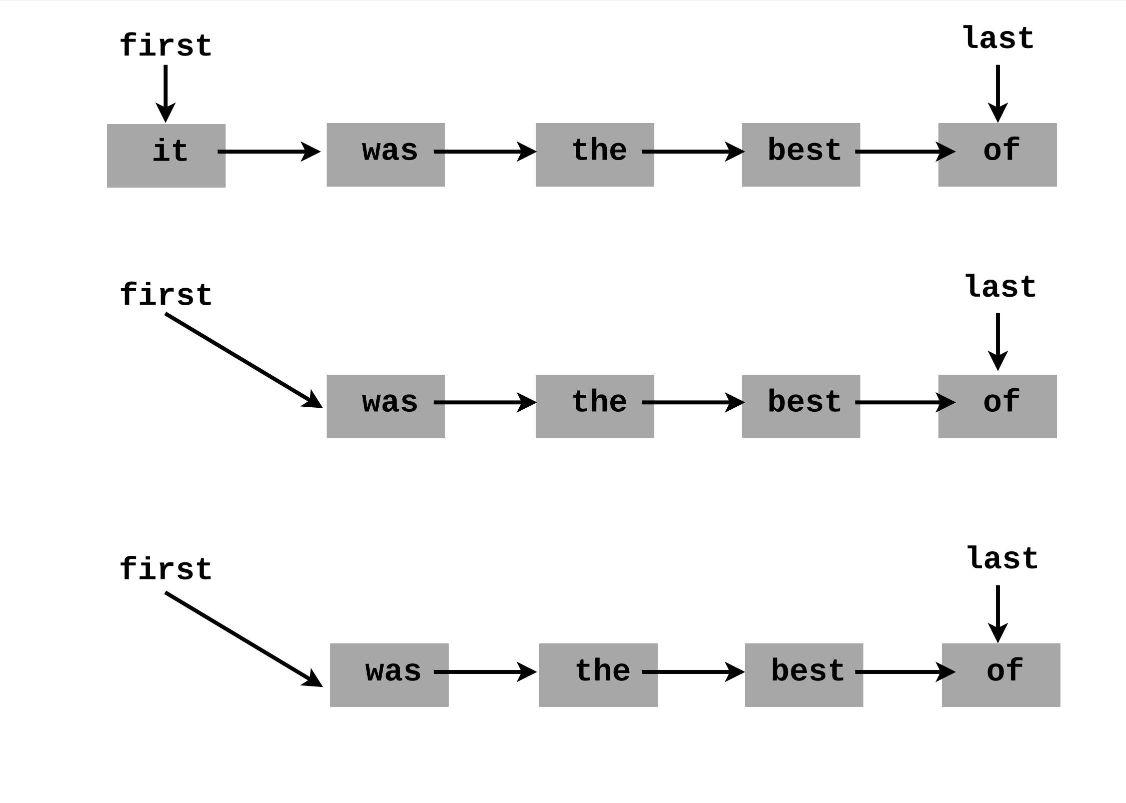
Queue operations.

- `enqueue()` Insert a new item onto queue.
- `dequeue()` Delete and return the item least recently added.
- `isEmpty()` Is the queue empty?

```
public static void main(String[] args)
{
   QueueOfStrings q = new QueueOfStrings();
   q.enqueue("Vertigo");
   q.enqueue("Just Lose It");
   q.enqueue("Pieces of Me");
   q.enqueue("Pieces of Me");
   System.out.println(q.dequeue());
   q.enqueue("Drop It Like It's Hot");

   while(!q.isEmpty()

      System.out.println(q.dequeue());
}
```

(CNN.COM GRPAHIC)

Dequeue: Linked List Implementation



```
item = first.item;
```

```
first = first.next;
```

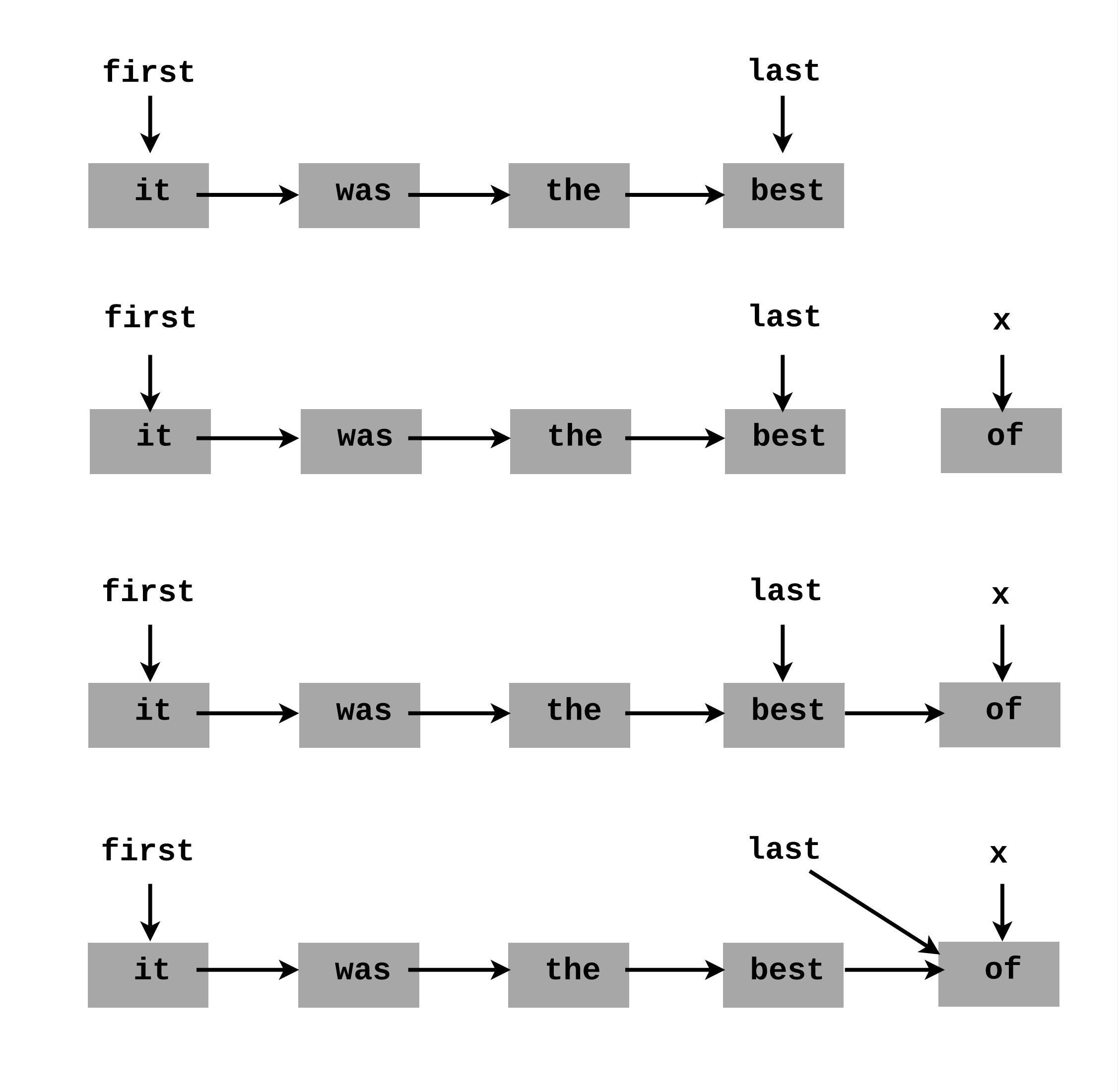
```
return item;
```

Aside:

dequeue (pronounced "DQ") means "remove from a queue"

deque (pronounced "deck") is a data structure (see PA 1)

Enqueue: Linked List Implementation



```
x = new Node();
x.item = item;
x.next = null;
```

```
last.next = x;
```

```
last = x;
```

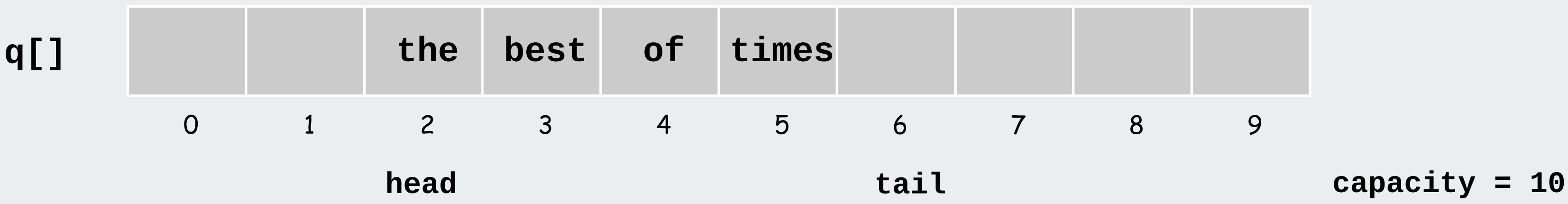
Queue: Linked List Implementation

```
public class QueueOfStrings
{
   private Node first;
   private Node last;

   private class Node
   { String item; Node next; }

   public boolean isEmpty()
   { return first == null; }

   public void enqueue(String item)
   {
      Node x = new Node();
      x.item = item;
      x.next = null;
      if (isEmpty()) { first     = x; last = x; }
      else           { last.next = x; last = x; }
   }

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      return item;
   }
}
```

Queue: Array implementation

Array implementation of a queue.

- Use array **q[]** to store items on queue.
- **enqueue()**: add new object at **q[tail]**.
- **dequeue()**: remove object from **q[head]**.
- Update **head** and **tail** modulo the **capacity**.



q[]			**the**	**best**	**of**	**times**				
	0	1	2	3	4	5	6	7	8	9
			head				**tail**			

capacity = 10

[details: good exercise or exam question]

- stacks
- dynamic resizing
- queues
- **generics**
- applications

Generics (parameterized data types)

We implemented: `StackOfStrings`, `QueueOfStrings`.

We also want: `StackOfURLs`, `QueueOfCustomers`, etc?

Attempt 1. Implement a separate stack class for each type.

- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

@#$*! most reasonable approach until Java 1.5 [hence, used in AlgsJava]

Stack of Objects

We implemented: **StackOfStrings**, **QueueOfStrings**.

We also want: **StackOfURLs**, **QueueOfCustomers**, etc?

Attempt 2. Implement a stack with items of type **Object**.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
Stack  s = new Stack();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

run-time error

Generics

Generics. Parameterize stack by a single type.

- Avoid casting in both client and implementation.
- Discover type mismatch errors at compile-time instead of run-time.

parameter

```
Stack<Apple> s = new Stack<Apple>();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

compile-time error

no cast needed in client

Guiding principles.

- Welcome compile-time errors
- Avoid run-time errors

Why?

Generic Stack: Linked List Implementation

```
public class StackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;   }

   public void push(String item)
   {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

```
public class Stack<Item>
{
   private Node first = null;

   private class Node
   {
      Item item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;   }

   public void push(Item item)
   {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
   }

   public Item pop()
   {
      Item item = first.item;
      first = first.next;
      return item;
   }
}
```

Generic type name

Generic stack: array implementation

The way it should be.

```
public class Stack<Item>
{
   private Item[] s;
   private int N = 0;

   public Stack(int cap)
   {  s = new Item[cap];   }

   public boolean isEmpty()
   { return N == 0; }

   public void push(Item item)
   {  s[N++] = item;   }

   public String pop()
   {
      Item item = s[N-1];
      s[N-1] = null;
      N--;
      return item;
   }

}
```

```
public class StackOfStrings
{
   private String[] s;
   private int N = 0;

   public StackOfStrings(int cap)
   {  s = new String[cap];   }

   public boolean isEmpty()
   { return N == 0; }

   public void push(String item)
   {  s[N++] = item;   }

   public String pop()
   {
      String item = s[N-1];
      s[N-1] = null;
      N--;
      return item;
   }

}
```

@#$*! generic array creation not allowed in Java

Generic stack: array implementation

The way it is: an ugly cast in the implementation.

```
public class Stack<Item>
{
   private Item[] s;
   private int N = 0;

   public Stack(int cap)
   { s = (Item[]) new Object[cap]; }

   public boolean isEmpty()
   { return N == 0; }

   public void push(Item item)
   {  s[N++] = item;   }

   public String pop()
   {
      Item item = s[N-1];
      s[N-1] = null;
      N--;
      return item;
   }

}
```

the ugly cast

Number of casts in good code: 0

Generic data types: autoboxing

Generic stack implementation is object-based.

What to do about primitive types?

Wrapper type.

- Each primitive type has a wrapper object type.
- Ex: **`Integer`** is wrapper type for **`int`**.

Autoboxing. Automatic cast between a primitive type and its wrapper.

Syntactic sugar. Behind-the-scenes casting.

```
Stack<Integer> s = new Stack<Integer>();
s.push(17);        // s.push(new Integer(17));
int a = s.pop();   // int a = ((int) s.pop()).intValue();
```

Bottom line: Client code can use generic stack for any type of data

- stacks
- dynamic resizing
- queues
- generics
- applications

Stack Applications

Real world applications.

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

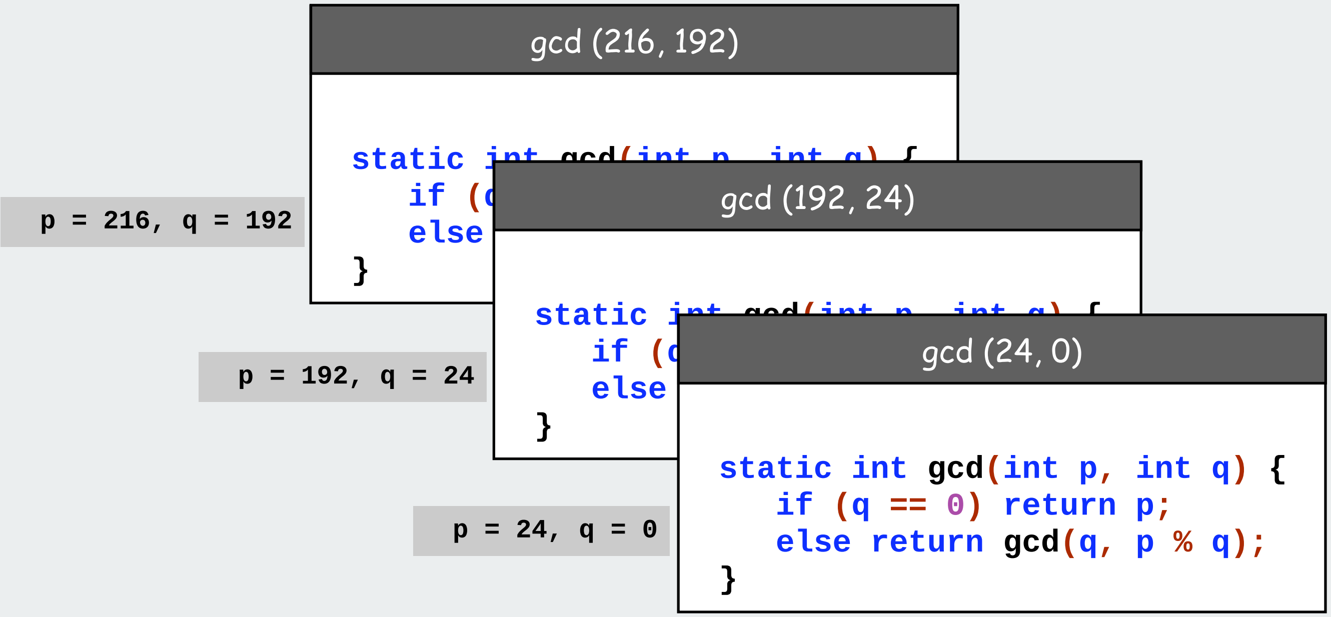
Function Calls

How a compiler implements functions.

- Function call: push local environment and return address.
- Return: pop return address and local environment.

Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.



```
static int gcd(int p, int q) {
   if (q == 0) return p;
   else return gcd(q, p % q);
}
```

Arithmetic Expression Evaluation

Goal. Evaluate infix expressions.

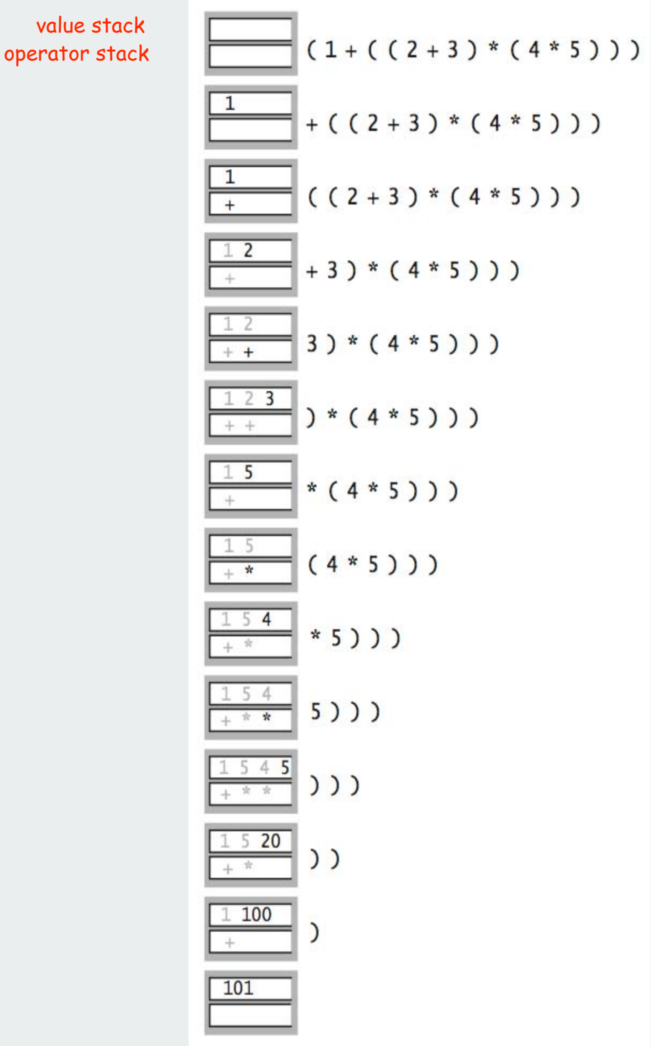
```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

operand

operator

Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parens: ignore.
- Right parens: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



Context. An interpreter!

Arithmetic Expression Evaluation

```
public class Evaluate {
   public static void main(String[] args) {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals(")")) {
            String op = ops.pop();
            if      (op.equals("+")) vals.push(vals.pop() + vals.pop());
            else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
         }
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Note: Old books have two-pass algorithm because generics were not available!

Correctness

Why correct?
When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

Extensions. More ops, precedence order, associativity.

```
1 + (2 - 3 - 4) * 5 * sqrt(6 + 7)
```

Stack-based programming languages

Observation 1.
Remarkably, the 2-stack algorithm computes the same value if the operator occurs after the two values.

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

Observation 2.
All of the parentheses are redundant!

```
1 2 3 + 4 5 * * +
```

Jan Lukasiewicz

Bottom line. Postfix or "reverse Polish" notation.

Applications. Postscript, Forth, calculators, Java virtual machine, …

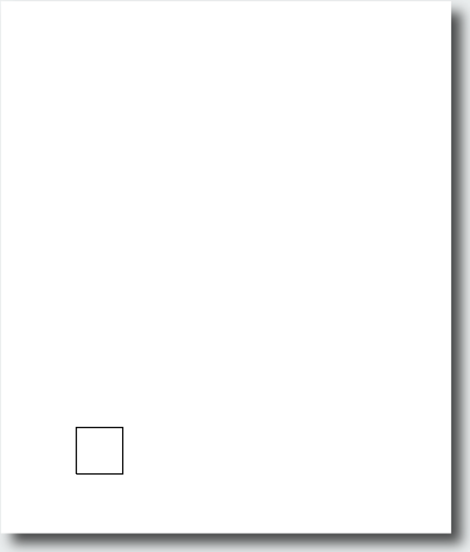
Stack-based programming languages: PostScript

Page description language

- explicit stack
- full computational model
- graphics engine

Basics

- %!: "I am a PostScript program"
- literal: "push me on the stack"
- function calls take args from stack
- turtle graphics built in



a PostScript program

```
%!
72 72 moveto
0 72 rlineto
72 0 rlineto
0 -72 rlineto
-72 0 rlineto
2 setlinewidth
stroke
```

Stack-based programming languages: PostScript

Data types

- basic: integer, floating point, boolean, ...
- graphics: font, path, ...
- full set of built-in operators

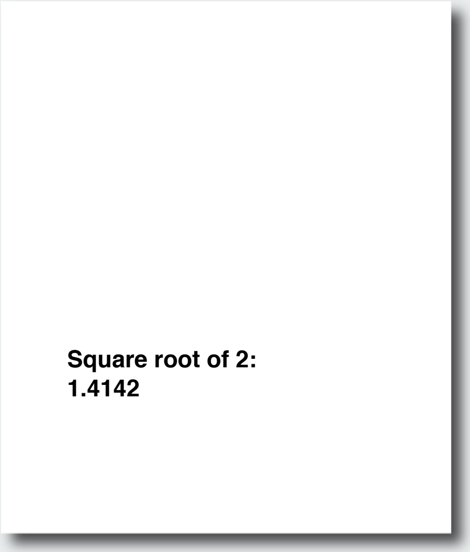
Text and strings

- full font support
- **show** (display a string, using current font)
- **cvs** (convert anything to a string)

like System.out.print()

like toString()

```
%!
/Helvetica-Bold findfont 16 scalefont setfont
72 168 moveto
(Square root of 2:) show
72 144 moveto
2 sqrt 10 string cvs show
```



Square root of 2:
1.4142

Stack-based programming languages: PostScript

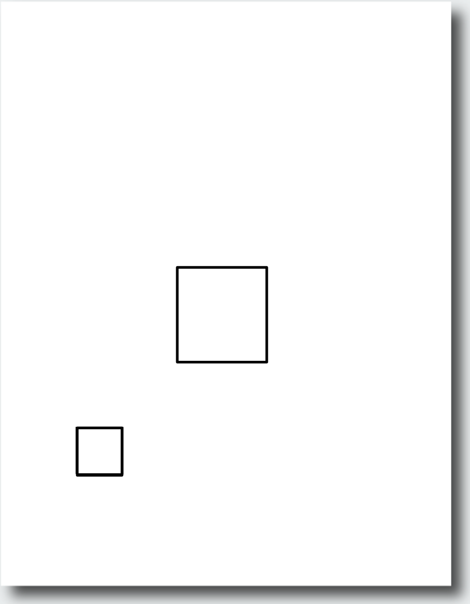
Variables (and functions)

- identifiers start with /
- def operator associates id with value
- braces
- args on stack

function definition

function calls

```
%!
/box
{
  /sz exch def
  0 sz rlineto
  sz 0 rlineto
  0 sz neg rlineto
  sz neg 0 rlineto
} def

72 144 moveto
72 box
288 288 moveto
144 box
2 setlinewidth
stroke
```

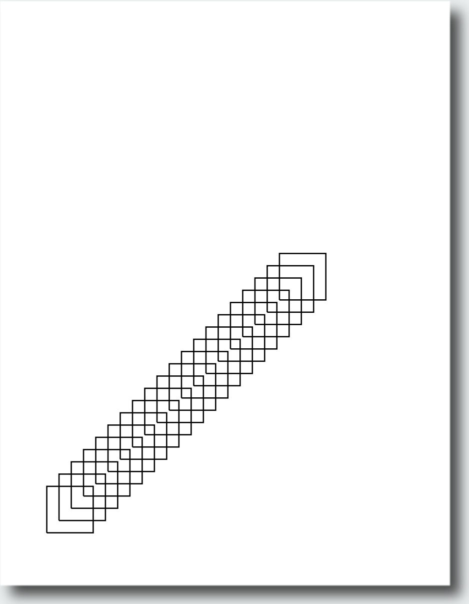


Stack-based programming languages: PostScript

for loop

- "from, increment, to" on stack
- loop body in braces
- **for** operator

```
1 1 20
{ 19 mul dup 2 add moveto 72 box }
for
```



if-else

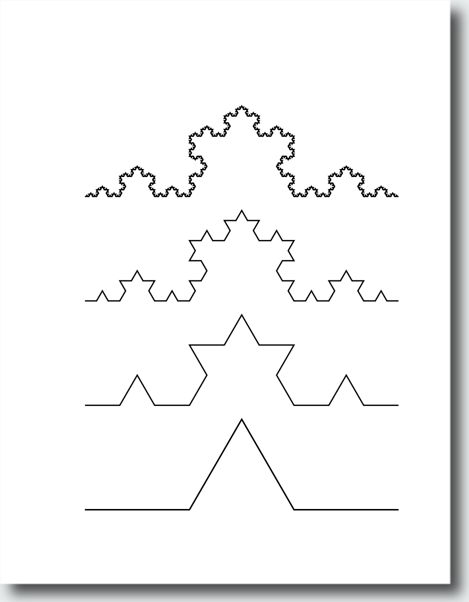
- boolean on stack
- alternatives in braces
- **if** operator

... (hundreds of operators)

Stack-based programming languages: PostScript

An application: all figures in Algorithms in Java

```
%!
72 72 translate

/kochR
  {
    2 copy ge { dup 0 rlineto }
      {
        3 div
        2 copy kochR 60 rotate
        2 copy kochR -120 rotate
        2 copy kochR 60 rotate
        2 copy kochR
      } ifelse
    pop pop
  } def

0   0 moveto   81 243 kochR
0  81 moveto   27 243 kochR
0 162 moveto    9 243 kochR
0 243 moveto    1 243 kochR
stroke
```



See page 218

Queue applications

Familiar applications.

- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

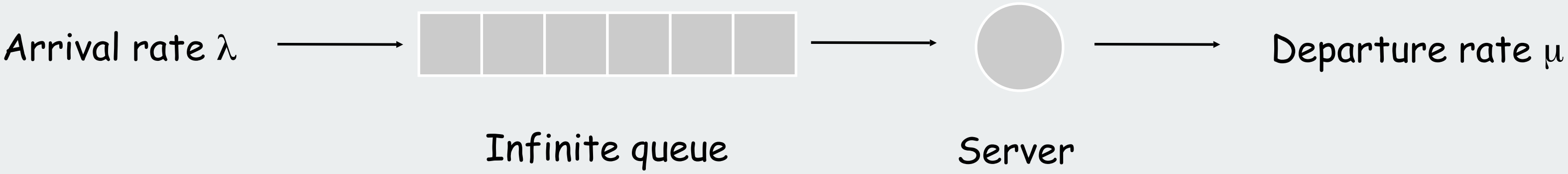
M/D/1 queuing model

M/D/1 queue.

- Customers are serviced at fixed rate of μ per minute.
- Customers arrive according to Poisson process at rate of λ per minute.

inter-arrival time has exponential distribution

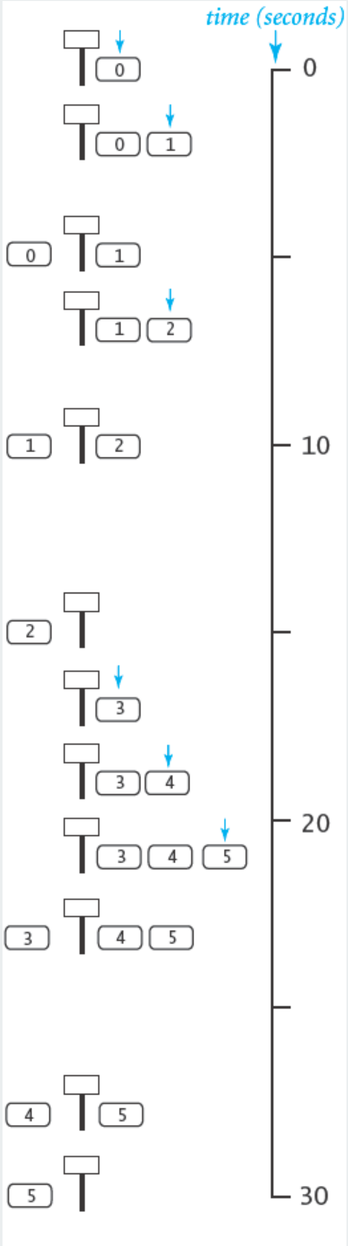
$$\Pr[X \le x] \;=\; 1 - e^{-\lambda x}$$



Q. What is average wait time W of a customer?

Q. What is average number of customers L in system?

M/D/1 queuing model: example

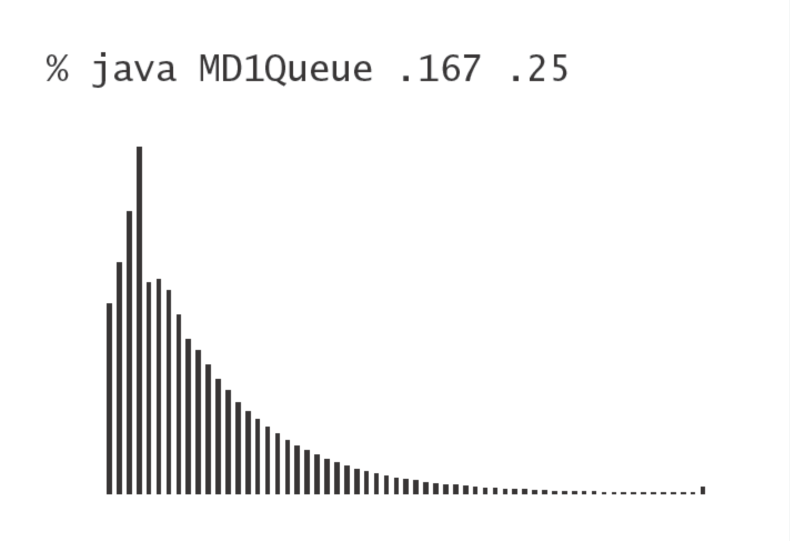


	arrival	departure	wait
0	0	5	5
1	2	10	8
2	7	15	8
3	17	23	6
4	19	28	9
5	21	30	9

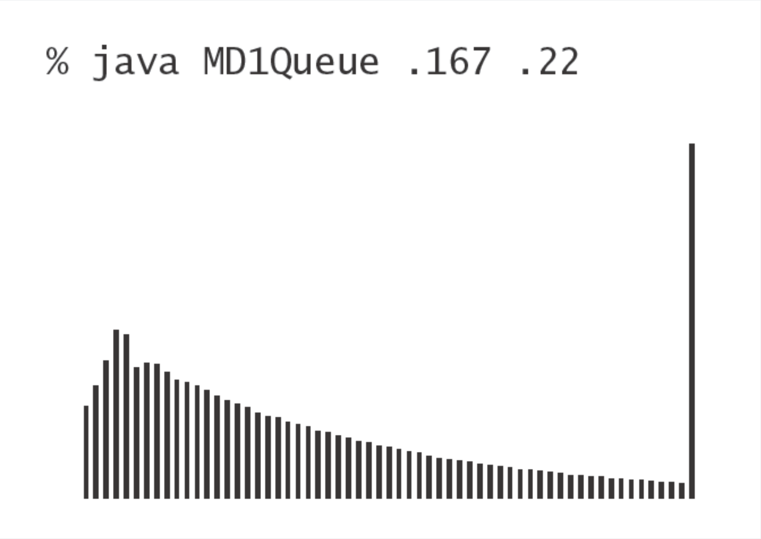
M/D/1 queuing model: experiments and analysis

Observation.

As service rate μ approaches arrival rate λ, service goes to h***.



```
% java MD1Queue .167 .25
```



```
% java MD1Queue .167 .22
```

Queueing theory (see ORFE 309).

$$W = \frac{\lambda}{2\,\mu\underset{\uparrow}{(\mu-\lambda)}} + \frac{1}{\mu}\,, \qquad L = \overset{\text{Little's Law}}{\overset{\downarrow}{\lambda}}\;W$$

wait time W and queue length L approach infinity as service rate approaches arrival rate

M/D/1 queuing model: event-based simulation

```
public class MD1Queue
{
   public static void main(String[] args)
   {
      double lambda = Double.parseDouble(args[0]);   // arrival rate
      double mu     = Double.parseDouble(args[1]);   // service rate
      Histogram hist = new Histogram(60);
      Queue<Double> q = new Queue<Double>();
      double nextArrival = StdRandom.exp(lambda);
      double nextService = 1/mu;
      while (true)
      {
         while (nextArrival < nextService)
         {
            q.enqueue(nextArrival);
            nextArrival += StdRandom.exp(lambda);
         }
         double wait = nextService - q.dequeue();
         hist.addDataPoint(Math.min(60,  (int) (wait)));
         if (!q.isEmpty())
            nextService = nextArrival + 1/mu;
         else
            nextService = nextService + 1/mu;
      }
   }
}
```

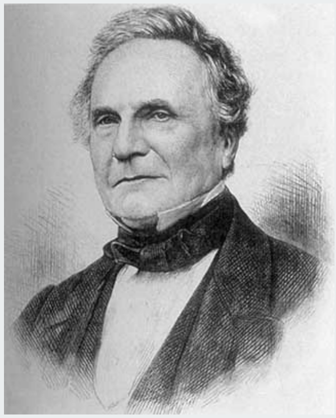
Analysis of Algorithms

- overview
- experiments
- models
- case study
- hypotheses

Updated from:
Algorithms in Java, Chapter 2
Intro to Programming in Java, Section 4.1

Running time

> As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - Charles Babbage

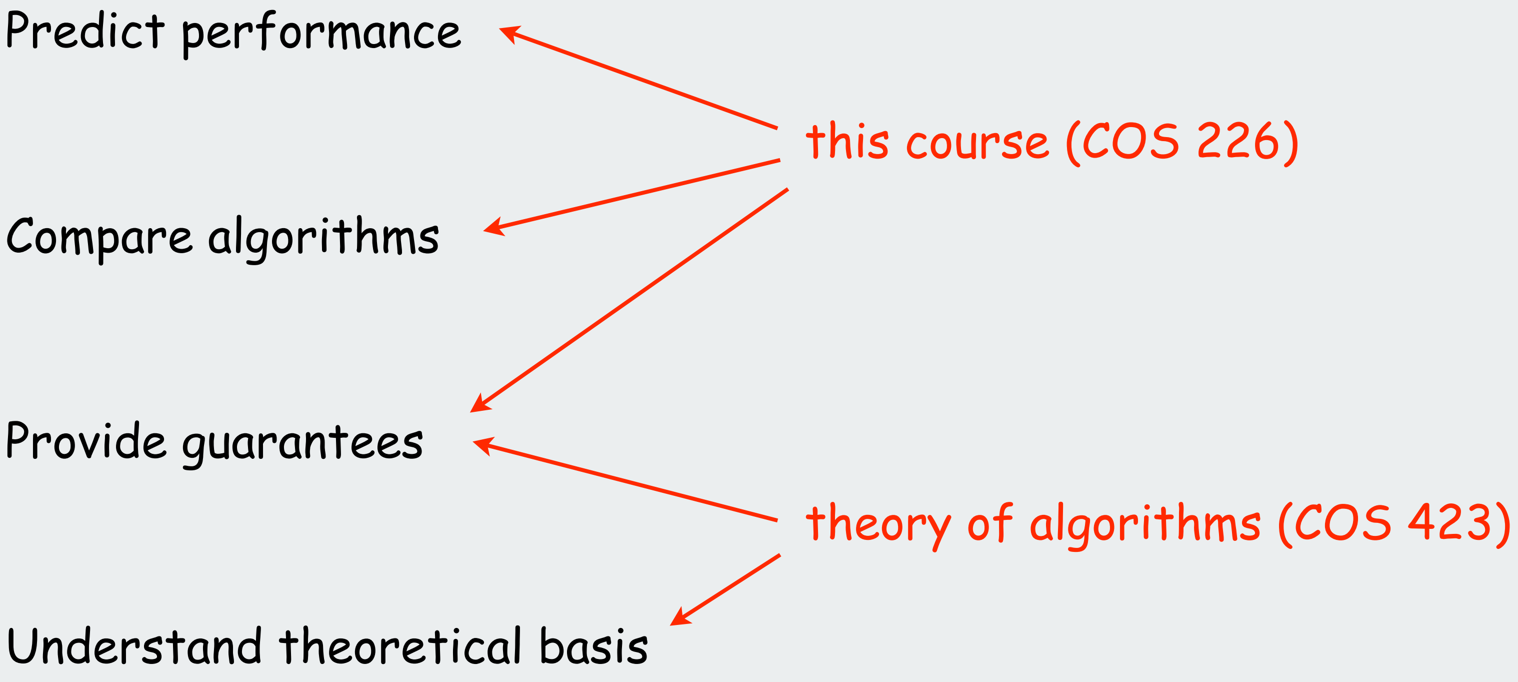


Charles Babbage (1864)

Analytic Engine

how many times do you have to turn the crank?

Reasons to analyze algorithms



Predict performance

Compare algorithms

Provide guarantees

Understand theoretical basis

this course (COS 226)

theory of algorithms (COS 423)

Primary practical reason: avoid performance bugs

Client gets poor performance because programmer did not understand performance characteristics

Overview

Scientific analysis of algorithms:
framework for predicting performance and comparing algorithms.

Scientific method.

- Observe some feature of the universe.
- Hypothesize a model that is consistent with observation.
- Predict events using the hypothesis.
- Verify the predictions by making further observations.
- Validate by repeating until the hypothesis and observations agree.

Principles.

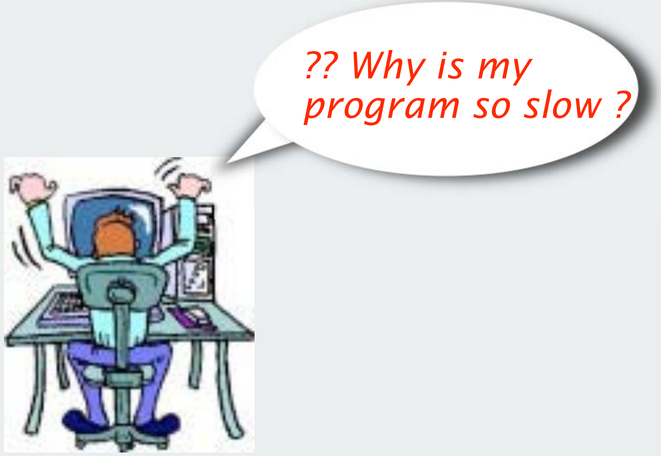
- Experiments must be reproducible.
- Hypotheses must be falsifiable.

Universe = computer itself.

- overview
- **experiments**
- models
- case study
- hypotheses

Experimental algorithmics

Every time you run a program you are doing an experiment!



First step:

Debug your program!

Second step:

Decide on model for experiments on large inputs.

Third step:

Run the program for problems of increasing size.

Experimental evidence: measuring time

- Manual:
- Automatic: `Stopwatch.java`

client code

```
Stopwatch sw = new Stopwatch();
// Run algorithm
double time = sw.elapsedTime();
StdOut.println("Running time: " + time + " seconds");
```

implementation

```
public class Stopwatch
{
   private final long start;

   public Stopwatch()
   {  start = System.currentTimeMillis();  }

   public double elapsedTime()
   {
      long now = System.currentTimeMillis();
      return (now - start) / 1000.0;
   }
}
```

Experimental algorithmics

Many obvious factors affect running time.

- machine
- compiler
- algorithm
- input data

More factors (not so obvious):

- caching
- garbage collection
- just-in-time compilation
- CPU use by other applications

Bad news: it is often difficult to get precise measurements

Good news: we can run a huge number of experiments [stay tuned]

Approach 1: Settle for affordable approximate results

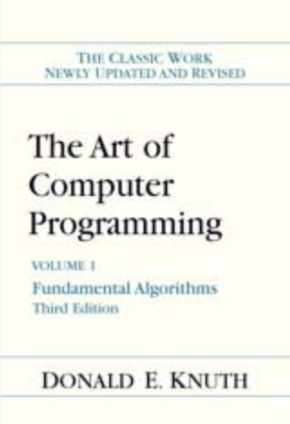
Approach 2: Count abstract operations (machine independent)

- overview
- experiments
- **models**
- case study
- hypotheses

Models for the analysis of algorithms

Total running time: sum of cost × frequency for all operations.

- Need to analyze program to determine set of operations
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.

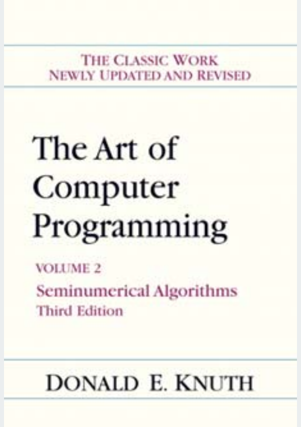


THE CLASSIC WORK
NEWLY UPDATED AND REVISED

The Art of
Computer
Programming

VOLUME 1
Fundamental Algorithms
Third Edition

DONALD E. KNUTH

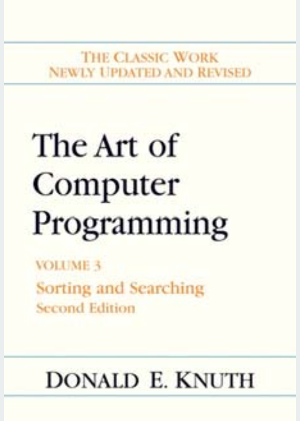


THE CLASSIC WORK
NEWLY UPDATED AND REVISED

The Art of
Computer
Programming

VOLUME 2
Seminumerical Algorithms
Third Edition

DONALD E. KNUTH



THE CLASSIC WORK
NEWLY UPDATED AND REVISED

The Art of
Computer
Programming

VOLUME 3
Sorting and Searching
Second Edition

DONALD E. KNUTH

Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available

Developing models for algorithm performance

In principle, accurate mathematical models are available [Knuth]

In practice,

- formulas can be complicated
- advanced mathematics might be required

costs (depend on machine, compiler)

Ex.

$$T_N = 24\, A_N + 11 B_N + 4 C_N + 3 D_N + 7N + 9 S_N$$

where

$$A_N = 2(N+1) / 3$$

$$B_N = (N + 1)\,(2H_{N+1} - 2H_3 - 1)/6 + 1/2$$

$$C_N = (N + 1)\,(2H_{N+1} - 2H_3 + 1)$$

$$D_N = (N + 1)(1 - 2H_3/3)$$

$$S_N = (N + 1)/5 - 1$$

frequencies
(depend on algorithm, input)

Exact models best left for experts

Bottom line: We use approximate models in this course: $T_N \sim c\, N \log N$

all constants rolled into one

Commonly used notations to model running time

notation	provides	example	shorthand for	used to
Big Theta	growth rate	$\Theta(N^2)$	N^2 $9000\ N^2$ $5\ N^2 + 22\ N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	N^2 $100\ N$ $22\ N \log N + 3N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$9000\ N^2$ N^5 $N^3 + 22\ N \log N + 3N$	develop lower bounds
Tilde	leading term	$\sim 10\ N^2$	$10\ N^2$ $10\ N^2 + 22\ N \log N$ $10\ N^2 + 2\ N + 37$	provide approximate model

↑ used in this course

Predictions and guarantees

Theory of algorithms: The running time of an algorithm is O(f(N))

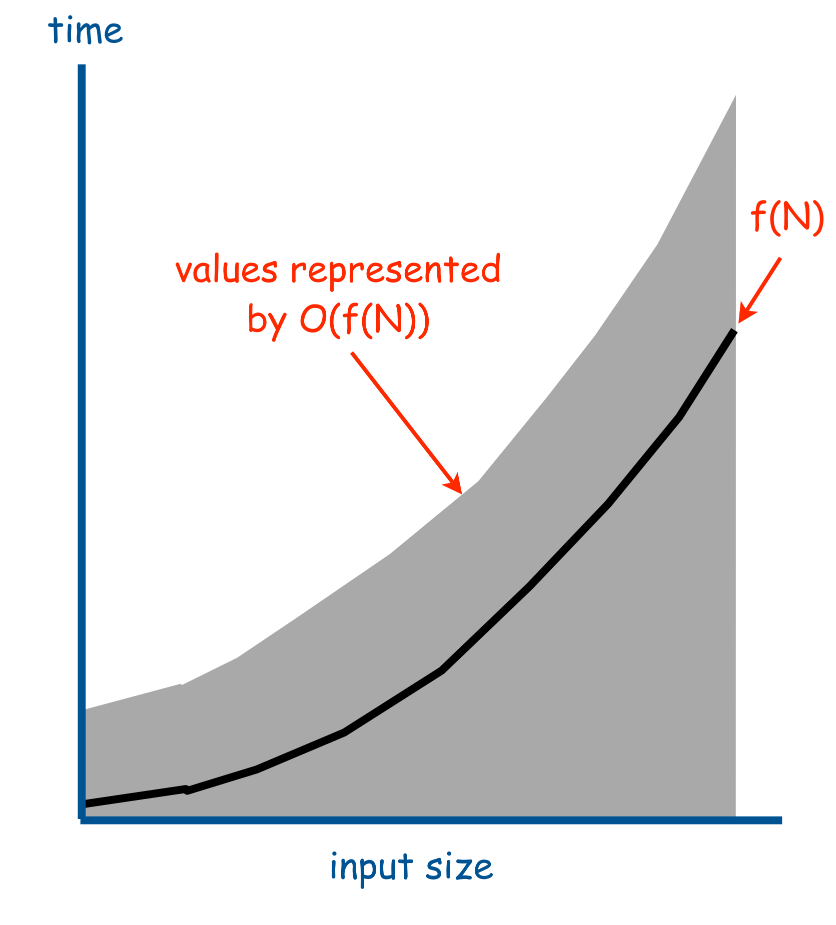
↑
worst case implied

advantages

- describes guaranteed performance
- O-notation absorbs input model

challenges

- cannot use to predict performance
- cannot use to compare algorithms



Predictions and guarantees (continued)

This course: The running time of an algorithm is ~ c f(N)

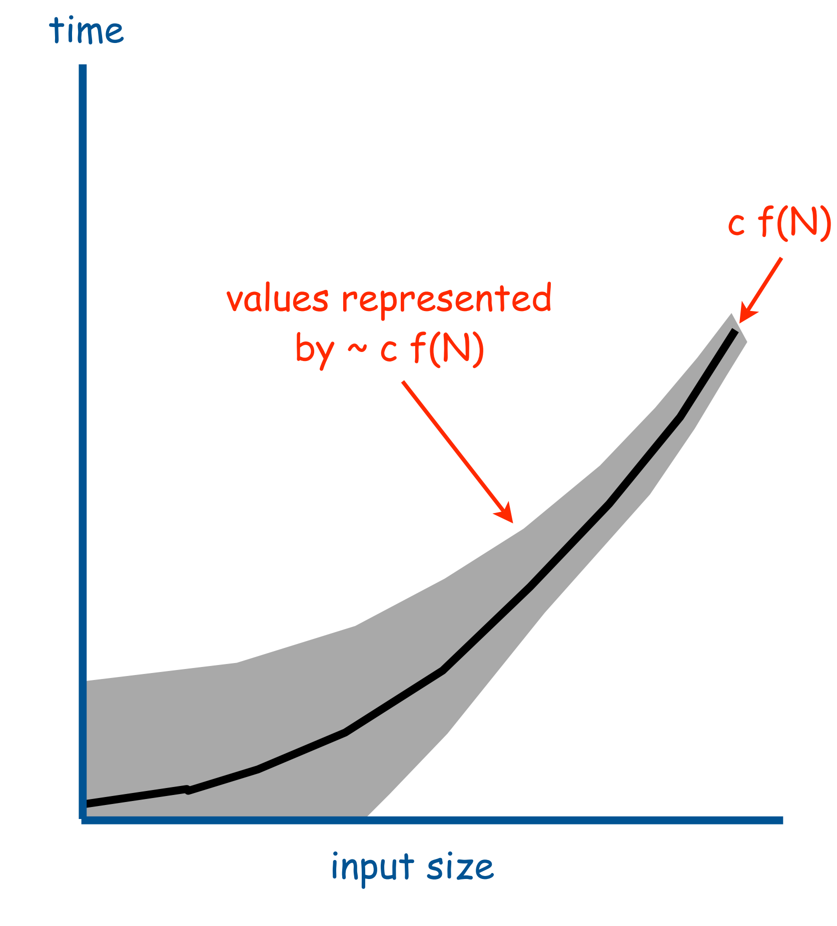
understanding of alg's dependence on input implied

advantages

- can use to predict performance
- can use to compare algorithms

challenges

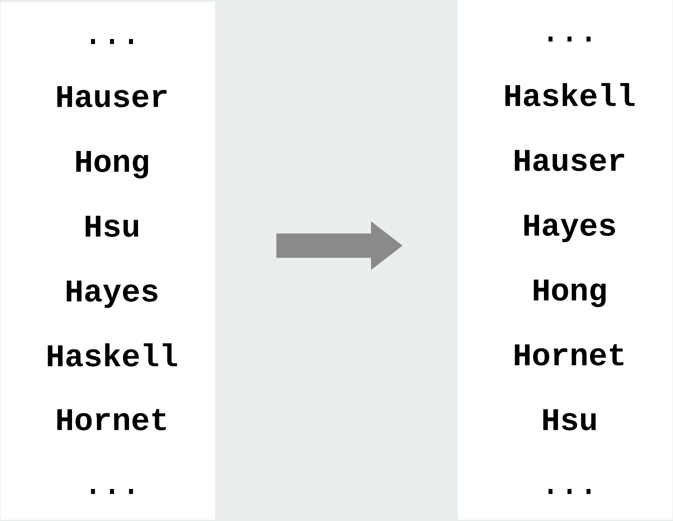
- need to develop accurate input model
- may not provide guarantees



- overview
- experiments
- models
- **case study**
- hypotheses

Case study [stay tuned for numerous algorithms and applications]

Sorting problem: rearrange N given items into ascending order



```
...
Hauser
Hong
Hsu
Hayes
Haskell
Hornet
...
```

```
...
Haskell
Hauser
Hayes
Hong
Hornet
Hsu
...
```

Basic operations: compares and exchanges

compare

```
public static void less(double x, double y)
{  return x < y; }
```

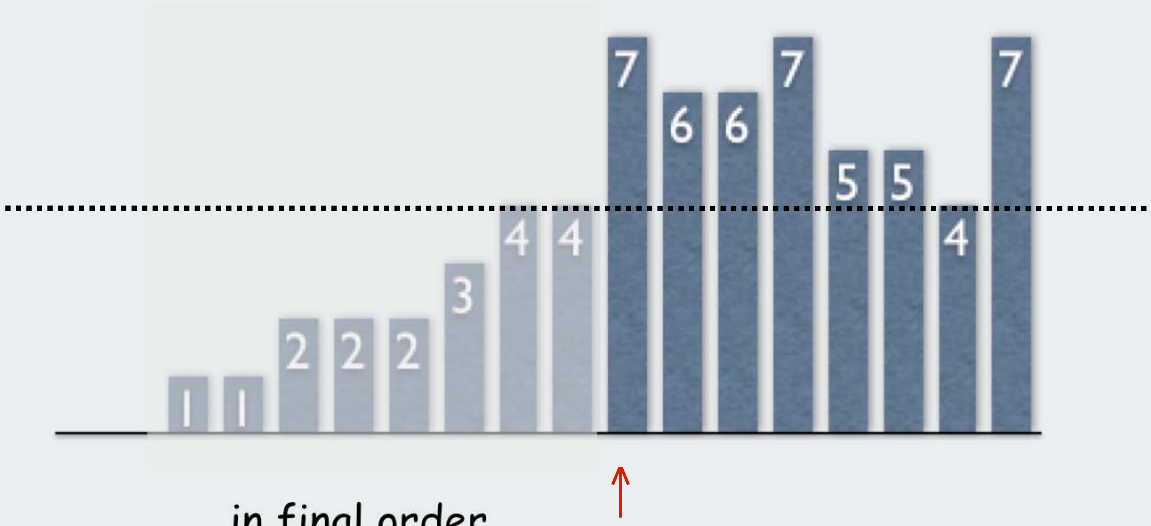
exchange

```
public static void exch(double[] a, int i, int j)
{
   double t = a[i];
   a[i] = a[j];
   a[j] = t;
}
```

Selection sort: an elementary sorting algorithm

Algorithm invariants

- ↑ scans from left to right.
- Elements to the left of ↑ are fixed and in ascending order.
- No element to left of ↑ is larger than any element to its right.



in final order

Selection sort inner loop

- move the pointer to the right

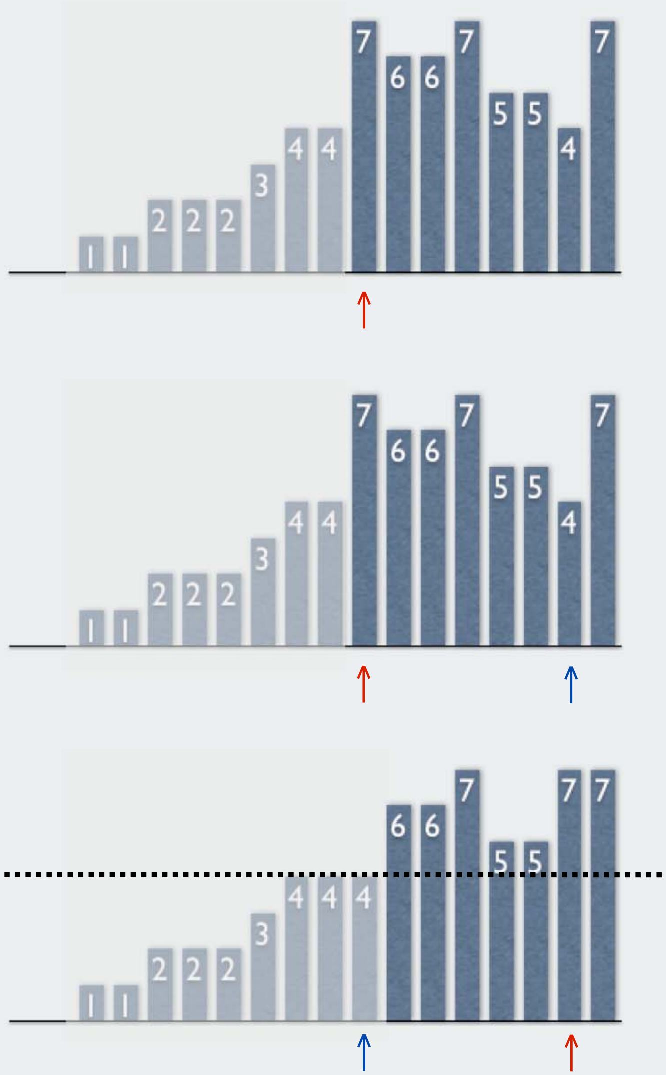
```
i++;
```

- identify index of minimum item on right.

```
int min = i;
for (int j = i+1; j < N; j++)
   if (less(a[j], a[min]))
      min = j;
```

- Exchange into position.

```
exch(a, i, min);
```



Maintains algorithm invariants

Selection sort: Java implementation

```java
public static void sort(double[] a)
{
   for (int i = 0; i < a.length; i++)
   {
      int min = i;
      for (int j = i+1; j < a.length; j++)
         if (less(a[j], a[min]))
            min = j;
      exch(a, i, min);
   }
}
```

most frequent operation
("inner loop")

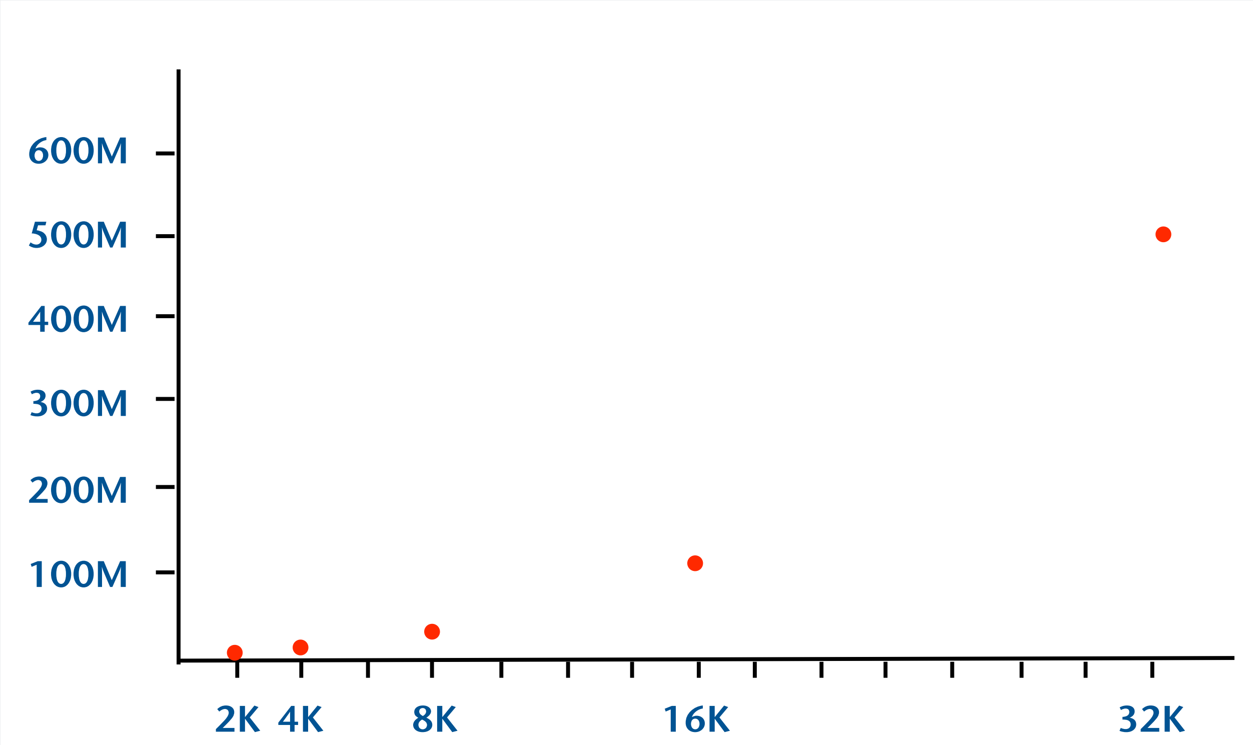
Selection sort: initial observations

Observe, tabulate and plot operation counts for various values of N.

- study most frequently performed operation (compares)
- input model: N random numbers between 0 and 1

add counter to `less()`

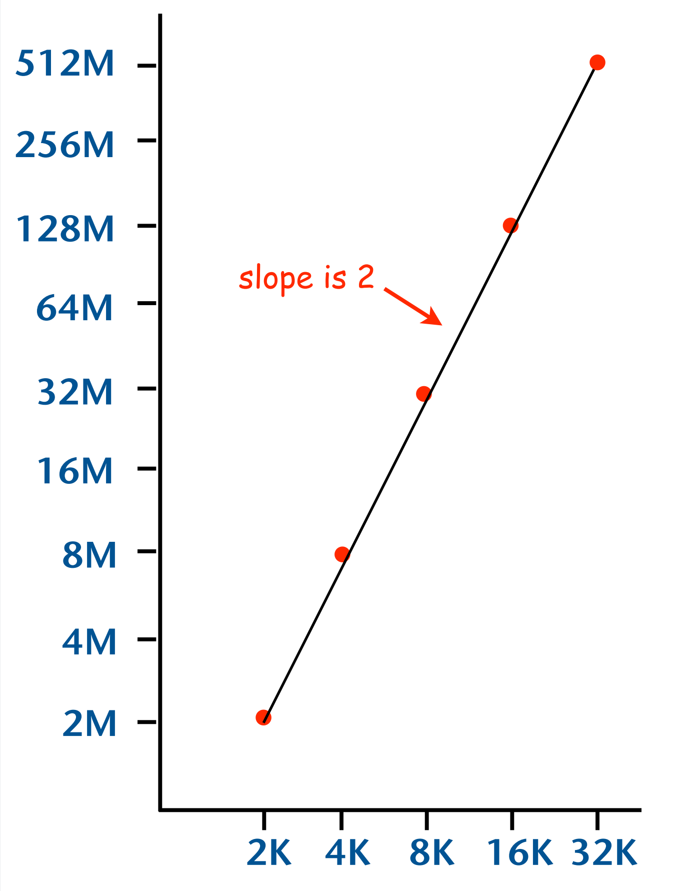
N	compares
2,000	2.1 million
4,000	7.9 million
8,000	32.1 million
16,000	125.9 million
32,000	514.7 million



Selection sort: experimental hypothesis

Data analysis. Plot # compares vs. input size on log-log scale.

N	compares
2,000	2.1 million
4,000	7.9 million
8,000	32.1 million
16,000	125.9 million
32,000	514.7 million



normal scale

$C = a\,N^b$

log-log scale

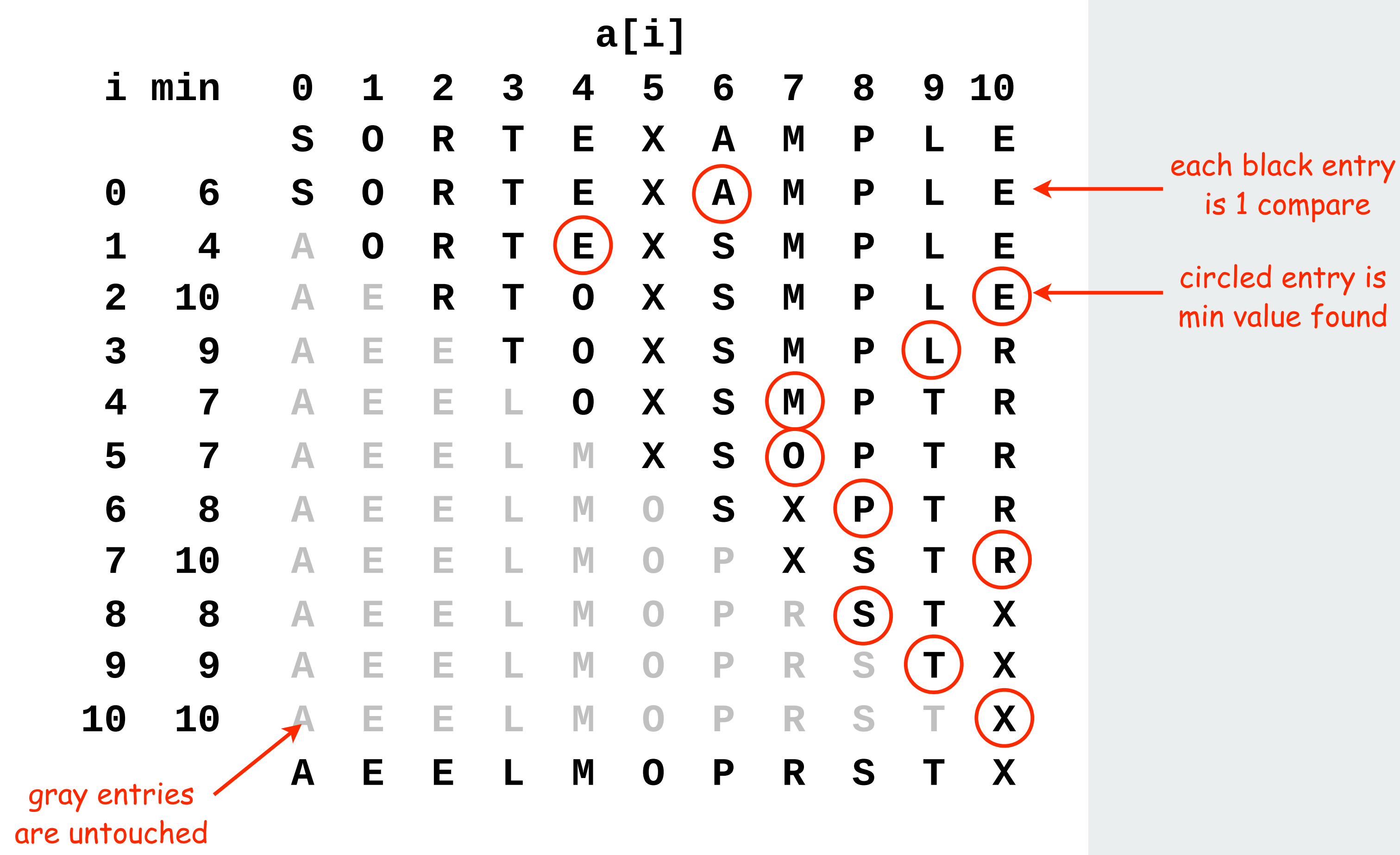
$\lg C = \lg a + b \lg N$

Regression. Fit straight line through data points $\approx\ a\,N^b$. power law

slope

Hypothesis. # compares is $\sim N^2/2$

Selection sort: theoretical model



		a[i]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

each black entry is 1 compare

circled entry is min value found

gray entries are untouched

Hypothesis: number of compares is $N + (N-1) + \ldots + 2 + 1 \sim N^2/2$

$= N(N + 1) / 2$
$= N^2/2 + N/2$
$\sim N^2/2$

Selection sort: Prediction and verification

Hypothesis (experimental and theoretical). # compares is $\sim N^2/2$.

Prediction. 800 million compares for N = 40,000.

Observations.

N	compares
40,000	801.3 million
40,000	799.7 million
40,000	801.6 million
40,000	800.8 million

Verifies.

Prediction. 20 billion compares for N = 200,000.

Observation.

N	compares
200,000	19.997 billion

Verifies.

Selection sort: validation

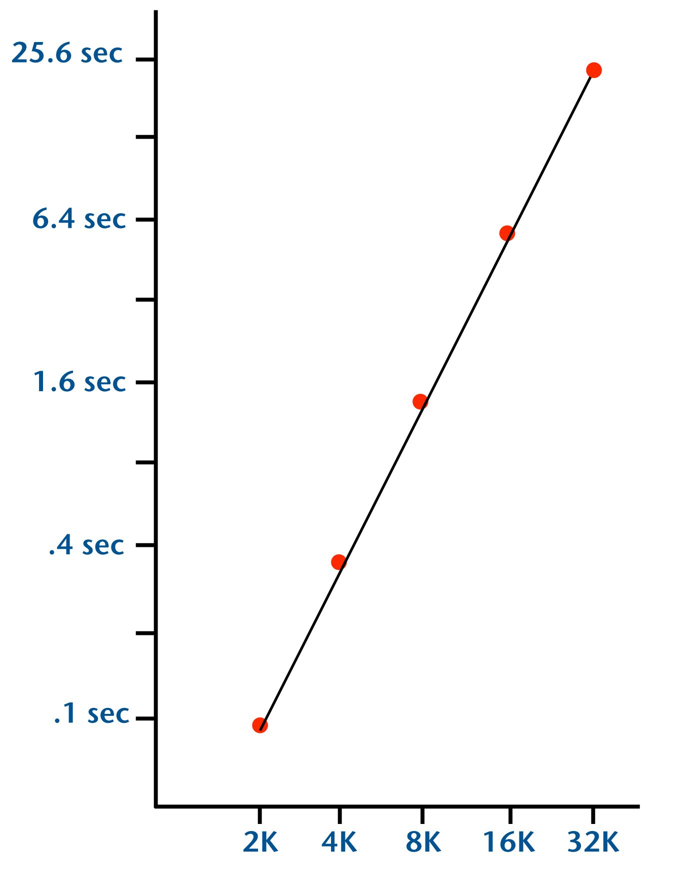
Implicit assumptions

- constant cost per compare
- cost of compares dominates running time

Hypothesis: Running time is ~ c N^2

Validation: Observe actual running time.

N	observed time	$.23 \times 10^{-7} N^2$
2,000	0.1 seconds	0.1
4,000	0.4 seconds	0.4
8,000	1.5 seconds	1.5
16,000	5.6 seconds	5.9
32,000	23.2 seconds	23.5



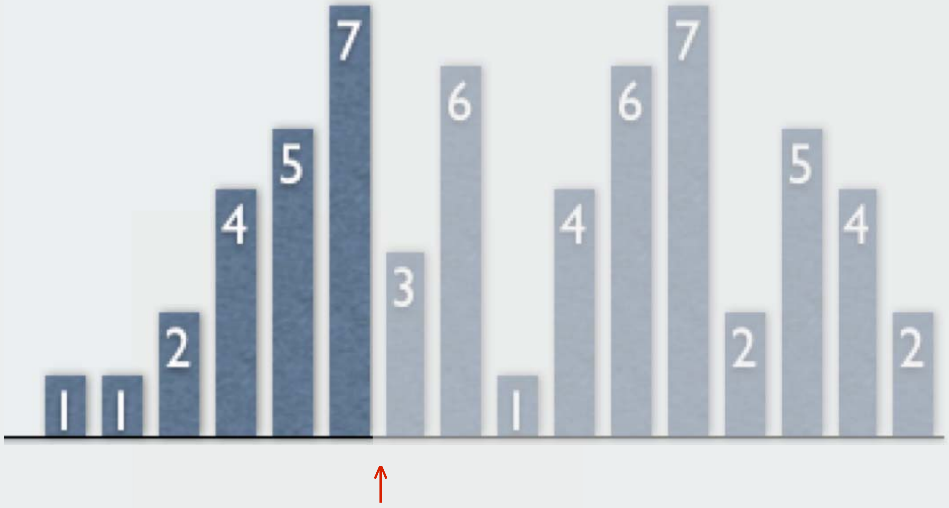
Regression fit validates hypothesis.

A scientific connection between program and natural world.

Insertion sort: another elementary sorting algorithm

Algorithm invariants

- ↑ scans from left to right.
- Elements to the left of ↑ are in ascending order.



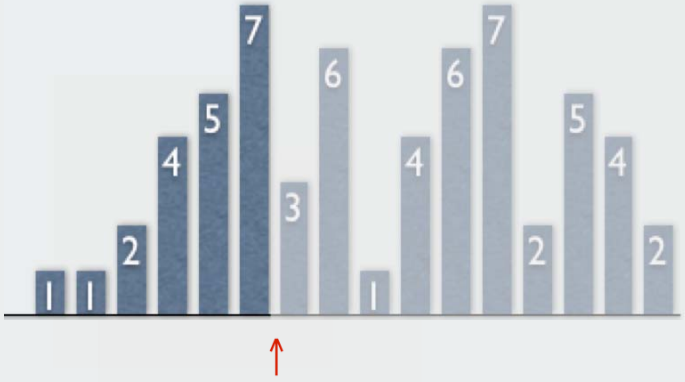
in order

not yet seen

Insertion sort inner loop

- move the pointer to the right

```
i++;
```

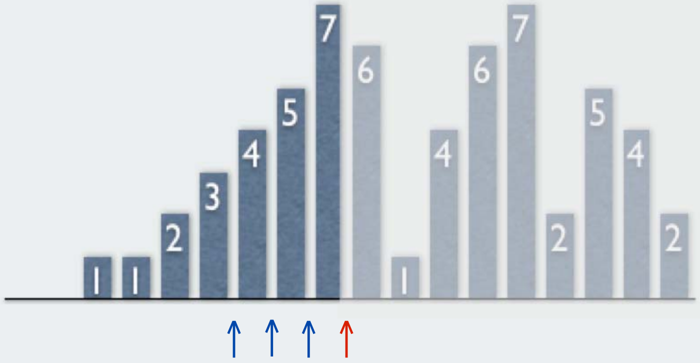


in order

not yet seen

- moving from right to left, exchange a[i] with each larger element to its left

```
for (int j = i; j > 0; j--)
   if (less(a[j], a[j-1]))
        exch(a, j, j-1);
   else break;
```



in order

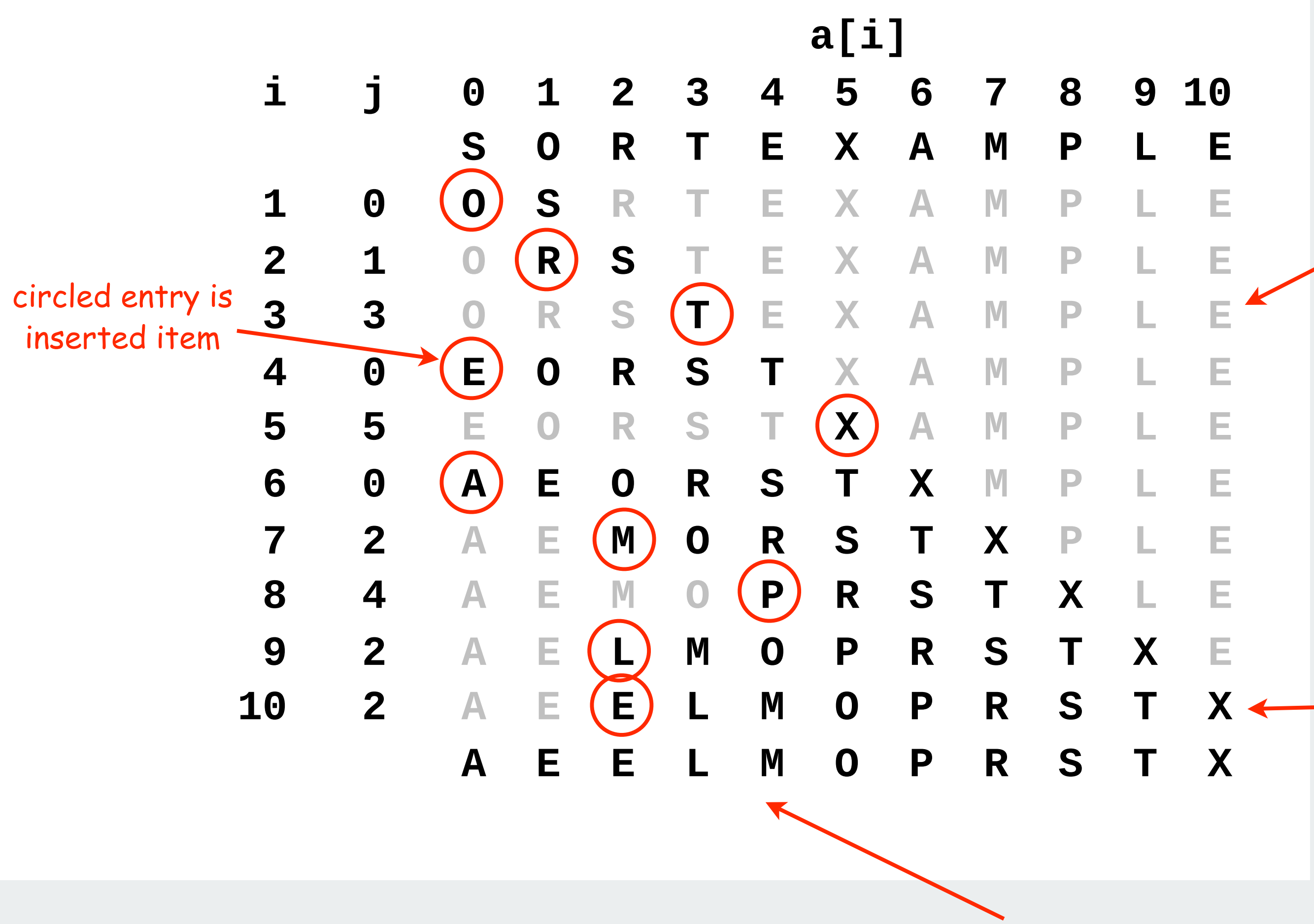
not yet seen

Maintains algorithm invariants

Insertion sort: Java implementation

```
public static void sort(Comparable[] a)
{
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i; j > 0; j--)
         if (less(a[j], a[j-1]))
              exch(a, j, j-1);
         else break;
}
```

Insertion sort: theoretical model



		a[i]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

circled entry is inserted item

gray entries are untouched

each black entry is 1 compare/exch

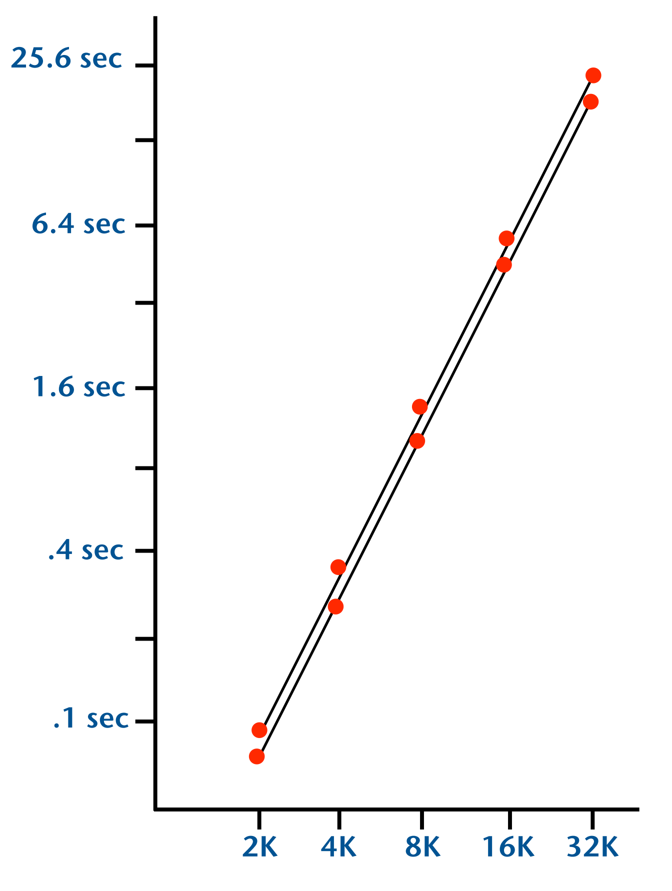
insertions are halfway back, on the average

Hypothesis: number of compares is $(1 + 2 + \ldots + (N-1) + N)/2 \sim N^2/4$ on the average, for randomly ordered input

Experimental comparison of insertion sort and selection sort

Plot both running times on log log scale

- slopes are the same (both 2)
- both are quadratic



Compute ratio of running times

```
% java SortCompare Insertion Selection 4000
For 4000 random double values
Insertion is 1.7 times faster than selection
```

Need detailed analysis
to prefer one over the other

Neither is useful for huge randomly-ordered files

Would Be Nice (if analysis of algorithms were always this easy), But

Mathematics might be difficult

Ex. It is known that properties of singularities of functions in the complex plane play a role in analysis of many algorithms

Leading term might not be good enough

Ex. Selection sort could be linear-time if cost of exchanges is huge

↑ assumption that compares dominate may be invalid

Actual data might not match model

Ex. Insertion sort could be linear-time if keys are roughly in order

↑ assumption that input is randomly ordered may be invalid

Timing may be flawed

- different results on different computers
- different results on same computer at different times

- overview
- experiments
- models
- case study
- **hypotheses**

Practical approach to developing hypotheses

First step: determine asymptotic growth rate for chosen model

- approach 1: run experiments, regression
- approach 2: do the math
- best: do both

Good news: the relatively small set of functions

$1,\ \log N,\ N,\ N \log N,\ N^2,\ N^3$, and 2^N

suffices to describe asymptotic growth rate of typical algorithms

After determining growth rate

- use doubling hypothesis (to predict performance)
- use ratio hypothesis (to compare algorithms)

Common asymptotic-growth hypotheses (summary)

growth rate	name	typical code framework	description	example
1	constant	`a = b + c;`	statement	add two numbers

log N	logarithmic	`while (N > 1)`
		`{   N = N / 2;   ...   }`
			divide in half	binary search

N	linear	`for (int i = 0; i < N; i++)`
		`{  ...         }`
			loop	find the maximum

N log N	linearithmic	[see next lecture]	divide and conquer	sort an array

N^2	quadratic	`for (int i = 0; i < N; i++)`
		`   for (int j = 0; j < N; j++)`
		`   {  ...         }`
			double loop	check all pairs

N^3	cubic	`for (int i = 0; i < N; i++)`
		`   for (int j = 0; j < N; j++)`
		`      for (int k = 0; k < N; k++)`
		`      {  ...         }`
			triple loop	check all triples

2^N	exponential	[see lecture 24]	exhaustive search	check all possibilities

Aside: practical implications of asymptotic growth

For back-of-envelope calculations, assume

decade	processor speed	instructions per second
1970s	1M Hz	10^6
1980s	10M Hz	10^7
1990s	100M Hz	10^8
2000s	1G Hz	10^9

seconds	equivalent
1	1 second
10	10 seconds
10^2	1.7 minutes
10^3	17 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
. . .	forever
10^{17}	age of universe

How long to process millions of inputs?

Ex. Population of NYC was “millions” in 1970s; still is

How many inputs can be processed in minutes?

Ex. Customers lost patience waiting “minutes” in 1970s; still do

Aside: practical implications of asymptotic growth

growth rate	problem size solvable in minutes				time to process millions of inputs			
	1970s	1980s	1990s	2000s	1970s	1980s	1990s	2000s
1	any	any	any	any	instant	instant	instant	instant
log N	any	any	any	any	instant	instant	instant	instant
N	millions	tens of millions	hundreds of millions	billions	minutes	seconds	second	instant
N log N	hundreds of thousands	millions	millions	hundreds of millions	hour	minutes	tens of seconds	seconds
N^2	hundreds	thousand	thousands	tens of thousands	decades	years	months	weeks
N^3	hundred	hundreds	thousand	thousands	never	never	never	millenia

Practical implications of asymptotic-growth: another view

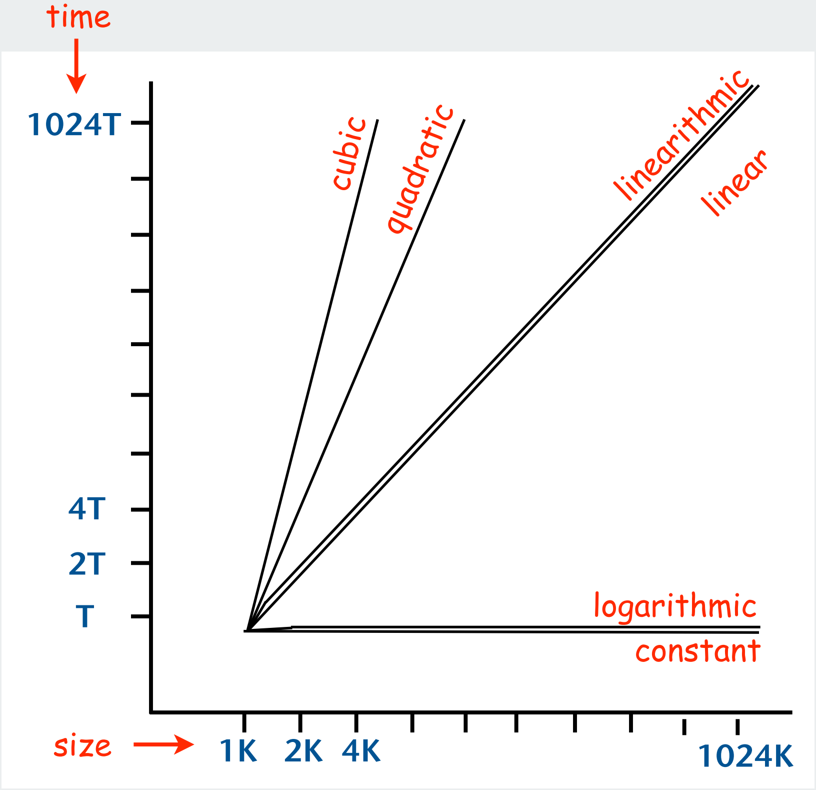
growth rate	name	description	effect on a program that runs for a few seconds	
			time for 100x more data	size for 100x faster computer
1	constant	independent of input size	a few seconds	same
log N	logarithmic	nearly independent of input size	a few seconds	same
N	linear	optimal for N inputs	a few minutes	100x
N log N	linearithmic	nearly optimal for N inputs	a few minutes	100x
N^2	quadratic	not practical for large problems	several hours	10x
N^3	cubic	not practical for large problems	several weeks	4-5x
2^N	exponential	useful only for tiny problems	forever	1x

Developing asymptotic order of growth hypotheses with doubling

To formulate hypothesis for asymptotic growth rate:

- compute T(2N)/T(N) as accurately (and for N as large) as is affordable
- use this table

ratio	hypothesis	reason
1	constant or logarithmic	$c / c = 1$ $c \log 2N / c \log N \sim 1$
2	linear or linearithmic	$c\ 2N / c\ N = 2$ $c\ 2\ N \log (2N) / c\ N \log N \sim 2$
4	quadratic	$c\ (2N)^2 / c\ N^2 = 4$
9	cubic	$c\ (2N)^2 / c\ N^2 = 9$



$= 2 \log(2N)/\log N$
$= 2 (\log 2 + \log N)/\log N$
$= 2 + 2 \log 2/\log N$
~ 2

Example revisited: methods for timing sort algorithms

Compute time to sort **a[]** with **alg**

```
public static double time(String alg, Double[] a)
{
   Stopwatch sw = new Stopwatch();
   if (alg.equals("Insertion")) Insertion.sort(a);
   if (alg.equals("Selection")) Selection.sort(a);
   if (alg.equals("Shell")) Shell.sort(a);
   if (alg.equals("Merge")) Merge.sort(a);
   if (alg.equals("Quick")) Quick.sort(a);
   return sw.elapsedTime();
}
```

Compute total time to to sort **trials** arrays of N random doubles with **alg**

```
public static double timetrials(String alg, int N, int trials)
{
   double total = 0.0;
   Double[] a = new Double[N];
   for (int t = 0; t < trials; t++)
   {
      for (int i = 0; i < N; i++)
         a[i] = StdRandom.uniform();
      total += time(alg, a);
   }
   return total;
}
```

Developing asymptotic order of growth hypotheses with doubling

CAUTION

THIS CODE
MAY NOT
BE READY
FOR THE
REAL WORLD

```
public class SortGrowth
{
   public static void main(String[] args)
   {
      String alg = args[0];
      int N = 1000;
      if (args.length > 1)
          N = Integer.parseInt(args[1]);
      int trials = 100;
      if (args.length > 2)
          trials = Integer.parseInt(args[2]);
      double ratio = timetrials(alg, 2*N, trials);
                        / timetrials(alg, N, trials);
      StdOut.printf("Ratio is %f\n", ratio);
      if (ratio > 1.8 && ratio < 2.2)
         StdOut.printf("  %s is linear or linearithmic\n", alg);
      if (ratio > 3.8 && ratio < 4.2)
         StdOut.printf("  %s is quadratic\n", alg);

   }
}
```

```
% java SortGrowth Selection
Ratio is 4.1
  Selection is quadratic
```

```
% java SortGrowth Insertion
Ratio is 3.645756

% java SortGrowth Insertion 4000 1000
Ratio is 3.969934
  Insertion is quadratic
```

Predicting performance with doubling hypotheses

A practical approach to predict running time:

- analyze algorithm and run experiments to develop hypothesis that asymptotic growth rate of running time is ~ c T(N)
- run algorithm for some value of N, measure running time
- prediction: increasing input size by a factor of 2
  increases running time by a factor of T(2N)/T(N)

growth rate	name	$\frac{T(2N)}{T(N)}$
1	constant	1
log N	logarithmic	~1
N	linear	2
N log N	linearithmic	~2
N^2	quadratic	4
N^3	cubic	9

Example: selection sort

N	observed time
2,000	0.1 seconds
4,000	0.4 seconds
8,000	1.5 seconds
16,000	5.6 seconds
32,000	23.2 seconds

numbers increase by a factor of 2

numbers increase by a factor of 4

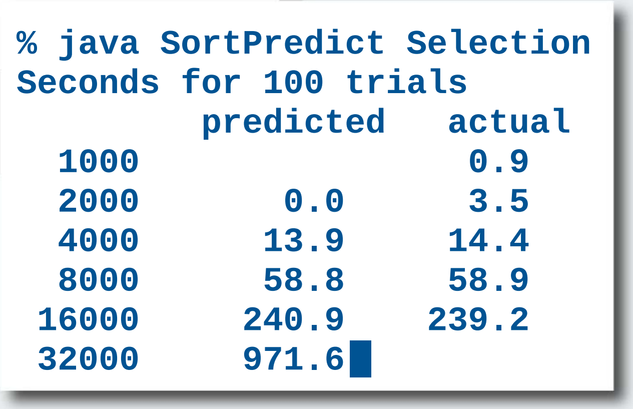
Use algorithm itself to implicitly compute leading-term constant

Predicting performance with doubling hypotheses

CAUTION

THIS CODE
MAY NOT
BE READY
FOR THE
REAL WORLD

```
public class SortPredict
{
   public static void main(String[] args)
   {
      String alg = args[0];
      int trials = 100;
      if (args.length > 1) trials = Integer.parseInt(args[1]);
      StdOut.printf("Seconds for %d trials\n", trials);
      StdOut.printf("       predicted actual\n  1000        ");
      double old = Double.POSITIVE_INFINITY;
      for (int N = 1000; true; N = 2*N)
      {
         total = timeTrials(alg, N, trials);
         double guess = (total/old)*total;
         StdOut.printf(" %7.1f\n %5d %7.1f", total, 2*N, guess);
         old = total;
      }
   }
}
```



```
% java SortPredict Selection
Seconds for 100 trials
         predicted    actual
  1000                  0.9
  2000         0.0      3.5
  4000        13.9     14.4
  8000        58.8     58.9
 16000       240.9    239.2
 32000       971.6█
```

Note: **SortGrowth** is not needed!

[This code works for any power law.]

and deep math says that running time
of typical algs must satisfy power law

Comparing algorithms with ratio hypotheses

A practical way to compare algorithms A and B with the same growth rate

- hypothesize that running times are ~ c_A f(N) and ~ c_B f(N)
- run algorithms for some value of N, measure running times
- Prediction: Algorithm A is a factor of c_A/c_B faster than Algorithm B

To compare algorithms with different growth rates

- hypothesize that the one with the smaller rate is faster
- validate hypothesis for inputs of interest
  [values of constants may be significant]

To determine whether growth rates are the same or different

- compute ratios of running times as input size doubles
- [growth rates are the same if ratios do not change]

Use algorithms themselves to compute complex leading-term constants

Comparing algorithms with ratio hypothesis

CAUTION

THIS CODE
MAY NOT
BE READY
FOR THE
REAL WORLD

```
public class SortCompare
{
   public static void main(String[] args)
   {
      String alg1 = args[0];
      String alg2 = args[1];
      int N  = Integer.parseInt(args[2]);
      int trials = 100;
      if (args.length > 3) trials = Integer.parseInt(args[3]);
      double time1 = 0.0;
      double time2 = 0.0;
      Double[] a1 = new Double[N];
      Double[] a2 = new Double[N];
      for (int t = 0; t < trials; t++)
      {
         for (int i = 0; i < N; i++)
         {  a1[i] = Math.random(); a2[i] = a1[i]; }
         time1 += time(alg1, a1);
         time2 += time(alg2, a2);
      }
      StdOut.printf("For %d random Double values\n    %s is", N, alg1);
      StdOut.printf(" %.1f times faster than %s\n", time2/time1, alg2);

   }
}
```

best to test algs on same input

```
% java SortCompare Insertion Selection 4000
For 4000 random Double values
    Insertion is 1.7 times faster than Selection
```

Summary: turning the crank

Yes, analysis of algorithms might be challenging, BUT

Mathematics might be difficult?

- only a few functions seem to turn up
- doubling, ratio tests cancel complicated constants

Leading term might not be good enough?

- debugging tools are available to identify bottlenecks
- typical programs have short inner loops

Actual data might not match model?

- need to understand input to effectively process it
- approach 1: design for the worst case
- approach 2: randomize, depend on probabilistic guarantee

Timing may be flawed?

- limits on experiments insignificant compared to other sciences
- different computers are different!

Sorting Algorithms

- rules of the game
- shellsort
- mergesort
- quicksort
- animations

Reference:
Algorithms in Java, Chapters 6-8

Classic sorting algorithms

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Shellsort.

- Warmup: easy way to break the N^2 barrier.
- Embedded systems.

Mergesort.

- Java sort for objects.
- Perl, Python stable sort.

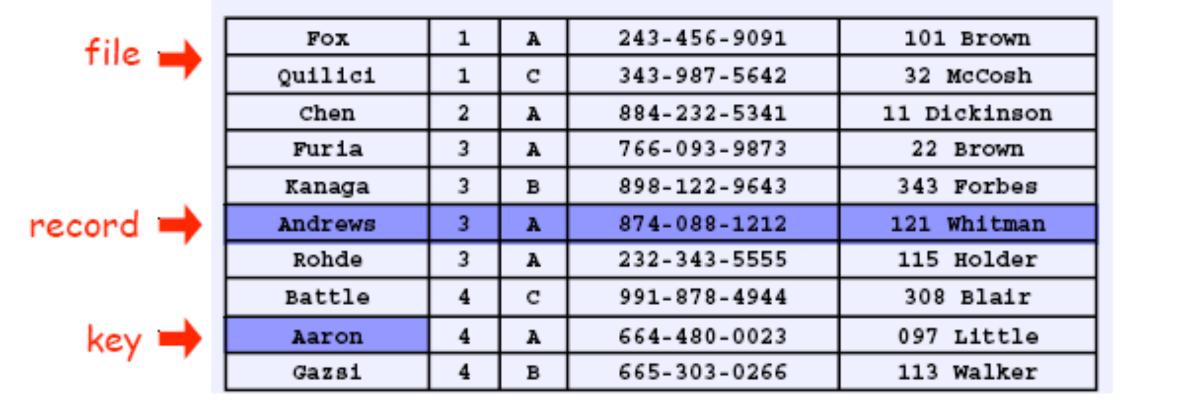
Quicksort.

- Java sort for primitive types.
- C qsort, Unix, g++, Visual C++, Python.

- rules of the game
- shellsort
- mergesort
- quicksort
- animations

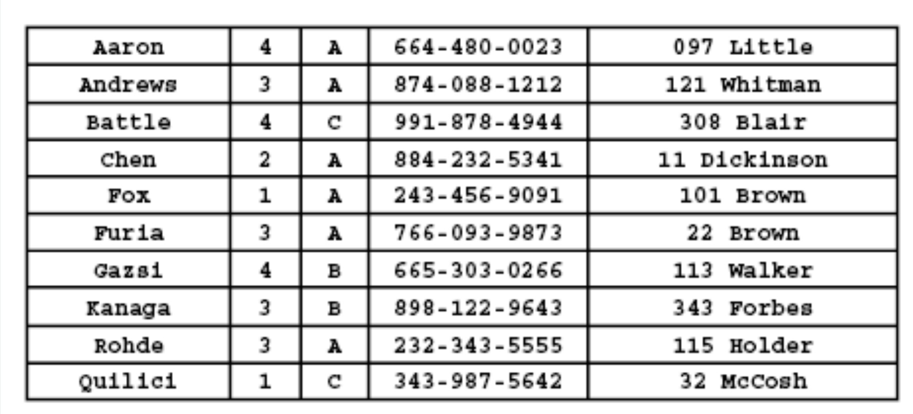
Basic terms

Ex: student record in a University.



Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

Sort: rearrange sequence of objects into ascending order.



Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Sample sort client

Goal: Sort any type of data

Example. List the files in the current directory, sorted by file name.

```
import java.io.File;
public class Files
{
   public static void main(String[] args)
   {
      File directory = new File(args[0]);
      File[] files = directory.listFiles();
      Insertion.sort(files);
      for (int i = 0; i < files.length; i++)
         System.out.println(files[i]);
   }
}
```

```
% java Files .
Insertion.class
Insertion.java
InsertionX.class
InsertionX.java
Selection.class
Selection.java
Shell.class
Shell.java
ShellX.class
ShellX.java
index.html
```

Next: How does sort compare file names?

Callbacks

Goal. Write robust sorting library method that can sort any type of data using the data type's natural order.

Callbacks.

- Client passes array of objects to sorting routine.
- Sorting routine calls back object's comparison function as needed.

Implementing callbacks.

- Java: interfaces.
- C: function pointers.
- C++: functors.

Callbacks

client

```
import java.io.File;
public class SortFiles
{
   public static void main(String[] args)
   {
      File directory = new File(args[0]);
      File[] files = directory.listFiles();
      Insertion.sort(files);
      for (int i = 0; i < files.length; i++)
         System.out.println(files[i]);
   }
}
```

object implementation

```
public class File
implements Comparable<File>
{
   ...
   public int compareTo(File b)
   {
      ...
      return -1;
      ...
      return +1;
      ...
      return 0;
   }
}
```

interface

```
interface Comparable <Item>
{
   public int compareTo(Item);
}
```

built in to Java

sort implementation

```
public static void sort(Comparable[] a)
{
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i; j > 0; j--)
         if (a[j].compareTo(a[j-1]))
            exch(a, j, j-1);
         else break;
}
```

Key point: no reference to File →

Callbacks

Goal. Write robust sorting library that can sort any type of data into sorted order using the data type's natural order.

Callbacks.

- Client passes array of objects to sorting routine.
- Sorting routine calls back object's comparison function as needed.

Implementing callbacks.

- Java: interfaces.
- C: function pointers.
- C++: functors.

Plus: Code reuse for all types of data

Minus: Significant overhead in inner loop

This course:

- enables focus on algorithm implementation
- use same code for experiments, real-world data

Interface specification for sorting

Comparable interface.

Must implement method `compareTo()` so that `v.compareTo(w)`returns:

- a negative integer if v is less than w
- a positive integer if v is greater than w
- zero if v is equal to w

Consistency.

Implementation must ensure a total order.

- if (a < b) and (b < c), then (a < c).
- either (a < b) or (b < a) or (a = b).

Built-in comparable types. `String`, `Double`, `Integer`, `Date`, `File`.

User-defined comparable types. Implement the `Comparable` interface.

Implementing the Comparable interface: example 1

Date data type (simplified version of built-in Java code)

```
public class Date implements Comparable<Date>
{
   private int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }

   public int compareTo(Date b)
   {
      Date a = this;
      if (a.year  < b.year ) return -1;
      if (a.year  > b.year ) return +1;
      if (a.month < b.month) return -1;
      if (a.month > b.month) return +1;
      if (a.day   < b.day  ) return -1;
      if (a.day   > b.day  ) return +1;
      return 0;
   }
}
```

only compare dates
to other dates

Implementing the Comparable interface: example 2

Domain names

- Subdomain: **`bolle.cs.princeton.edu`**.
- Reverse subdomain: **`edu.princeton.cs.bolle`**.
- Sort by reverse subdomain to group by category.

```
public class Domain implements Comparable<Domain>
{
   private String[] fields;
   private int N;
   public Domain(String name)
   {
       fields = name.split("\\.");
       N = fields.length;
   }
   public int compareTo(Domain b)
   {
      Domain a = this;
      for (int i = 0; i < Math.min(a.N, b.N); i++)
      {
         int c = a.fields[i].compareTo(b.fields[i]);
         if      (c < 0) return -1;
         else if (c > 0) return +1;
      }
      return a.N - b.N;
   }
}
```

details included for the bored...

unsorted

```
ee.princeton.edu
cs.princeton.edu
princeton.edu
cnn.com
google.com
apple.com
www.cs.princeton.edu
bolle.cs.princeton.edu
```

sorted

```
com.apple
com.cnn
com.google
edu.princeton
edu.princeton.cs
edu.princeton.cs.bolle
edu.princeton.cs.www
edu.princeton.ee
```

Sample sort clients

File names

```
import java.io.File;
public class Files
{
   public static void main(String[] args)
   {
      File directory = new File(args[0]);
      File[] files = directory.listFiles()
      Insertion.sort(files);
      for (int i = 0; i < files.length; i++)
         System.out.println(files[i]);
   }
}
```

```
% java Files .
Insertion.class
Insertion.java
InsertionX.class
InsertionX.java
Selection.class
Selection.java
Shell.class
Shell.java
```

Random numbers

```
public class Experiment
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Double[] a = new Double[N];
      for (int i = 0; i < N; i++)
         a[i] = Math.random();
      Selection.sort(a);
      for (int i = 0; i < N; i++)
         System.out.println(a[i]);
   }
}
```

```
% java Experiment 10
0.08614716385210452
0.09054270895414829
0.10708746304898642
0.21166190071646818
0.363292849257276
0.460954145685913
0.5340026311350087
0.7216129793703496
0.9003500354411443
0.9293994908845686
```

Several Java library data types implement Comparable

You can implement Comparable for your own types

Two useful abstractions

Helper functions. Refer to data only through two operations.

- less. Is v less than w ?

```
private static boolean less(Comparable v, Comparable w)
{
   return (v.compareTo(w) < 0);
}
```

- exchange. Swap object in array at index i with the one at index j.

```
private static void exch(Comparable[] a, int i, int j)
{
   Comparable t = a[i];
   a[i] = a[j];
   a[j] = t;
}
```

Sample sort implementations

selection sort

```
public class Selection
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a, j, min)) min = j;
         exch(a, i, min);
      }
   }
   ...
}
```

insertion sort

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 1; i < N; i++)
      for (int j = i; j > 0; j--)
         if (less(a[j], a[j-1]))
              exch(a, j, j-1);
         else break;
  }
   ...
}
```

Why use less() and exch() ?

Switch to faster implementation for primitive types

```
private static boolean less(double v, double w)
{
   return v < w;
}
```

Instrument for experimentation and animation

```
private static boolean less(double v, double
w)
{
   cnt++;
   return v < w;
```

Translate to other languages

```
 ...
for (int i = 1; i < a.length; i++)
      if (less(a[i], a[i-1]))
         return false;
   return true;}
```

Good code in C, C++, JavaScript, Ruby....

Properties of elementary sorts (review)

Selection sort

							a[i]					
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Running time: Quadratic (~c N2)

Exception: expensive exchanges (could be linear)

Insertion sort

							a[i]					
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

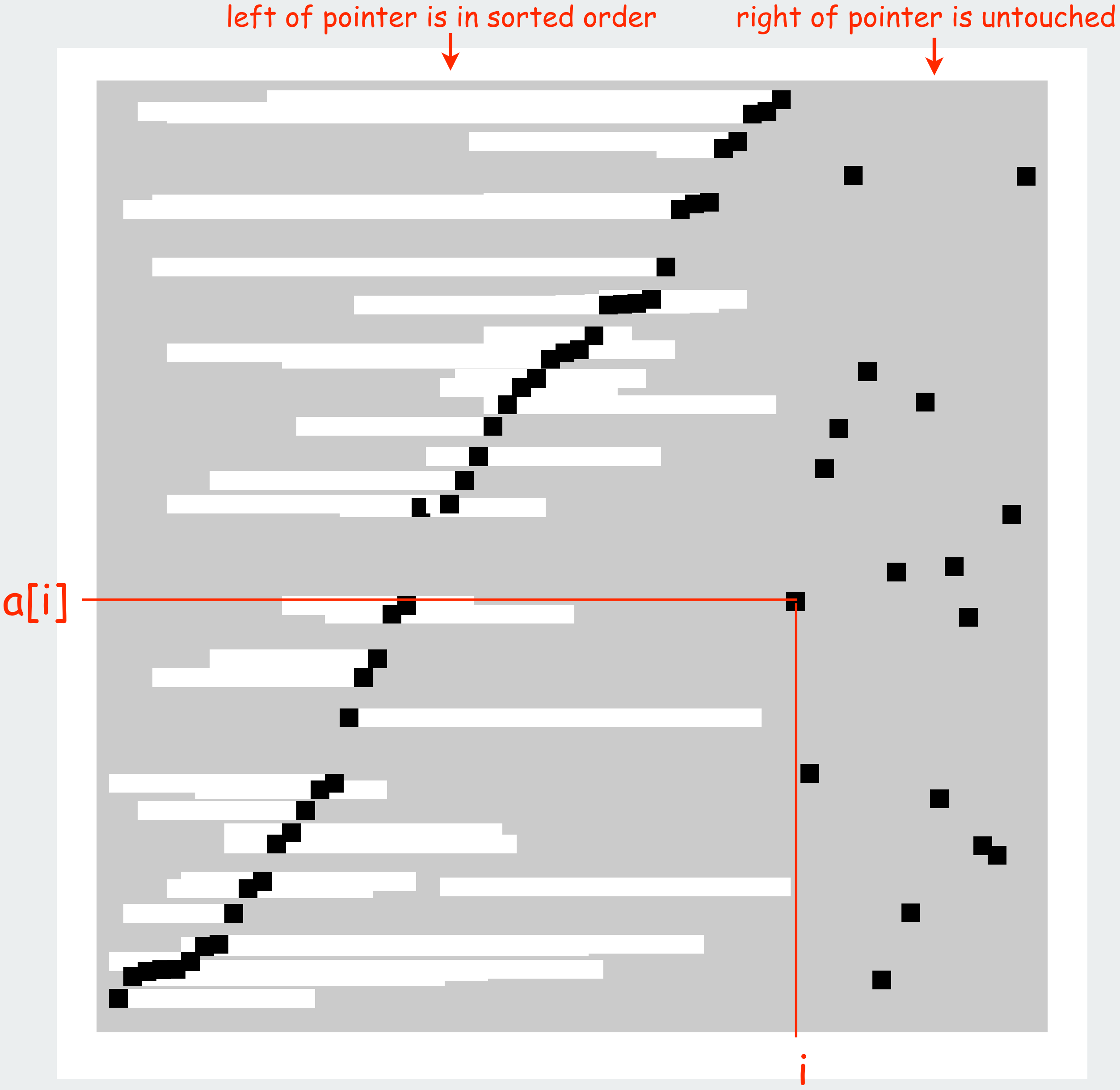
Running time: Quadratic (~c N2)

Exception: input nearly in order (could be linear)

Bottom line: both are quadratic (too slow) for large randomly ordered files

- rules of the game
- **shellsort**
- mergesort
- quicksort
- animations

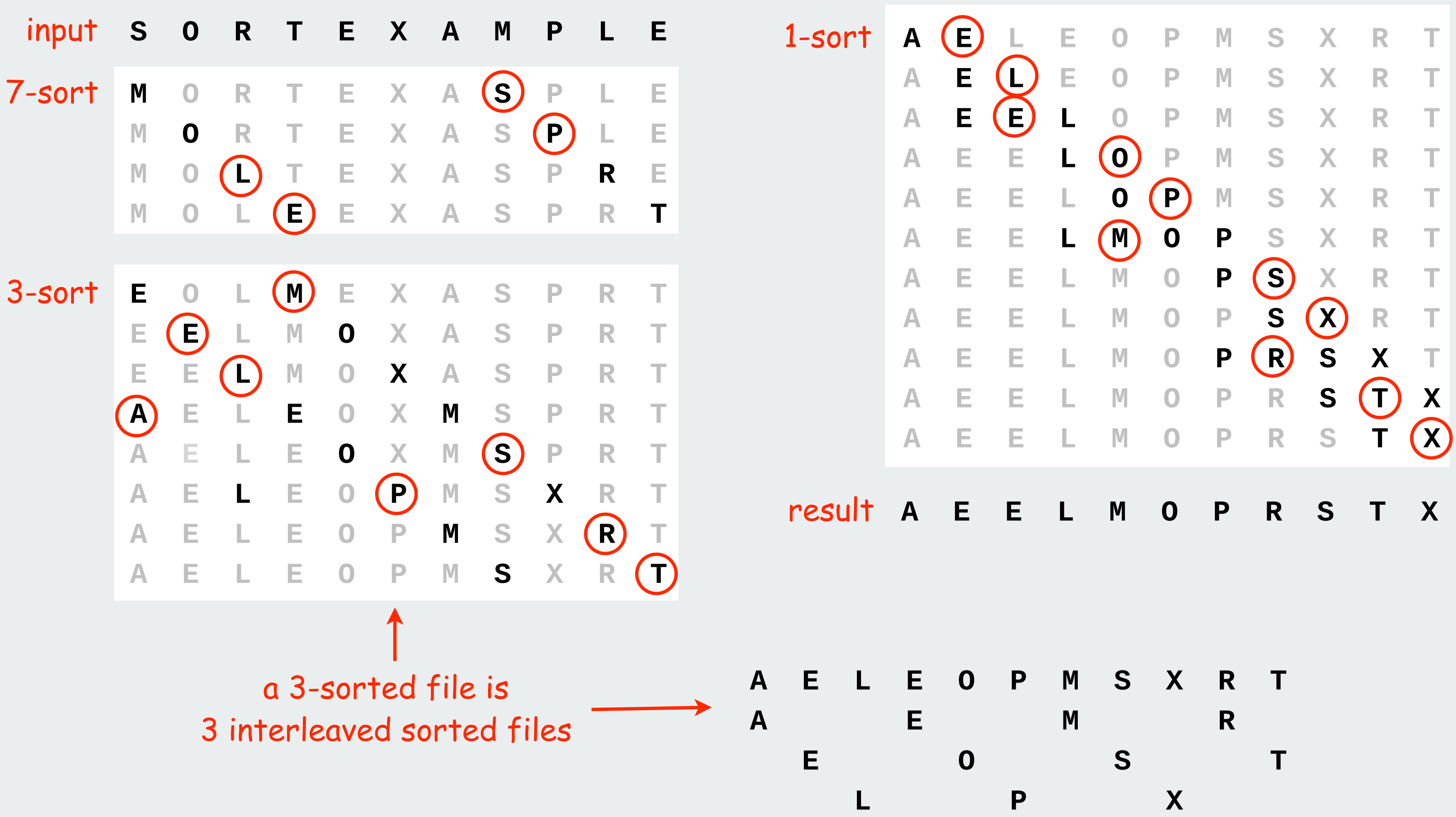
Visual representation of insertion sort


left of pointer is in sorted order
right of pointer is untouched
a[i]
i

Reason it is slow: data movement

Shellsort

Idea: move elements more than one position at a time
by h-sorting the file for a decreasing sequence of values of h



```
 input  S O R T E X A M P L E

7-sort  M O R T E X A S P L E
        M O R T E X A S P L E
        M O L T E X A S P R E
        M O L E E X A S P R T

3-sort  E O L M E X A S P R T
        E E L M O X A S P R T
        E E L M O X A S P R T
        A E L E O X M S P R T
        A E L E O X M S P R T
        A E L E O P M S X R T
        A E L E O P M S X R T
        A E L E O P M S X R T

1-sort  A E L E O P M S X R T
        A E L E O P M S X R T
        A E E L O P M S X R T
        A E E L O P M S X R T
        A E E L O P M S X R T
        A E E L M O P S X R T
        A E E L M O P S X R T
        A E E L M O P S X R T
        A E E L M O P R S X T
        A E E L M O P R S T X
        A E E L M O P R S T X

result  A E E L M O P R S T X
```

a 3-sorted file is
3 interleaved sorted files

```
A E L E O P M S X R T
A     E     M     R
  E     O     S     T
    L     P     X
```

Shellsort

Idea: move elements more than one position at a time
by h-sorting the file for a decreasing sequence of values of h

Use insertion sort, modified to h-sort

magic increment
sequence

big increments:
small subfiles

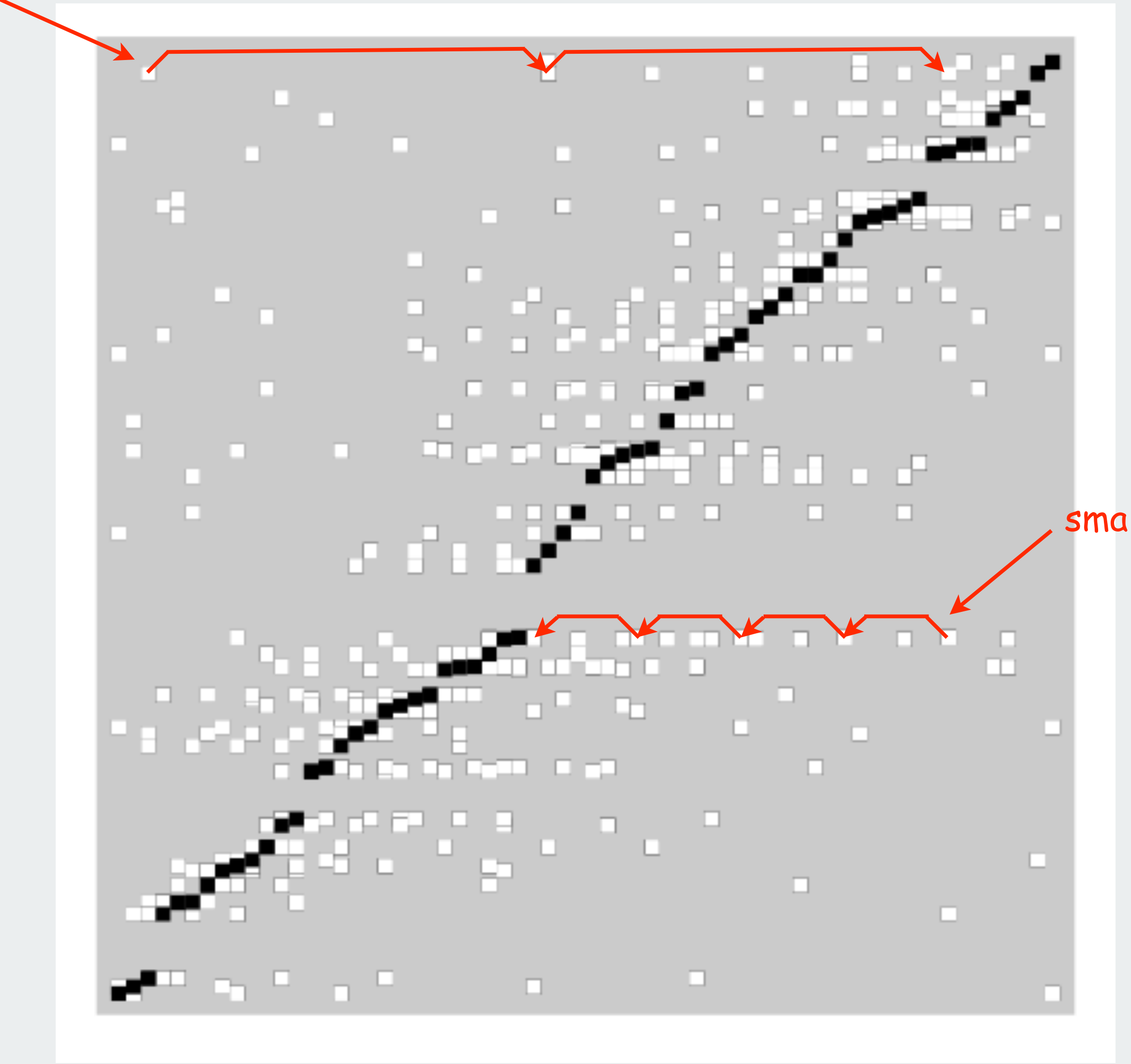
small increments:
subfiles nearly in order

method of choice for both
small subfiles
subfiles nearly in order

insertion sort!

```
public static void sort(double[] a)
{
    int N = a.length;
    int[] incs = { 1391376, 463792, 198768, 86961,
                   33936, 13776, 4592, 1968, 861,
                   336, 112, 48, 21, 7, 3, 1 };
    for (int k = 0; k < incs.length; k++)
    {
        int h = incs[k];
        for (int i = h; i < N; i++)
            for (int j = i; j >= h; j-= h)
                if (less(a[j], a[j-h]))
                     exch(a, j, j-h);
                else break;
    }
}
```

Visual representation of shellsort



big increment

small increment

Bottom line: substantially faster!

Analysis of shellsort

Model has not yet been discovered (!)

N	comparisons	$N^{1.289}$	2.5 N lg N
5,000	93	58	106
10,000	209	143	230
20,000	467	349	495
40,000	1022	855	1059
80,000	2266	2089	2257

measured in thousands

Why are we interested in shellsort?

Example of simple idea leading to substantial performance gains

Useful in practice

- fast unless file size is huge
- tiny, fixed footprint for code (used in embedded systems)
- hardware sort prototype

Simple algorithm, nontrivial performance, interesting questions

- asymptotic growth rate?
- best sequence of increments?
- average case performance?

Your first open problem in algorithmics (see Section 6.8):

Find a better increment sequence

```
mail rs@cs.princeton.edu
```

Lesson: some good algorithms are still waiting discovery

- rules of the game
- shellsort
- **mergesort**
- quicksort
- animations

Mergesort (von Neumann, 1945(!))

Basic plan:

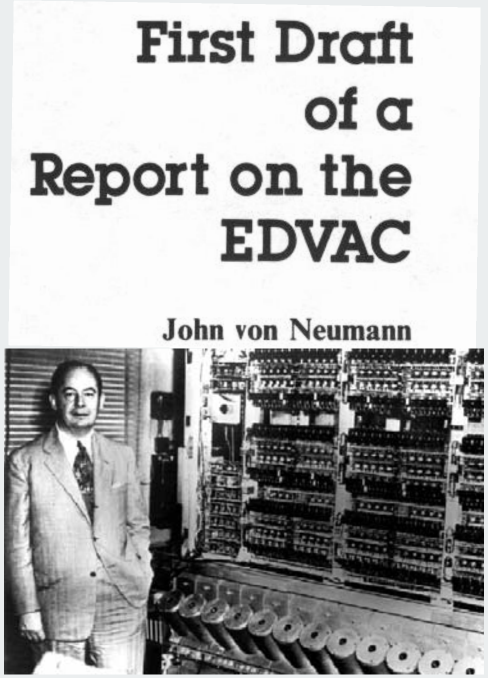
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

plan

M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

trace

		a[i]															
lo	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
0	1	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
2	3	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
0	3	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
4	5	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
6	7	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
4	7	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
0	7	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
8	9	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
10	11	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
8	11	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
12	13	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
14	15	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
12	15	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
8	15	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
0	15	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

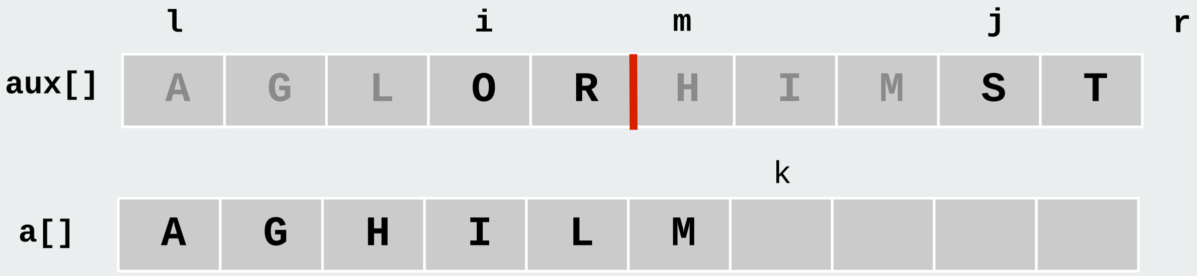

First Draft of a Report on the EDVAC

John von Neumann

Merging

Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently? Use an auxiliary array.



l i m j r

aux[] A G L O R H I M S T

k

a[] A G H I L M

copy →

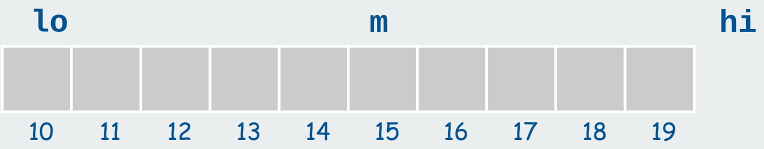
merge →

```
private static void merge(Comparable[] a,
                          Comparable[] aux, int l, int m, int r)
{
   for (int k = l; k < r; k++) aux[k] = a[k];
   int i = l, j = m;
   for (int k = l; k < r; k++)
      if      (i >= m)                    a[k] = aux[j++];
      else if (j >= r)                    a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                                a[k] = aux[i++];

}
```

see book for a trick to eliminate these

Mergesort: Java implementation of recursive sort

```
public class Merge
{
   private static void sort(Comparable[] a,
                            Comparable[] aux, int lo, int hi)
   {
      if (hi <= lo + 1) return;
      int m = lo + (hi - lo) / 2;
      sort(a, aux, lo, m);
      sort(a, aux, m, hi);
      merge(a, aux, lo, m, hi);
   }

   public static void sort(Comparable[] a)
   {
      Comparable[] aux = new Comparable[a.length];
      sort(a, aux, 0, a.length);
   }
}
```



Mergesort analysis: Memory

Q. How much memory does mergesort require?

A. Too much!

- Original input array = N.
- Auxiliary array for merging = N.
- Local variables: constant.
- Function call stack: $\log_2 N$ [stay tuned].
- Total = 2N + O(log N).

cannot "fill the memory and sort"

Q. How much memory do other sorting algorithms require?

- N + O(1) for insertion sort and selection sort.
- In-place = N + O(log N).

Challenge for the bored. In-place merge. [Kronrud, 1969]

Mergesort analysis

Def. $T(N) \equiv$ number of array stores to mergesort an input of size N

$$= \underset{\substack{\uparrow \\ \text{left half}}}{T(N/2)} \quad + \quad \underset{\substack{\uparrow \\ \text{right half}}}{T(N/2)} \quad + \quad \underset{\substack{\uparrow \\ \text{merge}}}{N}$$

Mergesort recurrence $T(N) = 2\,T(N/2) + N$

for $N > 1$, with $T(1) = 0$

- not quite right for odd N
- same recurrence holds for many algorithms
- same for any input of size N
- comparison count slightly smaller because of array ends

Solution of Mergesort recurrence $T(N) \sim N \lg N$

$\lg N \equiv \log_2 N$

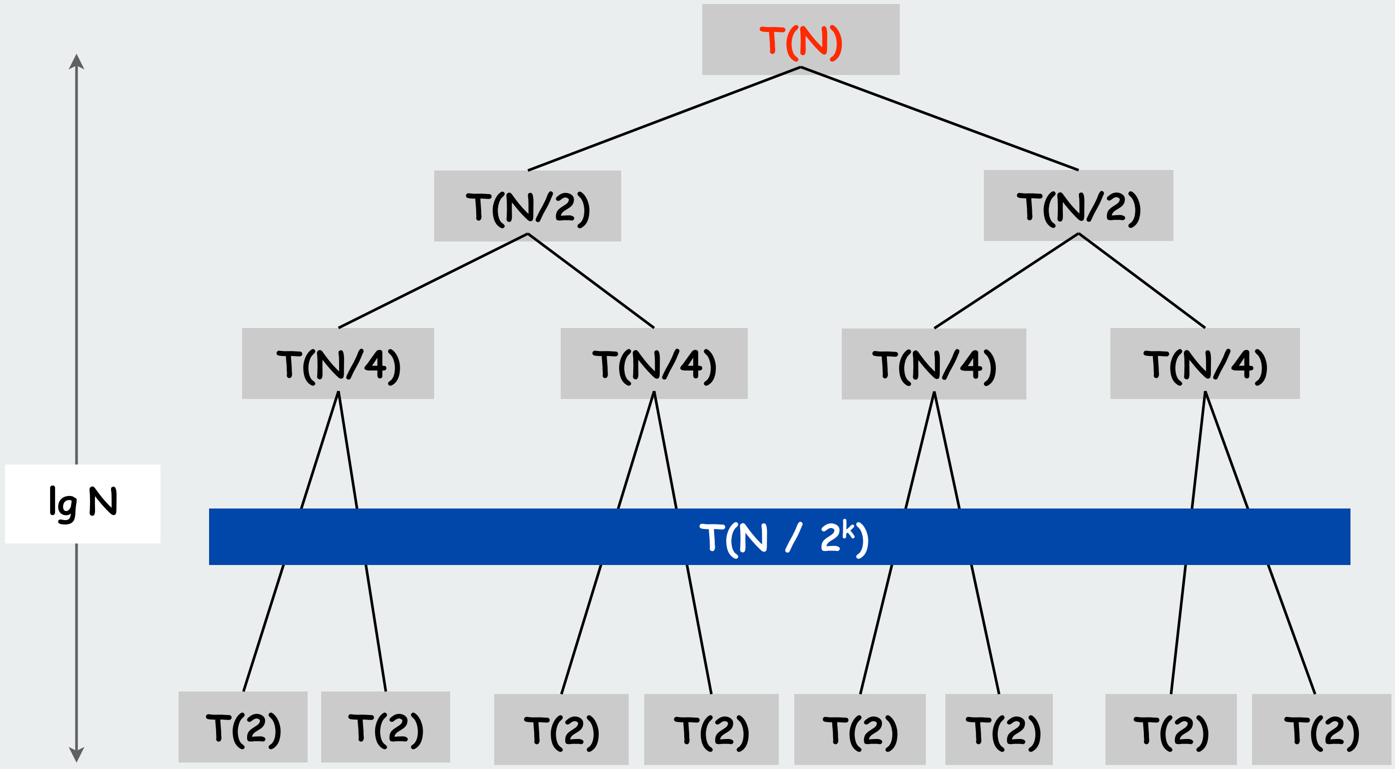
- true for all N
- easy to prove when N is a power of 2

Mergesort recurrence: Proof 1 (by recursion tree)

$$T(N) = 2\,T(N/2) + N$$

for $N > 1$, with $T(1) = 0$

(assume that N is a power of 2)



+

$N = N$

$2(N/2) = N$

. . .

$2^k(N/2^k) = N$

. . .

$N/2\ (2) = N$

$N \lg N$

$$T(N) = N \lg N$$

Mergesort recurrence: Proof 2 (by telescoping)

$$T(N) = 2\,T(N/2) + N$$
for $N > 1$, with $T(1) = 0$

(assume that N is a power of 2)

Pf.

$T(N) = 2\,T(N/2) + N$	given
$T(N)/N = 2\,T(N/2)/N + 1$	divide both sides by N
$= T(N/2)/(N/2) + 1$	algebra
$= T(N/4)/(N/4) + 1 + 1$	telescope (apply to first term)
$= T(N/8)/(N/8) + 1 + 1 + 1$	telescope again
$\ldots$	
$= T(N/N)/(N/N) + 1 + 1 + \ldots + 1$	stop telescoping, $T(1) = 0$
$= \lg N$	

$$T(N) = N \lg N$$

Mergesort recurrence: Proof 3 (by induction)

$$T(N) = 2\,T(N/2) + N$$

for $N > 1$, with $T(1) = 0$

(assume that N is a power of 2)

Claim. If $T(N)$ satisfies this recurrence, then $T(N) = N \lg N$.

Pf. [by induction on N]

- Base case: $N = 1$.
- Inductive hypothesis: $T(N) = N \lg N$
- Goal: show that $T(2N) + 2N \lg (2N)$.

$$\begin{aligned} T(2N) &= 2\,T(N) + 2N && \text{given} \\ &= 2\,N \lg N + 2\,N && \text{inductive hypothesis} \\ &= 2\,N\,(\lg (2N) - 1) + 2N && \text{algebra} \\ &= 2\,N \lg (2N) && \text{QED} \end{aligned}$$

Ex. (for COS 340). Extend to show that $T(N) \sim N \lg N$ for general N

Bottom-up mergesort

Basic plan:

- Pass through file, merging to double size of sorted subarrays.
- Do so for subarray sizes 1, 2, 4, 8, . . . , N/2, N.

proof 4 that mergesort uses N lgN compares

									a[i]								
lo	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		M	**E**	**R**	**G**	**E**	**S**	**O**	**R**	**T**	**E**	**X**	**A**	**M**	**P**	**L**	**E**
0	1	**E**	**M**	R	G	E	S	O	R	T	E	X	A	M	P	L	E
2	3	E	M	**G**	**R**	E	S	O	R	T	E	X	A	M	P	L	E
4	5	E	M	G	R	**E**	**S**	O	R	T	E	X	A	M	P	L	E
6	7	E	M	G	R	E	S	**O**	**R**	T	E	X	A	M	P	L	E
8	9	E	M	G	R	E	S	O	R	**E**	**T**	X	A	M	P	L	E
10	11	E	M	G	R	E	S	O	R	E	T	**A**	**X**	M	P	L	E
12	13	E	M	G	R	E	S	O	R	E	T	A	X	**M**	**P**	L	E
14	15	E	M	G	R	E	S	O	R	E	T	A	X	M	P	**E**	**L**
0	3	**E**	**G**	**M**	**R**	E	S	O	R	E	T	A	X	M	P	E	L
4	7	E	G	M	R	**E**	**O**	**R**	**S**	E	T	A	X	M	P	E	L
8	11	E	E	G	M	O	R	R	S	**A**	**E**	**T**	**X**	M	P	E	L
12	15	E	E	G	M	O	R	R	S	A	E	T	X	**E**	**L**	**M**	**P**
0	7	**E**	**E**	**G**	**M**	**O**	**R**	**R**	**S**	A	E	T	X	E	L	M	P
8	15	E	E	G	M	O	R	R	S	**A**	**E**	**E**	**L**	**M**	**P**	**T**	**X**
0	15	**A**	**E**	**E**	**E**	**E**	**G**	**L**	**M**	**M**	**O**	**P**	**R**	**R**	**S**	**T**	**X**

No recursion needed!

Bottom-up Mergesort: Java implementation

```
public class Merge
{
   private static void merge(Comparable[] a, Comparable[] aux,
                             int l, int m, int r)
   {
      for (int i = l; i < m; i++) aux[i] = a[i];
      for (int j = m; j < r; j++) aux[j] = a[m + r - j - 1];
      int i = l, j = r - 1;
      for (int k = l; k < r; k++)
          if (less(aux[j], aux[i])) a[k] = aux[j--];
          else                      a[k] = aux[i++];

   }

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      Comparable[] aux = new Comparable[N];
      for (int m = 1; m < N; m = m+m)
         for (int i = 0; i < N-m; i += m+m)
            merge(a, aux, i, i+m, Math.min(i+m+m, N));
   }
}
```

tricky merge
that uses sentinel
(see Program 8.2) →

Concise industrial-strength code if you have the space

Mergesort: Practical Improvements

Use sentinel.

- Two statements in inner loop are array-bounds checking.
- Reverse one subarray so that largest element is sentinel (Program 8.2)

Use insertion sort on small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 7 elements.

Stop if already sorted.

- Is biggest element in first half ≤ smallest element in second half?
- Helps for nearly ordered lists.

Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

See Program 8.4 (or Java system sort)

Sorting Analysis Summary

Running time estimates:

- Home pc executes 10^8 comparisons/second.
- Supercomputer executes 10^{12} comparisons/second.

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

Mergesort (N log N)

thousand	million	billion
instant	1 sec	18 min
instant	instant	instant

Lesson. Good algorithms are better than supercomputers.

Good enough?

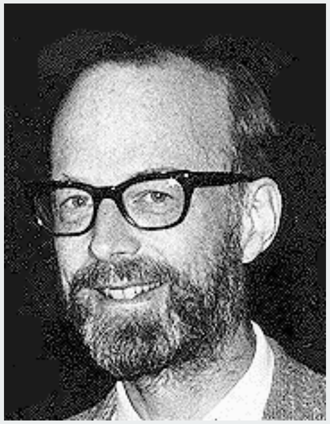
18 minutes might be too long for some applications

- rules of the game
- shellsort
- mergesort
- quicksort
- animations

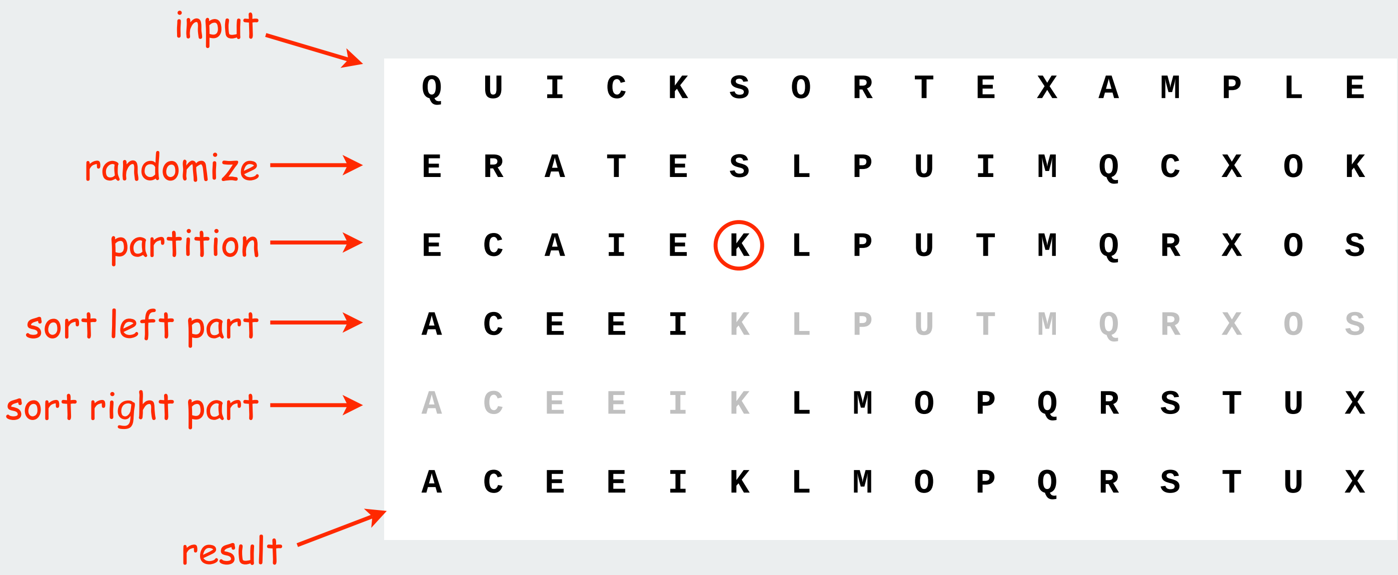
Quicksort (Hoare, 1959)

Basic plan.

- Shuffle the array.
- Partition so that, for some `i`
  - element `a[i]` is in place
  - no larger element to the left of `i`
  - no smaller element to the right of `i`
- Sort each piece recursively.



Sir Charles Antony Richard Hoare
1980 Turing Award

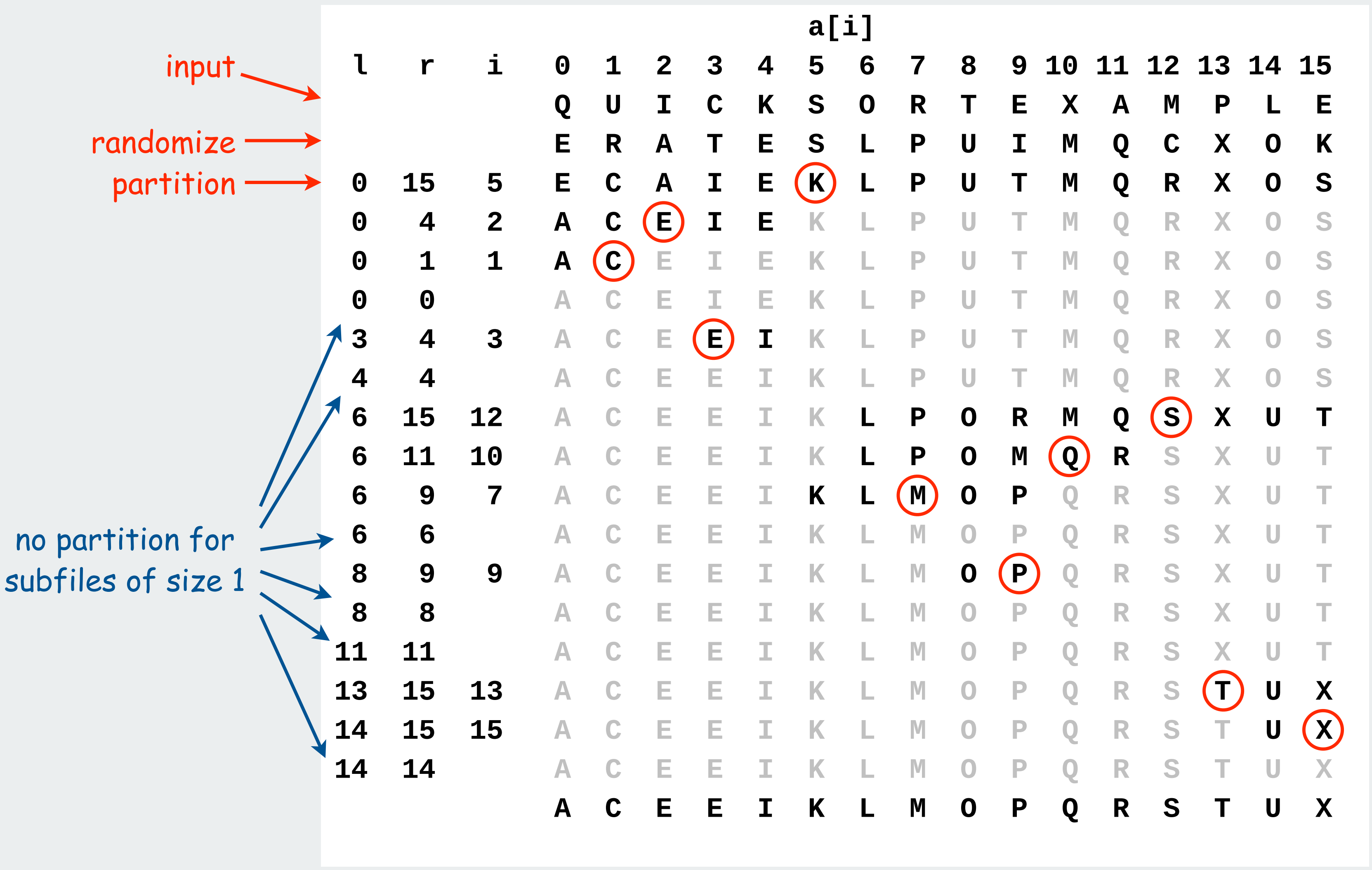


input	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
randomize	E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
sort left part	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
sort right part	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort: Java code for recursive sort

```
public class Quick
{

   public static void sort(Comparable[] a)
   {
      StdRandom.shuffle(a);
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int l, int r)
   {
      if (r <= l) return;
      int m = partition(a, l, r);
      sort(a, l, m-1);
      sort(a, m+1, r);
  }
}
```

Quicksort trace



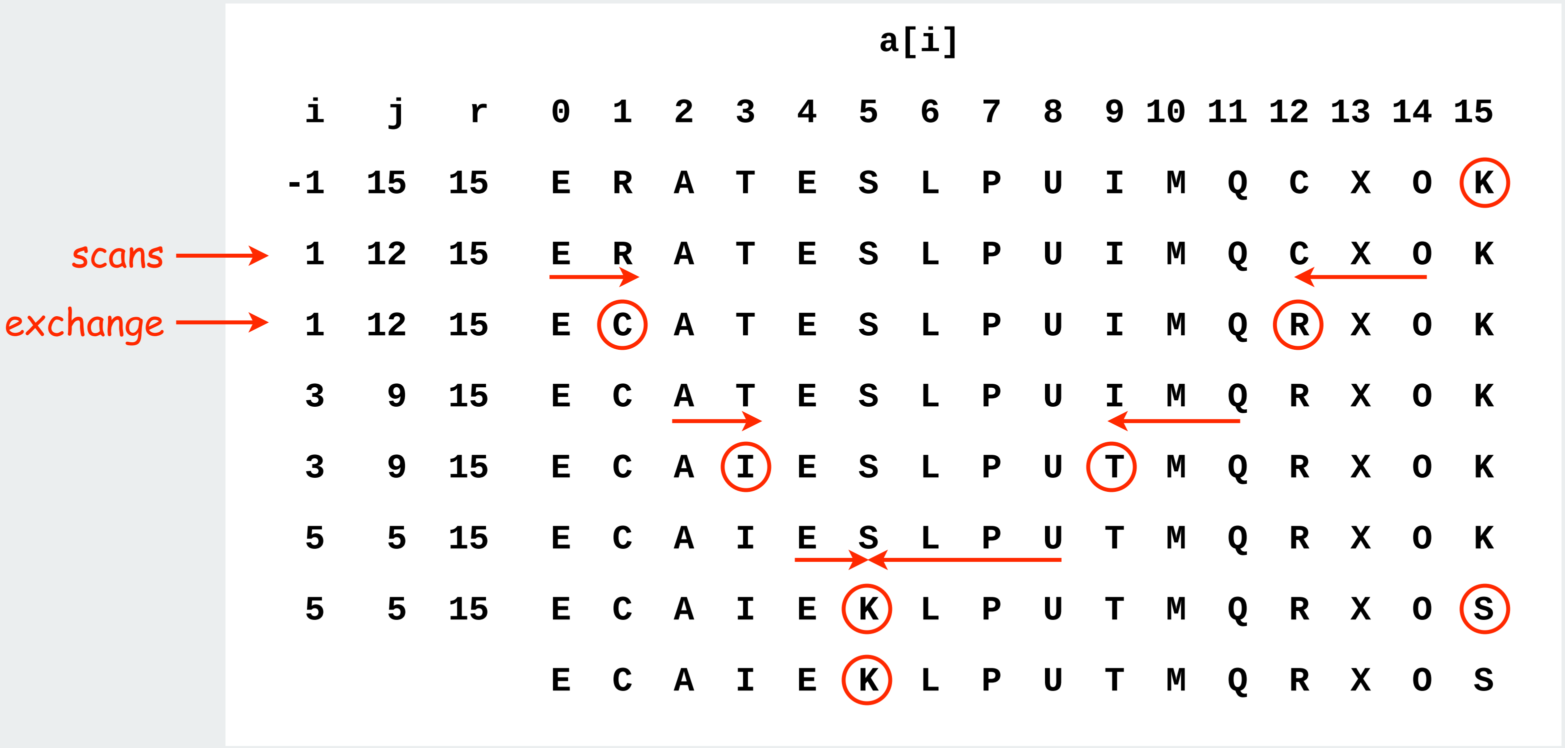
			a[i]															
l	r	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
			E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
0	15	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
0	4	2	A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
0	1	1	A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
0	0		A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
3	4	3	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
4	4		A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
6	15	12	A	C	E	E	I	K	L	P	O	R	M	Q	S	X	U	T
6	11	10	A	C	E	E	I	K	L	P	O	M	Q	R	S	X	U	T
6	9	7	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
6	6		A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
8	9	9	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
8	8		A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
11	11		A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
13	15	13	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
14	15	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
14	14		A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
			A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

array contents after each recursive sort

Quicksort partitioning

Basic plan:

- scan from left for an item that belongs on the right
- scan from right for item item that belongs on the left
- exchange
- continue until pointers cross

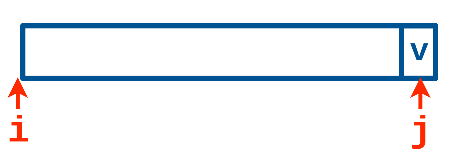


				a[i]															
	i	j	r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	-1	15	15	E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
scans	1	12	15	E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
exchange	1	12	15	E	C	A	T	E	S	L	P	U	I	M	Q	R	X	O	K
	3	9	15	E	C	A	T	E	S	L	P	U	I	M	Q	R	X	O	K
	3	9	15	E	C	A	I	E	S	L	P	U	T	M	Q	R	X	O	K
	5	5	15	E	C	A	I	E	S	L	P	U	T	M	Q	R	X	O	K
	5	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
				E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

array contents before and after each exchange

Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a, int l, int r)
{
   int i = l - 1;
   int j = r;
   while(true)
   {
```

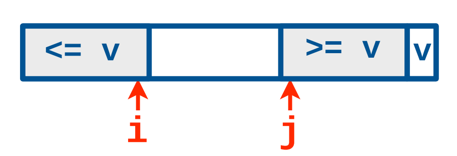


```
      while (less(a[++i], a[r]))
         if (i == r) break;
```

find item on left to swap

```
      while (less(a[r], a[--j]))
         if (j == l) break;
```

find item on right to swap



```
      if (i >= j) break;
```

check if pointers cross

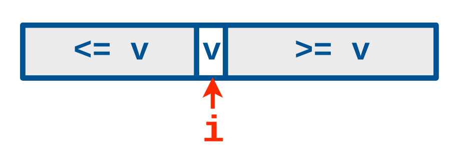
```
      exch(a, i, j);
   }
```

swap

```
   exch(a, i, r);
   return i;
}
```

swap with partitioning item

return index of item now known to be in place



Quicksort Implementation details

Partitioning in-place. Using a spare array makes partitioning easier, but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is a bit trickier than it might seem.

Staying in bounds. The `(i == r)` test is redundant, but the `(j == l)` test is not.

Preserving randomness. Shuffling is key for performance guarantee.

Equal keys. When duplicates are present, it is (counter-intuitively) best to stop on elements equal to partitioning element.

Quicksort: Average-case analysis

Theorem. The average number of comparisons C_N to quicksort a random file of N elements is about 2N ln N.

- The precise recurrence satisfies $C_0 = C_1 = 0$ and for $N \geq 2$:

$$C_N = \underset{\substack{\uparrow \\ \text{partition}}}{N + 1} + ((C_0 + C_{N-1}) + \ldots + (\underset{\substack{\uparrow \\ \text{left}}}{C_{k-1}} + \underset{\substack{\uparrow \\ \text{right}}}{C_{N-k}}) + \ldots + (C_{N-1} + C_1)) / \underset{\substack{\uparrow \\ \text{partitioning} \\ \text{probability}}}{N}$$

$$= N + 1 + 2\,(C_0 \ldots + C_{k-1} + \ldots + C_{N-1}) / N$$

- Multiply both sides by N

$$NC_N = N(N + 1) + 2\,(C_0 \ldots + C_{k-1} + \ldots + C_{N-1})$$

- Subtract the same formula for N-1:

$$NC_N - (N - 1)C_{N-1} = N(N + 1) - (N - 1)N + 2\,C_{N-1}$$

- Simplify:

$$NC_N = (N + 1)C_{N-1} + 2N$$

Quicksort: Average Case

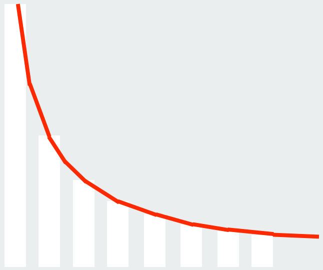
$$N C_N = (N+1) C_{N-1} + 2N$$

- Divide both sides by N(N+1) to get a telescoping sum:

$$\begin{aligned} C_N / (N+1) &= C_{N-1} / N + 2/(N+1) \\ &= C_{N-2}/(N-1) + 2/N + 2/(N+1) \\ &= C_{N-3}/(N-2) + 2/(N-1) + 2/N + 2/(N+1) \\ &= 2\left(1 + 1/2 + 1/3 + \ldots + 1/N + 1/(N+1)\right) \end{aligned}$$

- Approximate the exact answer by an integral:

$$\begin{aligned} C_N &\approx 2(N+1)\left(1 + 1/2 + 1/3 + \ldots + 1/N\right) \\ &= 2(N+1) H_N \approx 2(N+1)\int_1^N dx/x \end{aligned}$$



- Finally, the desired result:

$$C_N \approx 2(N+1) \ln N \approx 1.39 N \lg N$$

Quicksort: Summary of performance characteristics

Worst case. Number of comparisons is quadratic.

- $N + (N-1) + (N-2) + \ldots + 1 \approx N^2 / 2$.
- More likely that your computer is struck by lightning.

Average case. Number of comparisons is $\sim 1.39 N \lg N$.

- 39% more comparisons than mergesort.
- but faster than mergesort in practice because of lower cost of other high-frequency operations.

Random shuffle

- probabilistic guarantee against worst case
- basis for math model that can be validated with experiments

Caveat emptor. Many textbook implementations go quadratic if input:

- Is sorted.
- Is reverse sorted.
- Has many duplicates (even if randomized)! [stay tuned]

Sorting analysis summary

Running time estimates:

- Home pc executes 10^8 comparisons/second.
- Supercomputer executes 10^{12} comparisons/second.

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

Mergesort (N log N)

thousand	million	billion
instant	1 sec	18 min
instant	instant	instant

Quicksort (N log N)

thousand	million	billion
instant	0.3 sec	6 min
instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

Quicksort: Practical improvements

Median of sample.

- Best choice of pivot element = median.
- But how to compute the median?
- Estimate true median by taking median of sample.

Insertion sort small files.

- Even quicksort has too much overhead for tiny files.
- Can delay insertion sort until end.

Optimize parameters.

≈ 12/7 N log N comparisons

- Median-of-3 random elements.
- Cutoff to insertion sort for ≈ 10 elements.

Non-recursive version.

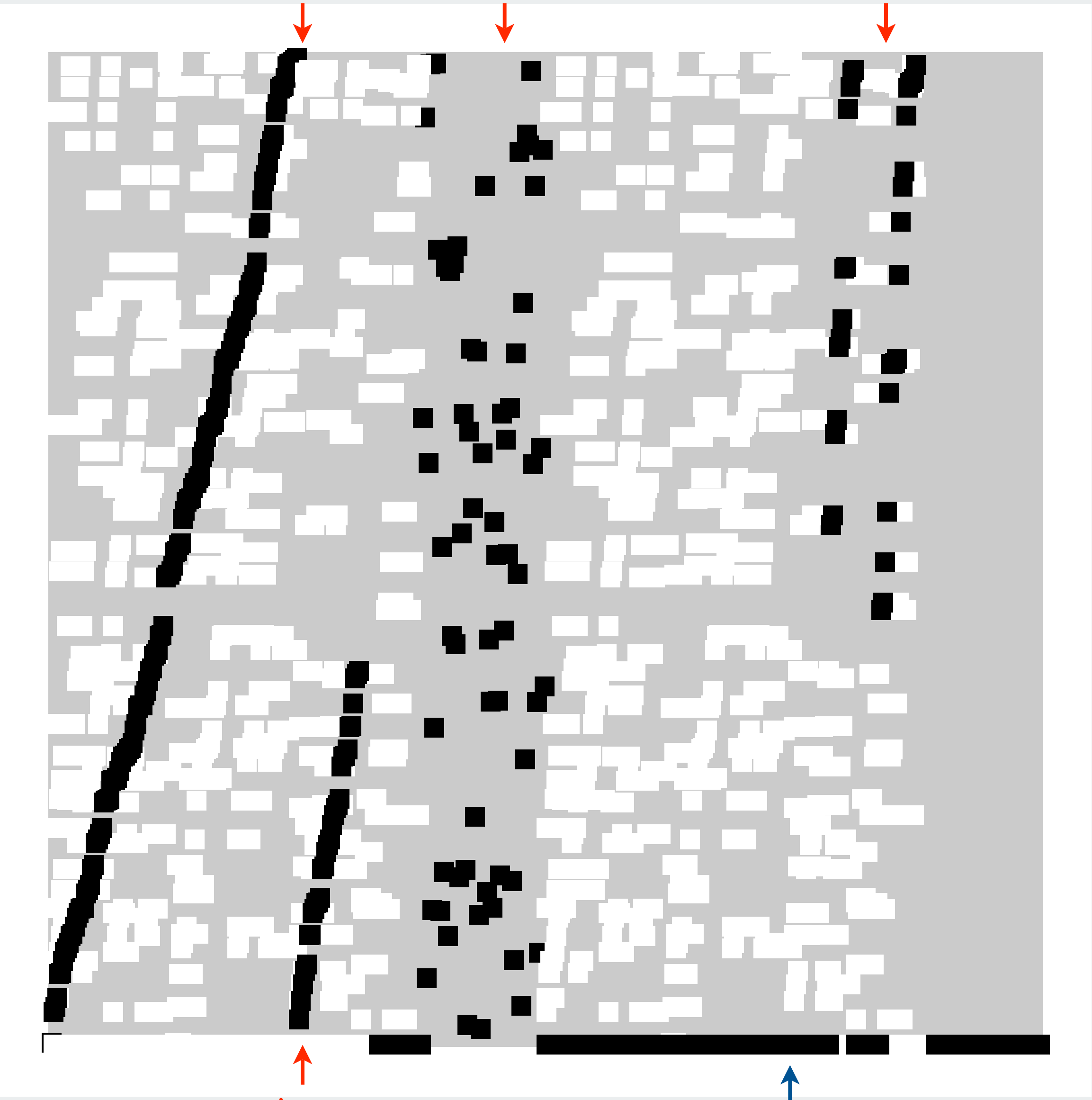
- Use explicit stack.
- Always sort smaller half first.

guarantees O(log N) stack size

All validated with refined math models and experiments

- rules of the game
- shellsort
- mergesort
- quicksort
- **animations**

Mergesort animation



done

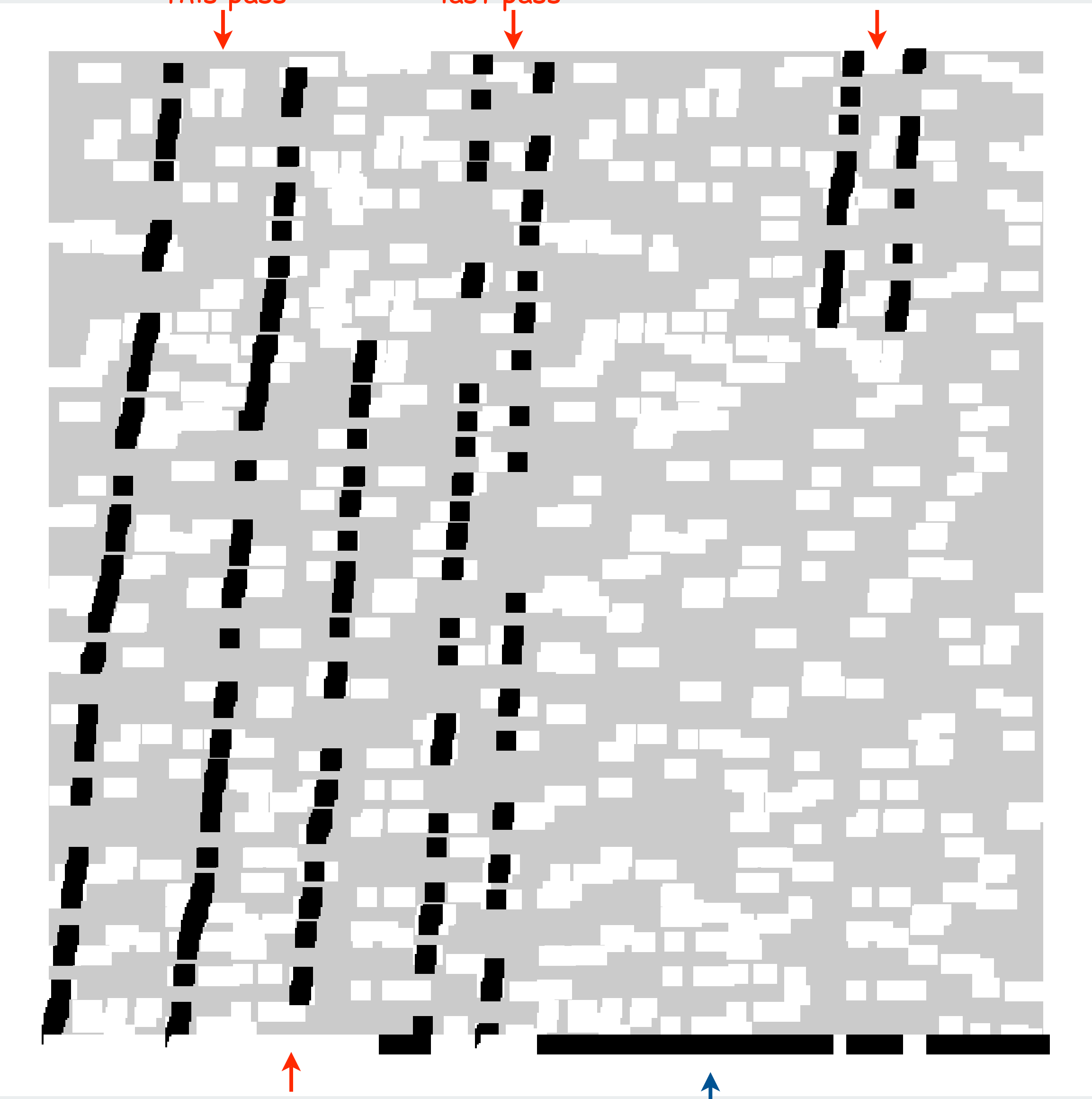
untouched

merge in progress
input

merge in progress
output

auxiliary array

Bottom-up mergesort animation



this pass

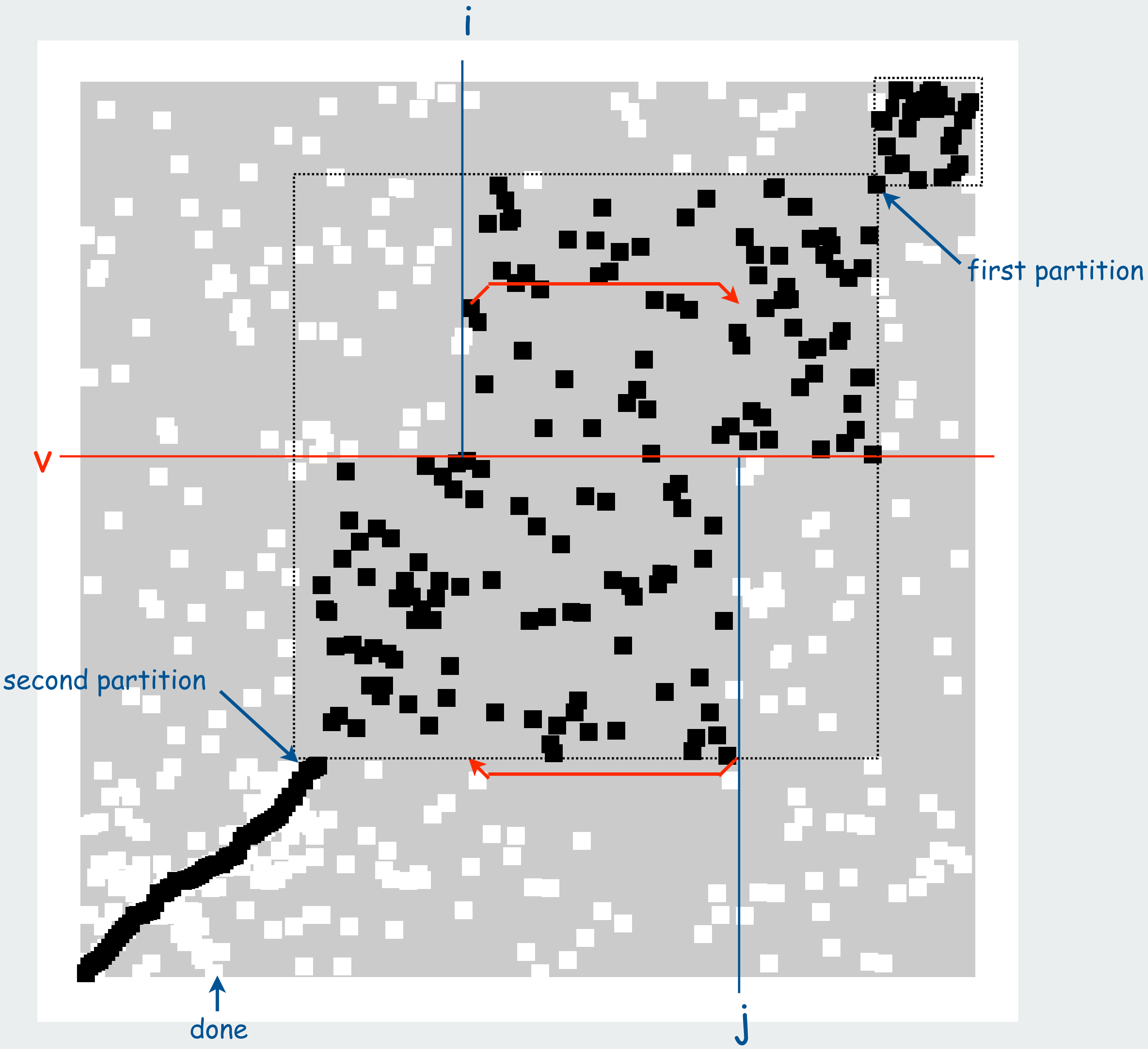
last pass

merge in progress input

merge in progress output

auxiliary array

Quicksort animation



Advanced Topics in Sorting

- complexity
- system sorts
- duplicate keys
- comparators

- complexity
- system sorts
- duplicate keys
- comparators

Complexity of sorting

Computational complexity. Framework to study efficiency of algorithms for solving a particular problem X.

Machine model. Focus on fundamental operations.

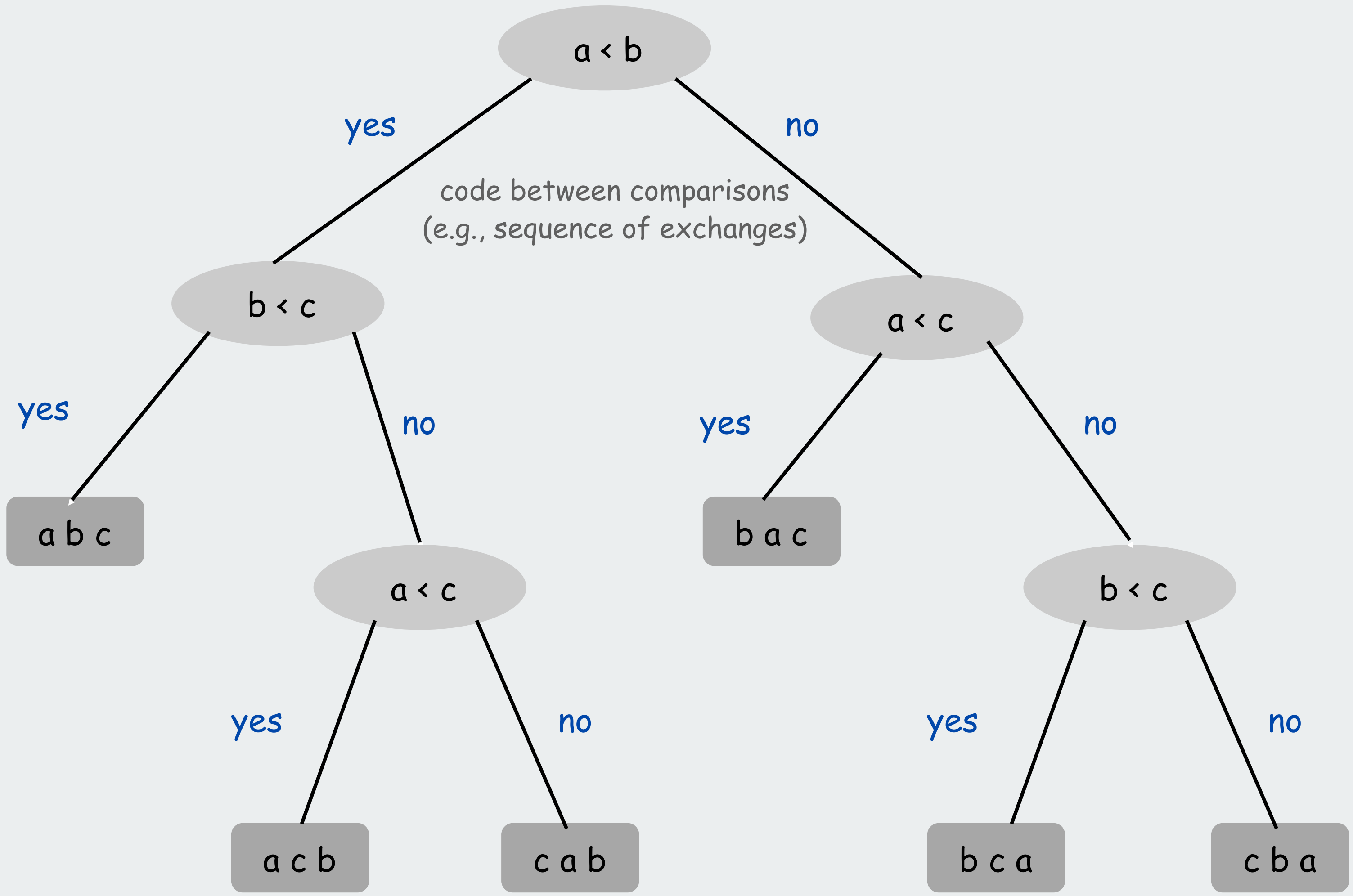
Upper bound. Cost guarantee provided by some algorithm for X.
Lower bound. Proven limit on cost guarantee of any algorithm for X.
Optimal algorithm. Algorithm with best cost guarantee for X.

lower bound ~ upper bound

Example: sorting.

- Machine model = # comparisons ← access information only through compares
- Upper bound = N lg N from mergesort.
- Lower bound ?

Decision Tree

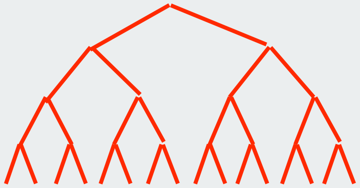


Comparison-based lower bound for sorting

Theorem. Any comparison based sorting algorithm must use more than N lg N - 1.44 N comparisons in the worst-case.

Pf.

- Assume input consists of N distinct values a_1 through a_N.
- Worst case dictated by tree height h.
- N ! different orderings.
- (At least) one leaf corresponds to each ordering.
- Binary tree with N ! leaves cannot have height less than lg (N!)



$$\begin{aligned} h &\geq \lg N! \\ &\geq \lg (N / e)^N \quad \longleftarrow \text{Stirling's formula} \\ &= N \lg N - N \lg e \\ &\geq N \lg N - 1.44\, N \end{aligned}$$

Complexity of sorting

Upper bound. Cost guarantee provided by some algorithm for X.
Lower bound. Proven limit on cost guarantee of any algorithm for X.
Optimal algorithm. Algorithm with best cost guarantee for X.

Example: sorting.

- Machine model = # comparisons
- Upper bound = $N \lg N$ (mergesort)
- Lower bound = $N \lg N - 1.44 N$

Mergesort is optimal (to within a small additive factor)

lower bound ≈ upper bound

First goal of algorithm design: optimal algorithms

Complexity results in context

Mergesort is optimal (to within a small additive factor)

Other operations?

- statement is only about number of compares
- quicksort is faster than mergesort (lower use of other operations)

Space?

- mergesort is not optimal with respect to space usage
- insertion sort, selection sort, shellsort, quicksort are space-optimal
- is there an algorithm that is both time- and space-optimal?

stay tuned for heapsort

Nonoptimal algorithms may be better in practice

- statement is only about guaranteed worst-case performance
- quicksort's probabilistic guarantee is just as good in practice

Lessons

- use theory as a guide
- know your algorithms

don't try to design an algorithm that uses half as many compares as mergesort

use quicksort when time and space are critical

Example: Selection

Find the k^{th} largest element.

- Min: k = 1.
- Max: k = N.
- Median: k = N/2.

Applications.

- Order statistics.
- Find the "top k"

Use theory as a guide

- easy O(N log N) upper bound: sort, return a[k]
- easy O(N) upper bound for some k: min, max
- easy Ω(N) lower bound: must examine every element

Which is true?

- Ω(N log N) lower bound? [is selection as hard as sorting?]
- O(N) upper bound? [linear algorithm for all k]

Complexity results in context (continued)

Lower bound may not hold if the algorithm has information about

- the key values
- their initial arrangement

Partially ordered arrays. Depending on the initial order of the input, we may not need N lg N compares.

insertion sort requires O(N) compares on an already sorted array

Duplicate keys. Depending on the input distribution of duplicates, we may not need N lg N compares.

stay tuned for 3-way quicksort

Digital properties of keys. We can use digit/character comparisons instead of key comparisons for numbers and strings.

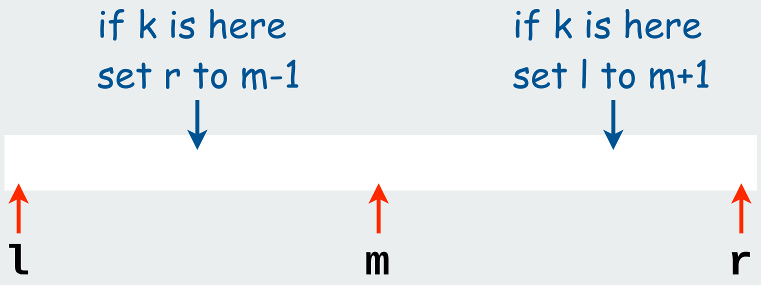
stay tuned for radix sorts

Selection: quick-select algorithm

Partition array so that:

- element `a[m]` is in place
- no larger element to the left of `m`
- no smaller element to the right of `m`

Repeat in one subarray, depending on `m`.



if k is here
set r to m-1

if k is here
set l to m+1

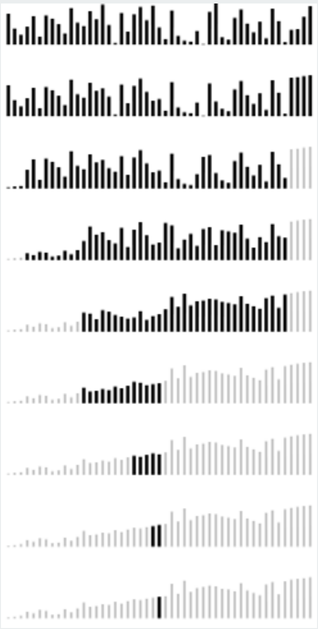
l

m

r

Finished when m = k ← a[k] is in place, no larger element to the left, no smaller element to the right

```
public static void select(Comparable[] a, int k)
{
    StdRandom.shuffle(a);
    int l = 0;
    int r = a.length - 1;
    while (r > l)
    {
       int i = partition(a, l, r);
       if      (m > k) r = m - 1;
       else if (m < k) l = m + 1;
       else return;
    }
}
```



Quick-select analysis

Theorem. Quick-select takes linear time on average.

Pf.

- Intuitively, each partitioning step roughly splits array in half.
- $N + N/2 + N/4 + \ldots + 1 \approx 2N$ comparisons.
- Formal analysis similar to quicksort analysis:

$$C_N = 2N + k \ln(N/k) + (N-k) \ln(N/(N-k))$$

Ex: $(2 + 2 \ln 2) N$ comparisons to find the median

Note. Might use $\sim N^2/2$ comparisons, but as with quicksort, the random shuffle provides a probabilistic guarantee.

Theorem. [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] There exists a selection algorithm that take linear time in the worst case.

Note. Algorithm is far too complicated to be useful in practice.

Use theory as a guide

- still worthwhile to seek practical linear-time (worst-case) algorithm
- until one is discovered, use quick-select if you don't need a full sort

- complexity
- **system sorts**
- duplicate keys
- comparators

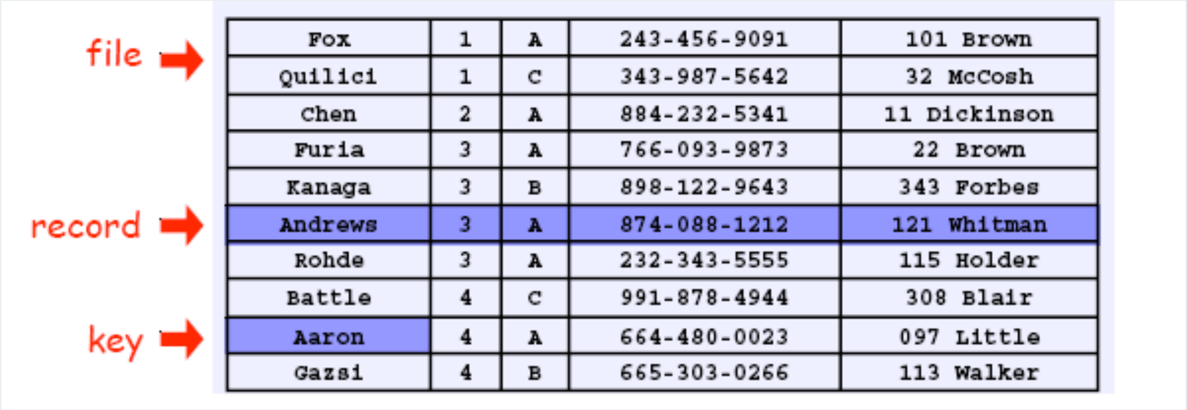
Sorting Challenge 1

Problem: sort a file of huge records with tiny keys.

Ex: reorganizing your MP3 files.

Which sorting method to use?

1. mergesort
2. insertion sort
3. selection sort



file
record
key

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

Sorting Challenge 1

Problem: sort a file of huge records with tiny keys.
Ex: reorganizing your MP3 files.

Which sorting method to use?

1. mergesort probably no, selection sort simpler and faster
2. insertion sort no, too many exchanges
3. selection sort YES, linear time under reasonable assumptions

Ex: 5,000 records, each 2 million bytes with 100-byte keys.

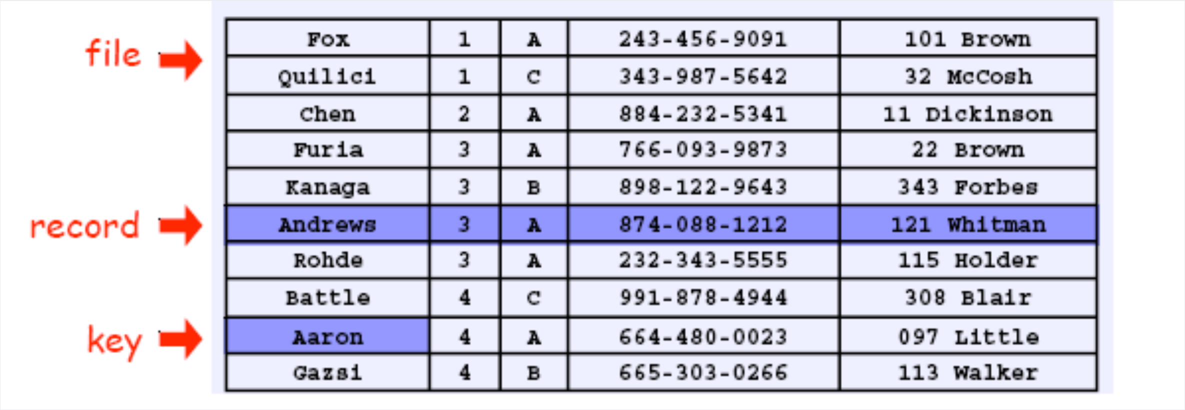
- Cost of comparisons: $100 \times 5000^2 / 2 = 1.25$ billion
- Cost of exchanges: $2{,}000{,}000 \times 5{,}000 = 10$ trillion
- Mergesort might be a factor of log (5000) slower.

Sorting Challenge 2

Problem: sort a huge randomly-ordered file of small records.
Ex: process transaction records for a phone company.

Which sorting method to use?

1. quicksort
2. insertion sort
3. selection sort


file
record
key

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

Sorting Challenge 2

Problem: sort a huge randomly-ordered file of small records.

Ex: process transaction records for a phone company.

Which sorting method to use?

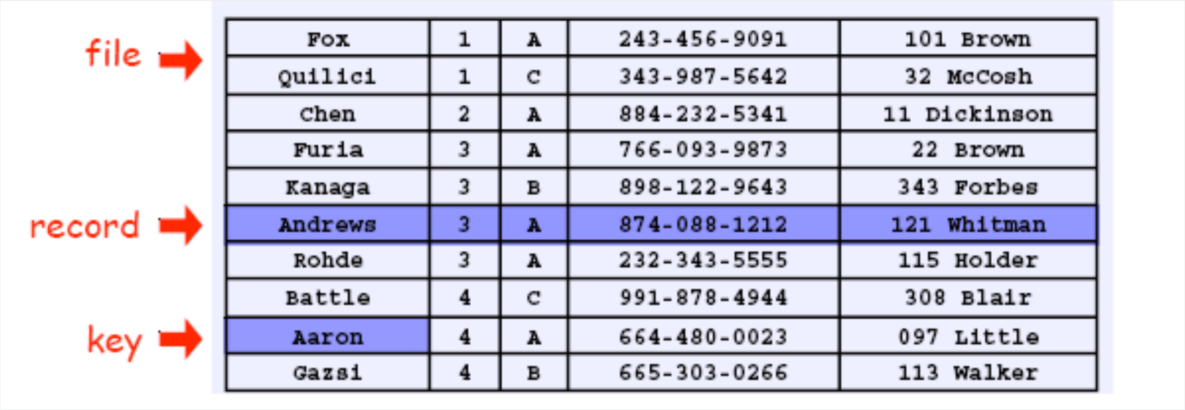
1.	quicksort	YES, it's designed for this problem
2.	insertion sort	no, quadratic time for randomly-ordered files
3.	selection sort	no, always takes quadratic time

Sorting Challenge 3

Problem: sort a huge number of tiny files (each file is independent)
Ex: daily customer transaction records.

Which sorting method to use?

1. quicksort
2. insertion sort
3. selection sort


file
record
key

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

Sorting Challenge 3

Problem: sort a huge number of tiny files (each file is independent)
Ex: daily customer transaction records.

Which sorting method to use?

1. quicksort no, too much overhead
2. insertion sort YES, much less overhead than system sort
3. selection sort YES, much less overhead than system sort

Ex: 4 record file.

- $4 N \log N + 35 = 70$
- $2N^2 = 32$

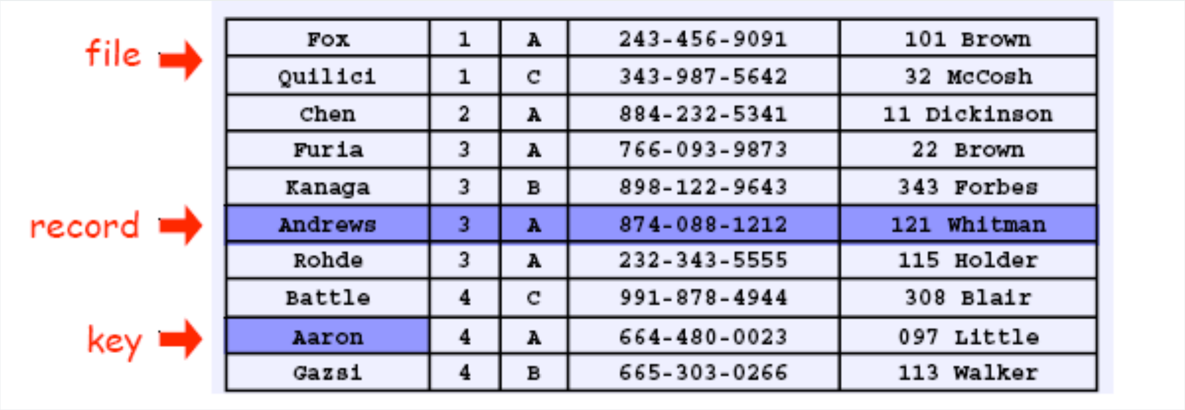
Sorting Challenge 4

Problem: sort a huge file that is already almost in order.

Ex: re-sort a huge database after a few changes.

Which sorting method to use?

1. quicksort
2. insertion sort
3. selection sort



file

record

key

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

Sorting Challenge 4

Problem: sort a huge file that is already almost in order.

Ex: re-sort a huge database after a few changes.

Which sorting method to use?

1. quicksort probably no, insertion simpler and faster
2. insertion sort YES, linear time for most definitions of "in order"
3. selection sort no, always takes quadratic time

Ex:

- `A B C D E F H I J G P K L M N O Q R S T U V W X Y Z`
- `Z A B C D E F G H I J K L M N O P Q R S T U V W X Y`

Sorting Applications

Sorting algorithms are essential in a broad variety of applications

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

obvious applications

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

problems become easy once items are in sorted order

- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Load balancing on a parallel computer.

. . .

non-obvious applications

Every system needs (and has) a system sort!

System sort: Which algorithm to use?

Many sorting algorithms to choose from

internal sorts.

- Insertion sort, selection sort, bubblesort, shaker sort.
- Quicksort, mergesort, heapsort, samplesort, shellsort.
- Solitaire sort, red-black sort, splaysort, Dobosiewicz sort, psort, ...

external sorts. Poly-phase mergesort, cascade-merge, oscillating sort.

radix sorts.

- Distribution, MSD, LSD.
- 3-way radix quicksort.

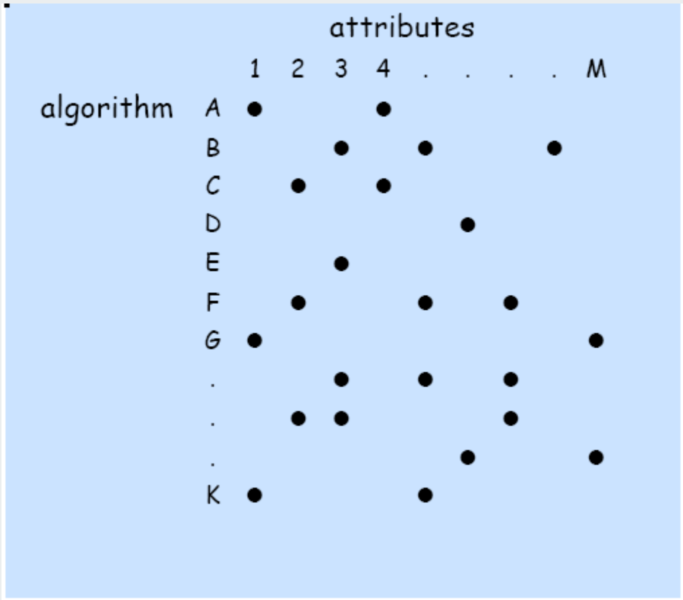
parallel sorts.

- Bitonic sort, Batcher even-odd sort.
- Smooth sort, cube sort, column sort.
- GPUsort.

System sort: Which algorithm to use?

Applications have diverse attributes

- Stable?
- Multiple keys?
- Deterministic?
- Keys all distinct?
- Multiple key types?
- Linked list or arrays?
- Large or small records?
- Is your file randomly ordered?
- Need guaranteed performance?



		attributes								
		1	2	3	4	.	.	.	.	M
algorithm	A	●			●					
	B			●		●			●	
	C		●		●					
	D						●			
	E			●						
	F		●			●		●		
	G	●								●
	.			●		●		●		
	.		●	●				●		
	.						●			●
	K	●				●				

many more combinations of attributes than algorithms

Elementary sort may be method of choice for some combination.
Cannot cover all combinations of attributes.

Q. Is the system sort good enough?
A. Maybe (no matter which algorithm it uses).

- complexity
- system sorts
- **duplicate keys**
- comparators

Duplicate keys

Often, purpose of sort is to bring records with duplicate keys together.

- Sort population by age.
- Finding collinear points.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge file.
- Small number of key values.

Mergesort with duplicate keys: always ~ N lg N compares

Quicksort with duplicate keys

- algorithm goes quadratic unless partitioning stops on equal keys!
- [many textbook and system implementations have this problem]
- 1990s Unix user found this problem in qsort()

Duplicate keys: the problem

Assume all keys are equal.
Recursive code guarantees that case will predominate!

Mistake: Put all keys equal to the partitioning element on one side

- easy to code
- guarantees N^2 running time when all keys equal

B A A B A B C C B C B   A A A A A A A A A A A

Recommended: Stop scans on keys equal to the partitioning element

- easy to code
- guarantees N lg N compares when all keys equal

B A A B A B C C B C B   A A A A A A A A A A A

Desirable: Put all keys equal to the partitioning element in place

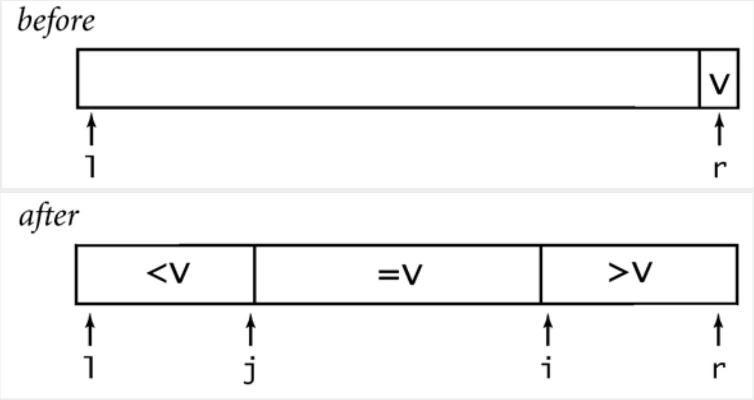
A A A B B B B B C C C   A A A A A A A A A A A

Common wisdom to 1990s: not worth adding code to inner loop

3-Way Partitioning

3-way partitioning. Partition elements into 3 parts:

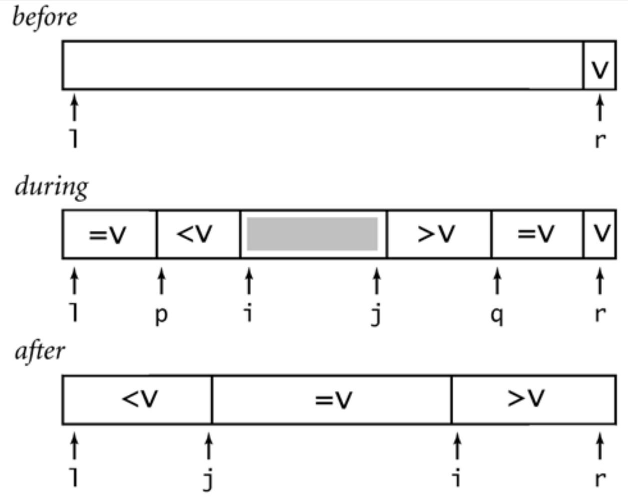
- Elements between i and j equal to partition element v.
- No larger elements to left of i.
- No smaller elements to right of j.



Dutch national flag problem.

- not done in practical sorts before mid-1990s.
- new approach discovered when fixing mistake in Unix qsort()
- now incorporated into Java system sort

Solution to Dutch national flag problem.



3-way partitioning (Bentley-McIlroy).

- Partition elements into 4 parts:
  - no larger elements to left of i
  - no smaller elements to right of j
  - equal elements to left of p
  - equal elements to right of q
- Afterwards, swap equal keys into center.

All the right properties.

- in-place.
- not much code.
- linear if keys are all equal.
- small overhead if no equal keys.

3-way Quicksort: Java Implementation

```
private static void sort(Comparable[] a, int l, int r)
{
   if (r <= l) return;
   int i = l-1, j = r;
   int p = l-1, q = r;

   while(true)
   {
      while (less(a[++i], a[r])) ;
      while (less(a[r], a[--j])) if (j == l) break;
      if (i >= j) break;
      exch(a, i, j);
      if (eq(a[i], a[r])) exch(a, ++p, i);
      if (eq(a[j], a[r])) exch(a, --q, j);
   }
   exch(a, i, r);

   j = i - 1;
   i = i + 1;
   for (int k = l  ; k <= p; k++) exch(a, k, j--);
   for (int k = r-1; k >= q; k--) exch(a, k, i++);

   sort(a, l, j);
   sort(a, i, r);
}
```

4-way partitioning

swap equal keys to left or right

swap equal keys back to middle

recursively sort left and right

Duplicate keys: lower bound

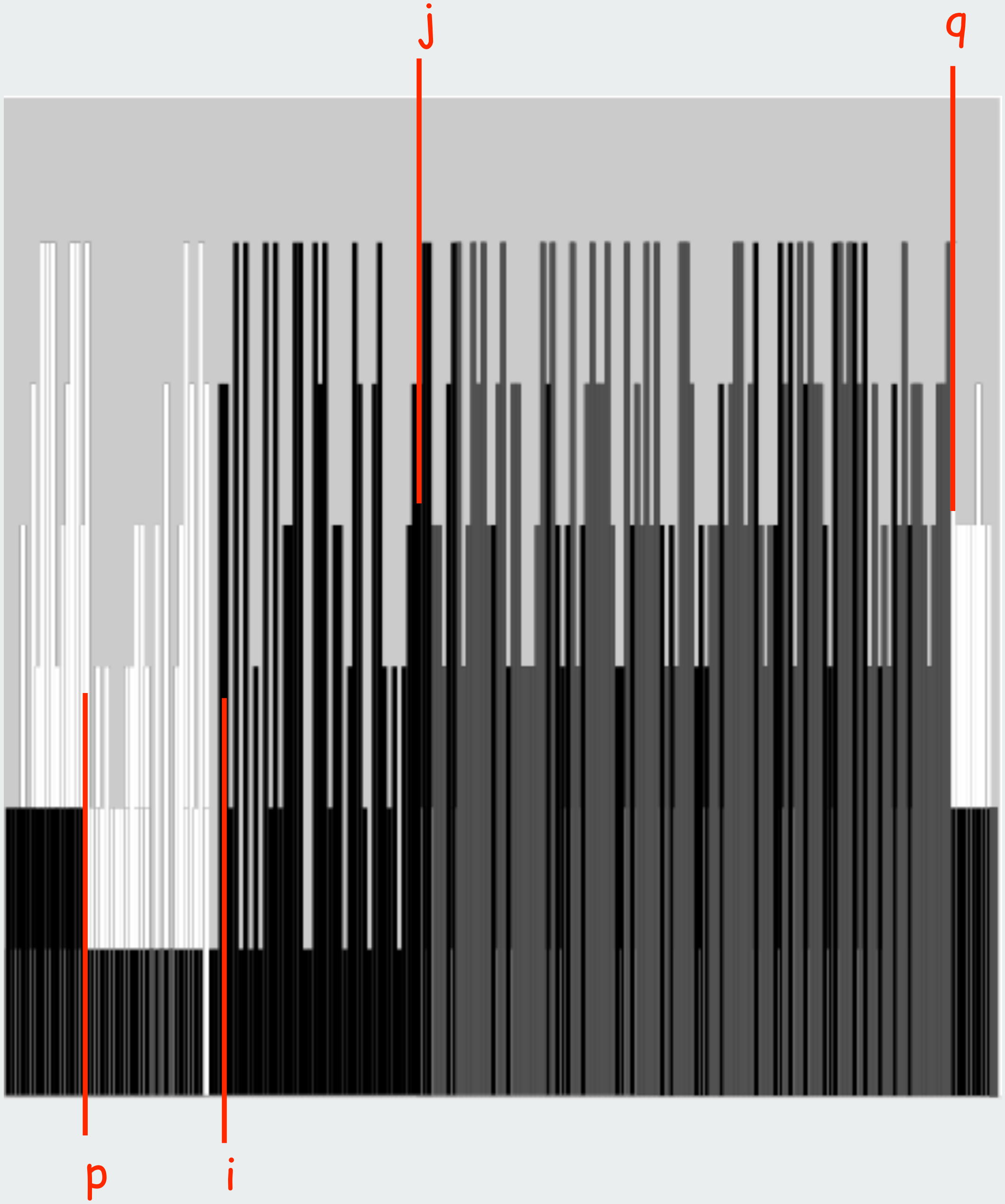
Theorem. [Sedgewick-Bentley] Quicksort with 3-way partitioning is optimal for random keys with duplicates.

Proof (beyond scope of 226).

- generalize decision tree
- tie cost to entropy
- note: cost is linear when number of key values is O(1)

Bottom line: Randomized Quicksort with 3-way partitioning reduces cost from linearithmic to linear (!) in broad class of applications

3-way partitioning animation



- complexity
- system sorts
- duplicate keys
- comparators

Generalized compare

Comparable interface: sort uses type's **compareTo()** function:

```
public class Date implements Comparable<Date>
{
   private int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }

   public int compareTo(Date b)
   {
      Date a = this;
      if (a.year  < b.year ) return -1;
      if (a.year  > b.year ) return +1;
      if (a.month < b.month) return -1;
      if (a.month > b.month) return +1;
      if (a.day   < b.day  ) return -1;
      if (a.day   > b.day  ) return +1;
      return 0;
   }
}
```

Generalized compare

Comparable interface: sort uses type's **compareTo()** function:

Problem 1: Not type-safe
Problem 2: May want to use a different order.
Problem 3: Some types may have no "natural" order.

Ex. Sort strings by:

• Natural order.	`Now is the time`
• Case insensitive.	`is Now the time`
• French.	`real réal rico`
• Spanish.	`café cuidado champiñón dulce`

ch and rr are single letters

```
String[] a;
...
Arrays.sort(a);
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
Arrays.sort(a, Collator.getInstance(Locale.FRENCH));
Arrays.sort(a, Collator.getInstance(Locale.SPANISH));
```

`import java.text.Collator;`

Generalized compare

Comparable interface: sort uses type's **compareTo()** function:

Problem 1: Not type-safe

Problem 2: May want to use a different order.

Problem 3: Some types may have no "natural" order.

A bad client

```
public class BadClient
  {
    public static void main(String[] args)
      {
        int N = Integer.parseInt(args[0]);
        Comparable[] a = new Comparable[N];
        ...
        a[i] = 1;
        ...
        a[j] = 2.0;
        ...
        Insertion.sort(a);
      }
}
```

autoboxed to **Integer** → `a[i] = 1;`

autoboxed to **Double** → `a[j] = 2.0;`

```
Exception ... java.lang.ClassCastException: java.lang.Double
        at java.lang.Integer.compareTo(Integer.java:35)
```

Generalized compare

Comparable interface: sort uses type's **compareTo()** function:

Problem 1: Not type-safe

Problem 2: May want to use a different order.

Problem 3: Some types may have no "natural" order.

Fix: generics

```
public class Insertion
{
   public static <Key extends Comparable<Key>>
                    void sort(Key[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0; j--)
            if (less(a[j], a[j-1])) exch(a, j, j-1);
            else                    break;
   }
}
```

Client can sort array of any **Comparable** type: **Double[], File[], Date[], ...**

Necessary in system library code; not in this course (for brevity)

Generalized compare

Comparable interface: sort uses type's **compareTo()** function:

Problem 1: Not type-safe

Problem 2: May want to use a different order.

Problem 3: Some types may have no "natural" order.

Solution: Use **Comparator** interface

Comparator interface. Require a method **compare()** so that **compare(v, w)** is a total order that behaves like **compareTo()**.

Advantage. Separates the definition of the data type from definition of what it means to compare two objects of that type.

- add any number of new orders to a data type.
- add an order to a library data type with no natural order.

Generalized compare

Comparable interface: sort uses type's **compareTo()** function:

Problem 2: May want to use a different order.
Problem 3: Some types may have no "natural" order.

Solution: Use **Comparator** interface

Example:

```
public class ReverseOrder implements Comparator<String>
{
   public int compare(String a, String b)
   {   return - a.compareTo(b);   }
}
```

reverse sense of comparison

```
   ...
   Arrays.sort(a, new ReverseOrder());
   ...
```

Generalized compare

Easy modification to support comparators in our sort implementations

- pass comparator to **sort()**, **less()**
- use it in **less()**

Example: (insertion sort)

```
public static void sort(Object[] a, Comparator comparator)
{
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i; j > 0; j--)
         if (less(comparator, a[j], a[j-1]))
              exch(a, j, j-1);
         else break;
}

private static boolean less(Comparator c, Object v, Object w)
{  return c.compare(v, w) < 0;   }

private static void exch(Object[] a, int i, int j)
{  Object t = a[i]; a[i] = a[j]; a[j] = t;  }
```

Generalized compare

Comparators enable multiple sorts of single file (different keys)

Example. Enable sorting students by name or by section.

```
Arrays.sort(students, Student.BY_NAME);
Arrays.sort(students, Student.BY_SECT);
```

sort by name ↓

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	2	A	991-878-4944	308 Blair
Fox	1	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	101 Brown
Gazsi	4	B	665-303-0266	22 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes

then sort by section ↓

Fox	1	A	884-232-5341	11 Dickinson
Chen	2	A	991-878-4944	308 Blair
Andrews	3	A	664-480-0023	097 Little
Furia	3	A	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	665-303-0266	22 Brown

Generalized compare

Comparators enable multiple sorts of single file (different keys)

Example. Enable sorting students by name or by section.

```
public class Student
{
   public static final Comparator<Student> BY_NAME = new ByName();
   public static final Comparator<Student> BY_SECT = new BySect();

   private String name;
   private int section;
   ...
   private static class ByName implements Comparator<Student>
   {
      public int compare(Student a, Student b)
      {  return a.name.compareTo(b.name);  }
   }

   private static class BySect implements Comparator<Student>
   {
      public int compare(Student a, Student b)
      {  return a.section - b.section;  }
   }
}
```

only use this trick if no danger of overflow

Generalized compare problem

A typical application
- first, sort by name
- then, sort by section

```
Arrays.sort(students, Student.BY_NAME);
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	2	A	991-878-4944	308 Blair
Fox	1	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	101 Brown
Gazsi	4	B	665-303-0266	22 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes

```
Arrays.sort(students, Student.BY_SECT);
```

Fox	1	A	884-232-5341	11 Dickinson
Chen	2	A	991-878-4944	308 Blair
Kanaga	3	B	898-122-9643	22 Brown
Andrews	3	A	664-480-0023	097 Little
Furia	3	A	766-093-9873	101 Brown
Rohde	3	A	232-343-5555	343 Forbes
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	665-303-0266	22 Brown

@#%&@!! Students in section 3 no longer in order by name.

A stable sort preserves the relative order of records with equal keys.

Is the system sort stable?

Stability

Q. Which sorts are stable?

- Selection sort?
- Insertion sort?
- Shellsort?
- Quicksort?
- Mergesort?

A. Careful look at code required.

Annoying fact. Many useful sorting algorithms are unstable.

Easy solutions.

- add an integer rank to the key
- careful implementation of mergesort

Open: Stable, inplace, optimal, practical sort??

Java system sorts

Use theory as a guide: Java uses both mergesort and quicksort.

- Can sort array of type `Comparable` or any primitive type.
- Uses quicksort for primitive types.
- Uses mergesort for objects.

```
import java.util.Arrays;
public class IntegerSort
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      int[] a = new int[N];
      for (int i = 0; i < N; i++)
         a[i] = StdIn.readInt();
      Arrays.sort(a);
      for (int i = 0; i < N; i++)
         System.out.println(a[i]);
   }
}
```

Q. Why use two different sorts?

A. Use of primitive types indicates time and space are critical

A. Use of objects indicates time and space not so critical

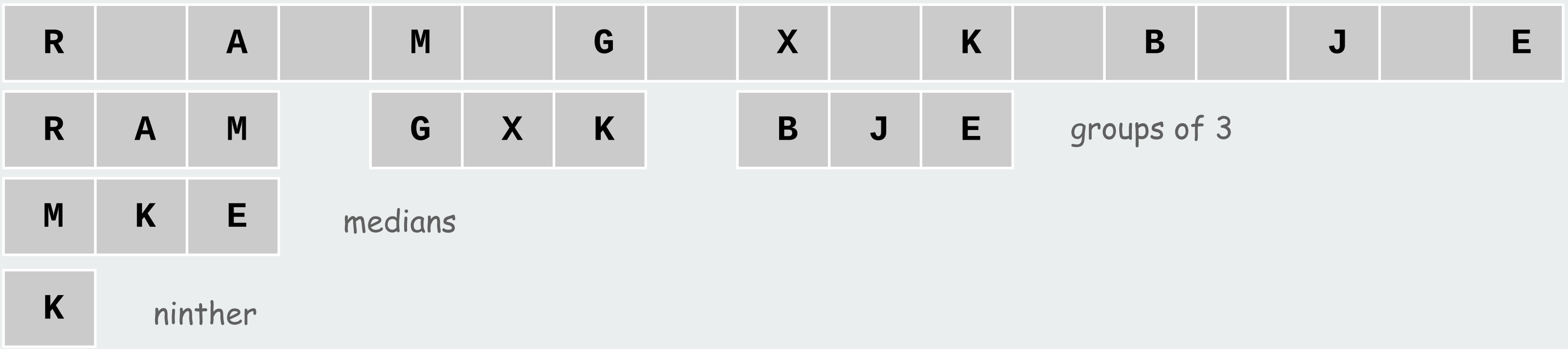
`Arrays.sort()` for primitive types

Bentley-McIlroy. [Engineeering a Sort Function]

- Original motivation: improve `qsort()` function in C.
- Basic algorithm = 3-way quicksort with cutoff to insertion sort.
- Partition on Tukey's ninther: median-of-3 elements, each of which is a median-of-3 elements.

approximate median-of-9

nine evenly spaced elements



R A M G X K B J E

R A M G X K B J E

groups of 3

M K E

medians

K

ninther

Why use ninther?

- better partitioning than sampling
- quick and easy to implement with macros
- less costly than random ← Good idea? Stay tuned.

Achilles heel in Bentley-McIlroy implementation (Java system sort)

Based on all this research, Java's system sort is solid, right?

McIlroy's devious idea. [A Killer Adversary for Quicksort]

- Construct malicious input while running system quicksort, in response to elements compared.
- If p is pivot, commit to `(x < p)` and `(y < p)`, but don't commit to `(x < y)` or `(x > y)` until x and y are compared.

Consequences.

- Confirms theoretical possibility.
- Algorithmic complexity attack: you enter linear amount of data; server performs quadratic amount of work.

Achilles heel in Bentley-McIlroy implementation (Java system sort)

A killer input:

more disastrous possibilities in C

- blows function call stack in Java and crashes program
- would take quadratic time if it didn't crash first

```
% more 250000.txt
0
218750
222662
11
166672
247070
83339
156253
...
```

```
% java IntegerSort < 250000.txt
Exception in thread "main" java.lang.StackOverflowError
   at java.util.Arrays.sort1(Arrays.java:562)
   at java.util.Arrays.sort1(Arrays.java:606)
   at java.util.Arrays.sort1(Arrays.java:608)
   at java.util.Arrays.sort1(Arrays.java:608)
   at java.util.Arrays.sort1(Arrays.java:608)
   . . .
```

250,000 integers between
0 and 250,000

Java's sorting library crashes, even if
you give it as much stack space as Windows allows.

Attack is not effective if file is randomly ordered before sort

System sort: Which algorithm to use?

Applications have diverse attributes

- Stable?
- Multiple keys?
- Deterministic?
- Keys all distinct?
- Multiple key types?
- Linked list or arrays?
- Large or small records?
- Is your file randomly ordered?
- Need guaranteed performance?

attributes

		1	2	3	4	.	.	.	.	M
algorithm	A	●			●					
	B			●		●			●	
	C		●		●					
	D						●			
	E			●						
	F		●			●		●		
	G	●								●
	.			●		●		●		
	.		●	●				●		
	.						●			●
	K	●				●				

many more combinations of attributes than algorithms

Elementary sort may be method of choice for some combination.
Cannot cover all combinations of attributes.

Q. Is the system sort good enough?
A. Maybe (no matter which algorithm it uses).

Priority Queues

- API
- elementary implementations
- binary heaps
- heapsort
- event-driven simulation

References:
Algorithms in Java, Chapter 9
http://www.cs.princeton.edu/introalgsds/34pq

- API
- elementary implementations
- binary heaps
- heapsort
- event-driven simulation

Priority Queues

Data. Items that can be compared.

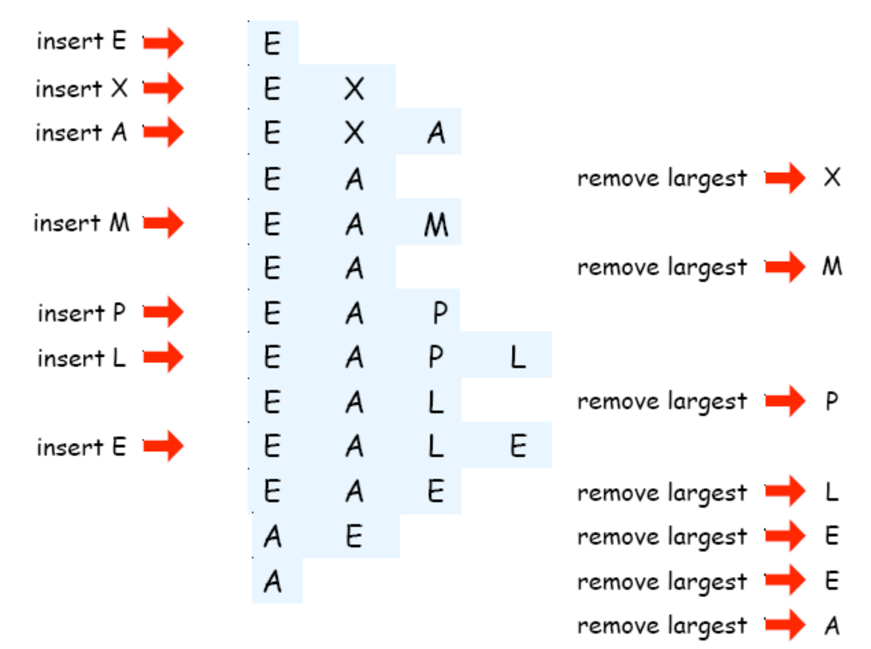
Basic operations.

- Insert.
- Remove largest.

defining ops

- Copy.
- Create.
- Destroy.
- Test if empty.

generic ops



insert E →	E				
insert X →	E	X			
insert A →	E	X	A		
	E	A			remove largest → X
insert M →	E	A	M		
	E	A			remove largest → M
insert P →	E	A	P		
insert L →	E	A	P	L	
	E	A	L		remove largest → P
insert E →	E	A	L	E	
	E	A	E		remove largest → L
	A	E			remove largest → E
	A				remove largest → E
					remove largest → A

Priority Queue Applications

Event-driven simulation.	[customers in a line, colliding particles]
Numerical computation.	[reducing roundoff error]
Data compression.	[Huffman codes]
Graph searching.	[Dijkstra's algorithm, Prim's algorithm]
Computational number theory.	[sum of powers]
Artificial intelligence.	[A* search]
Statistics.	[maintain largest M values in a sequence]
Operating systems.	[load balancing, interrupt handling]
Discrete optimization.	[bin packing, scheduling]
Spam filtering.	[Bayesian spam filter]

Generalizes: stack, queue, randomized queue.

Priority queue client example

Problem: Find the largest M of a stream of N elements.

- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

Constraint. Not enough memory to store N elements.

Solution. Use a priority queue.

Operation	time	space
sort	N lg N	N
elementary PQ	M N	M
binary heap	N lg M	M
best in theory	N	M

```
MinPQ<Transaction> pq
             = new MinPQ<Transaction>();

while(!StdIn.isEmpty())
{
   String s = StdIn.readLine();
   t = new Transaction(s);
   pq.insert(t);
   if (pq.size() > M)
      pq.delMin();
}


while (!pq.isEmpty())
   System.out.println(pq.delMin());
```

- API
- **elementary implementations**
- binary heaps
- heapsort
- event-driven simulation

Priority queue: unordered array implementation

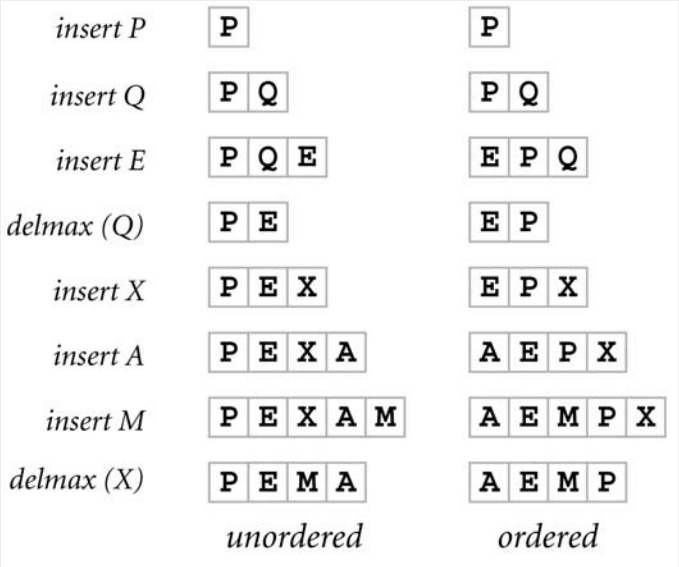
```
public class UnorderedPQ<Item extends Comparable>
{
   private Item[] pq;  // pq[i] = ith element on PQ
   private int N;      // number of elements on PQ

   public UnorderedPQ(int maxN)
   {  pq = (Item[]) new Comparable[maxN];  }

   public boolean isEmpty()
   {  return N == 0; }

   public void insert(Item x)
   {  pq[N++] = x;  }

   public Item delMax()
   {
      int max = 0;
      for (int i = 1; i < N; i++)
         if (less(max, i)) max = i;
      exch(max, N-1);
      return pq[--N];
   }
}
```

no generic array creation

Priority queue elementary implementations

Implementation	Insert	Del Max
unordered array	1	N
ordered array	N	1

worst-case asymptotic costs for PQ with N items



	unordered	ordered
insert P	P	P
insert Q	P Q	P Q
insert E	P Q E	E P Q
delmax (Q)	P E	E P
insert X	P E X	E P X
insert A	P E X A	A E P X
insert M	P E X A M	A E M P X
delmax (X)	P E M A	A E M P

Challenge. Implement both operations efficiently.

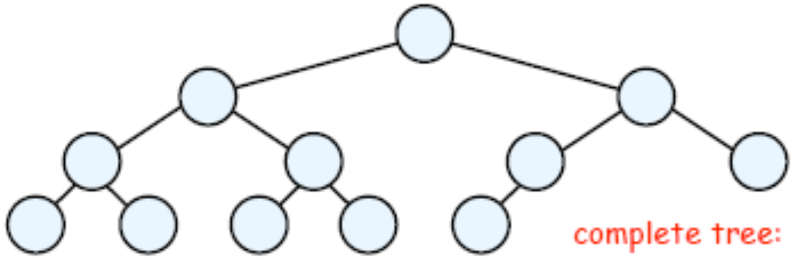
- API
- elementary implementations
- **binary heaps**
- heapsort
- event-driven simulation

Binary Heap

Heap: Array representation of a heap-ordered complete binary tree.

Binary tree.

- Empty or
- Node with links to left and right trees.



complete tree: balanced except for bottom level

Binary Heap

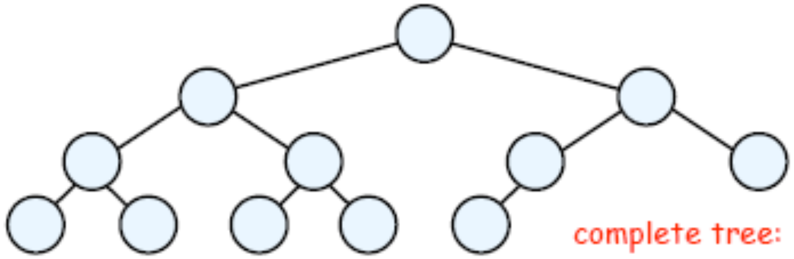
Heap: Array representation of a heap-ordered complete binary tree.

Binary tree.

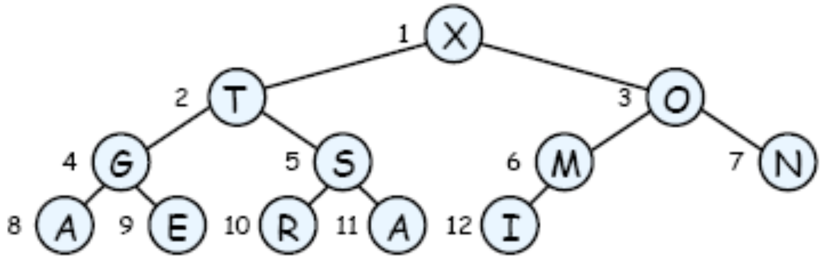
- Empty or
- Node with links to left and right trees.

Heap-ordered binary tree.

- Keys in nodes.
- No smaller than children's keys.



complete tree: balanced except for bottom level



Binary Heap

Heap: Array representation of a heap-ordered complete binary tree.

Binary tree.

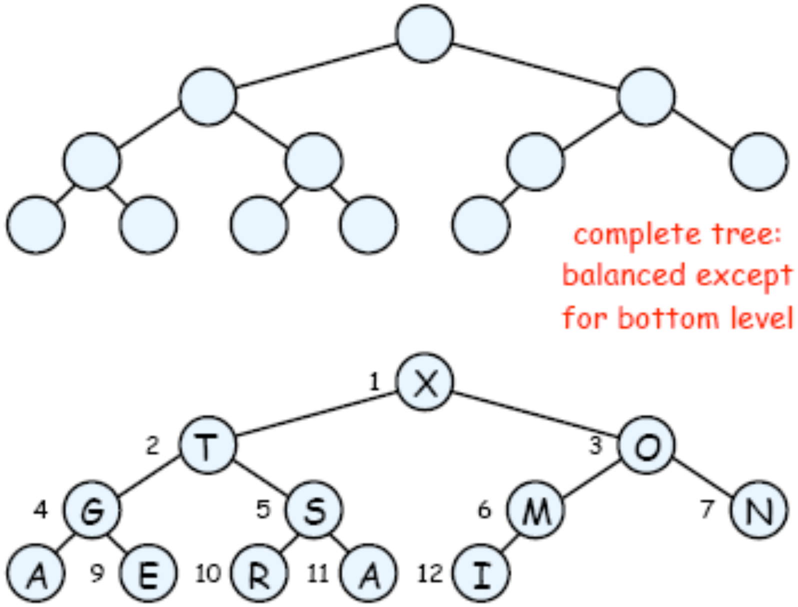
- Empty or
- Node with links to left and right trees.

Heap-ordered binary tree.

- Keys in nodes.
- No smaller than children's keys.

Array representation.

- Take nodes in level order.
- No explicit links needed since tree is complete.

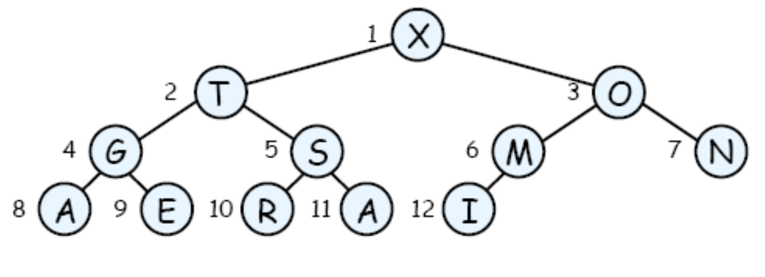


complete tree: balanced except for bottom level

1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

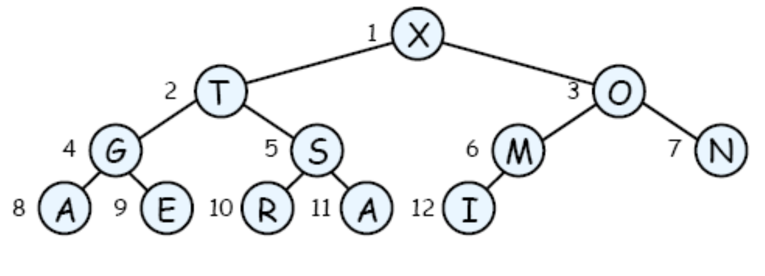
Binary Heap Properties

Property A. Largest key is at root.



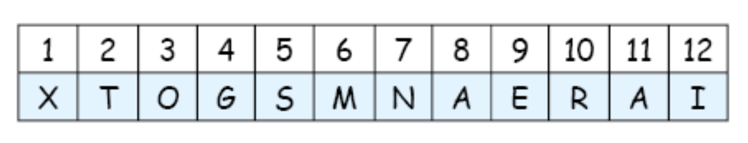
Binary Heap Properties

Property A. Largest key is at root.



Property B. Can use array indices to move through tree.

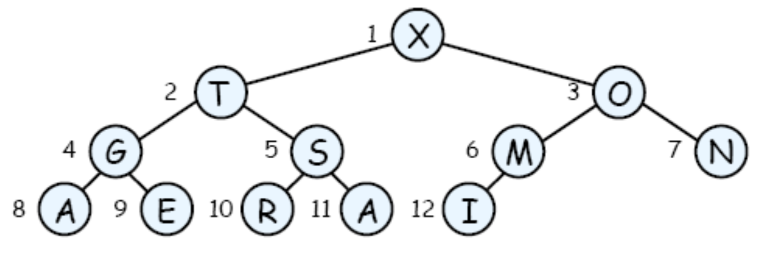
- Note: indices start at 1.
- Parent of node at k is at k/2.
- Children of node at k are at 2k and 2k+1.



1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

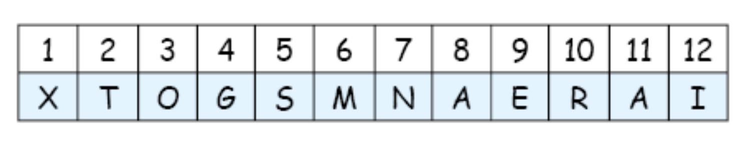
Binary Heap Properties

Property A. Largest key is at root.



Property B. Can use array indices to move through tree.

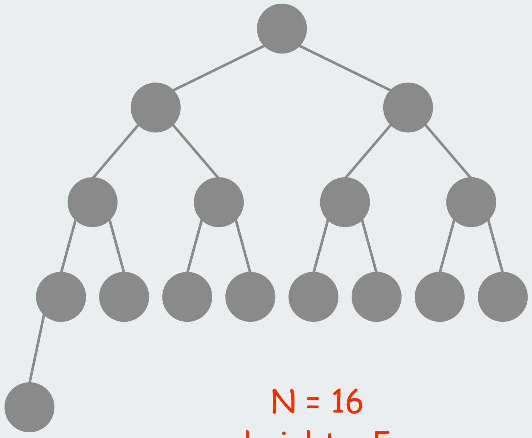
- Note: indices start at 1.
- Parent of node at k is at k/2.
- Children of node at k are at 2k and 2k+1.



1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

Property C. Height of N node heap is $1 + \lfloor \lg N \rfloor$.

height increases only when
N is a power of 2



N = 16
height = 5

Promotion In a Heap

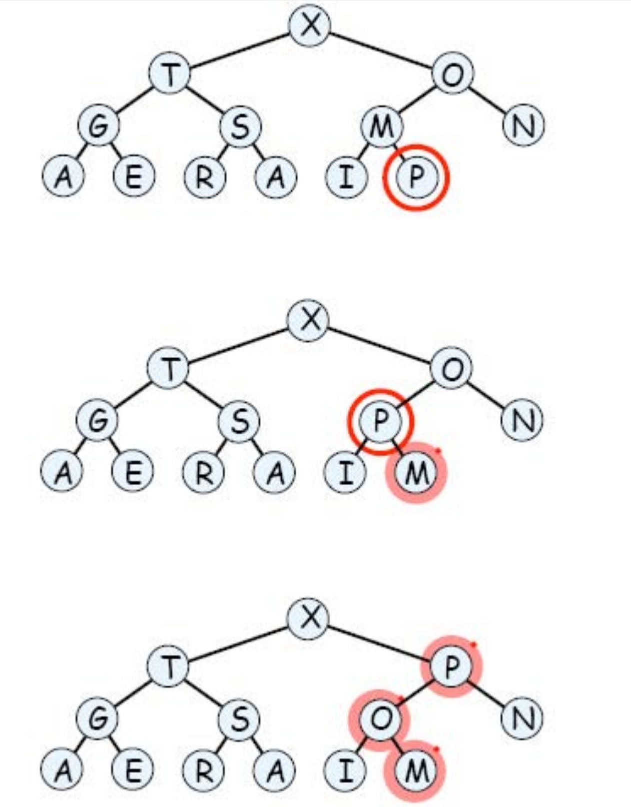
Scenario. Exactly one node has a larger key than its parent.

To eliminate the violation:

- Exchange with its parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
   while (k > 1 && less(k/2, k))
   {
      exch(k, k/2);
      k = k/2;
   }
}
```

parent of node at k is at k/2



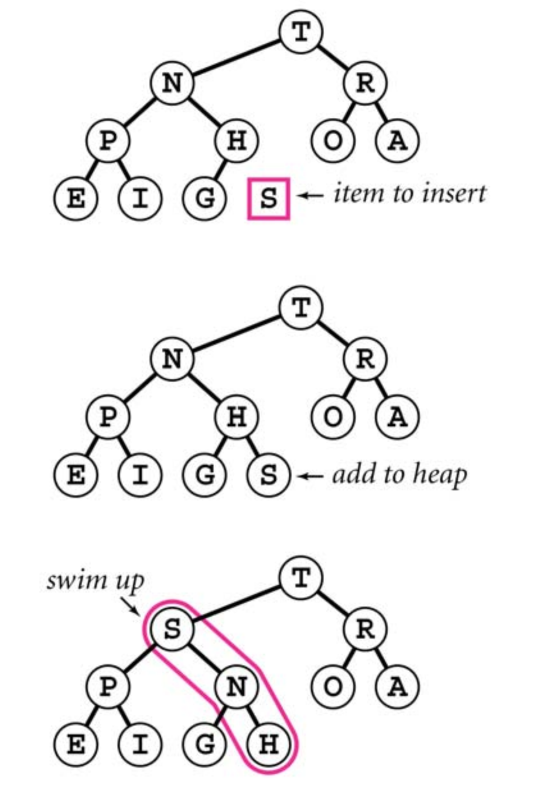
1	2	3	4	5	6	7	8	9	10	11	12	13
X	T	O	G	S	M	N	A	E	R	A	I	P
X	T	P	G	S	O	N	A	E	R	A	I	M

Peter principle: node promoted to level of incompetence.

Insert

Insert. Add node at end, then promote.

```
public void insert(Item x)
{
   pq[++N] = x;
   swim(N);
}
```



Demotion In a Heap

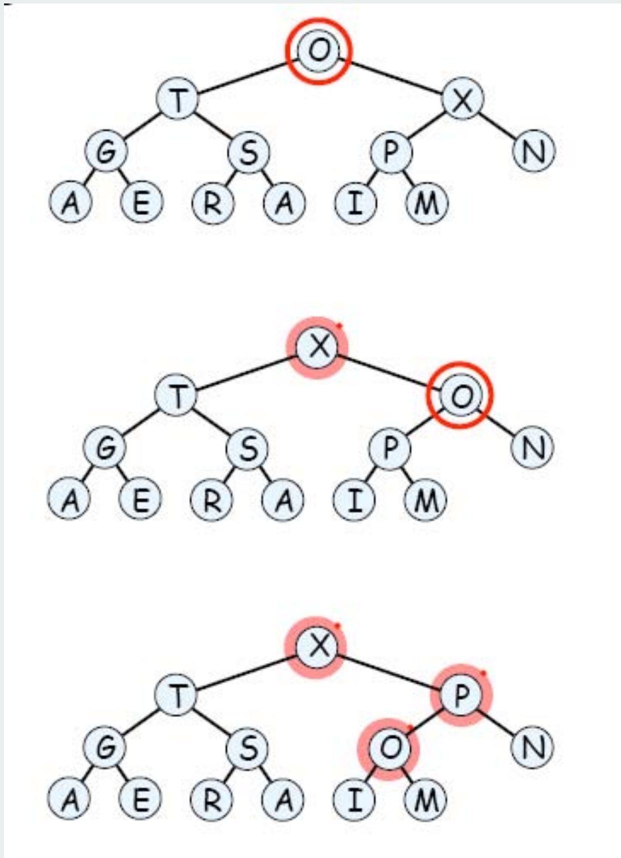
Scenario. Exactly one node has a smaller key than does a child.

To eliminate the violation:

- Exchange with larger child.
- Repeat until heap order restored.

```
private void sink(int k)
{
   while (2*k <= N)
   {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

children of node at k are 2k and 2k+1



1	2	3	4	5	6	7	8	9	10	11	12	13
O	T	X	G	S	P	N	A	E	R	A	I	M
X	T	P	G	S	O	N	A	E	R	A	I	M

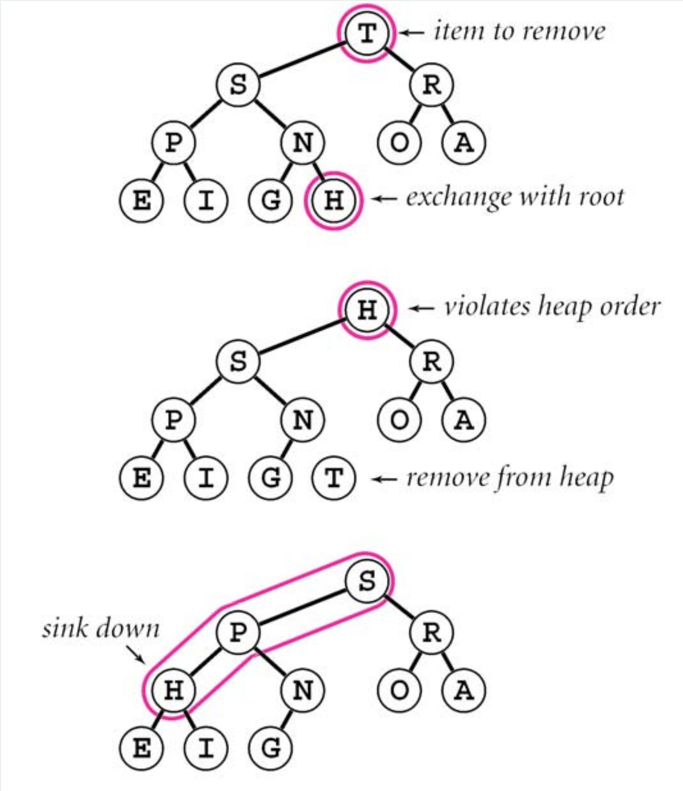
Power struggle: better subordinate promoted.

Remove the Maximum

Remove max. Exchange root with node at end, then demote.

```
public Item delMax()
{
   Item max = pq[1];
   exch(1, N--);
   sink(1);
   pq[N+1] = null;
   return max;
}
```

prevent loitering



Binary heap implementation summary

```
public class MaxPQ<Item extends Comparable>
{
   private Item[] pq;
   private int N;

   public MaxPQ(int maxN)
   {   . . .   }
   public boolean isEmpty()
   {   . . .   }

   public void insert(Item x)
   {   . . .   }
   public Item delMax()
   {   . . .   }

   private void swim(int k)
   {   . . .   }
   private void sink(int k)
   {   . . .   }

   private boolean less(int i, int j)
   {   . . .   }
   private void    exch(int i, int j)
   {   . . .   }
}
```

same as array-based PQ,
but allocate one extra element

PQ ops

heap helper functions

array helper functions

Binary heap considerations

Minimum oriented priority queue

- replace **`less()`** with **`greater()`**
- implement **`greater()`**.

Array resizing

- add no-arg constructor
- apply repeated doubling. ← leads to O(log N) amortized time per op

Immutability of keys.

- assumption: client does not change keys while they're on the PQ
- best practice: use immutable keys

Other operations.

- remove an arbitrary item.
- change the priority of an item.

easy to implement with **`sink()`** and **`swim()`** [stay tuned]

Priority Queues Implementation Cost Summary

Operation	Insert	Remove Max	Find Max
ordered array	N	1	1
ordered list	N	1	1
unordered array	1	N	N
unordered list	1	N	N
binary heap	lg N	lg N	1

worst-case asymptotic costs for PQ with N items

Hopeless challenge. Make all ops O(1).

Why hopeless?

- API
- elementary implementations
- binary heaps
- **heapsort**
- event-driven simulation

Digression: Heapsort

First pass: build heap.

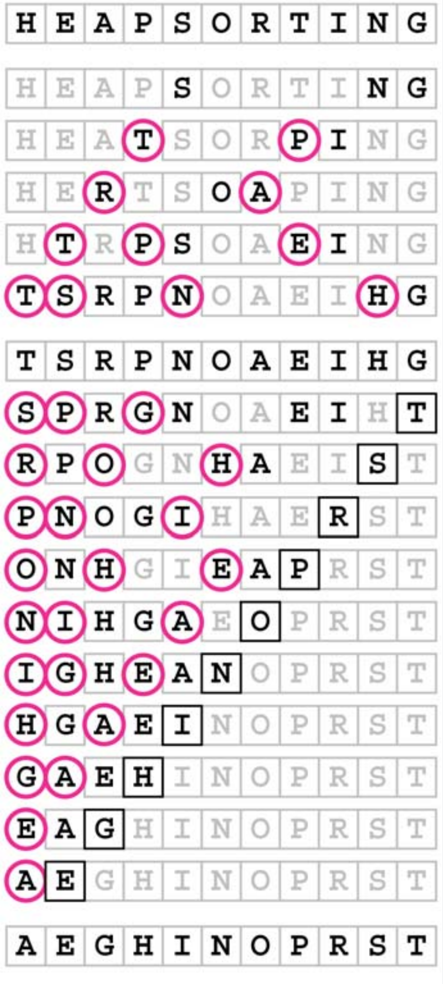
- Insert items into heap, one at at time.
- Or can use faster bottom-up method; see book.

```
for (int k = N/2; k >= 1; k--)
   sink(a, k, N);
```

Second pass: sort.

- Remove maximum items, one at a time.
- Leave in array, instead of nulling out.

```
while (N > 1
{
   exch(a, 1, N--);
   sink(a, 1, N);
}
```



```
H E A P S O R T I N G

H E A P S O R T I N G
H E A T S O R P I N G
H E R T S O A P I N G
H T R P S O A E I N G
T S R P N O A E I H G

T S R P N O A E I H G
S P R G N O A E I H T
R P O G N H A E I S T
P N O G I H A E R S T
O N H G I E A P R S T
N I H G A E O P R S T
I G H E A N O P R S T
H G A E I N O P R S T
G A E H I N O P R S T
E A G H I N O P R S T
A E G H I N O P R S T

A E G H I N O P R S T
```

Property D. At most 2 N lg N comparisons.

Significance of Heapsort

Q. Sort in O(N log N) worst-case without using extra memory?

A. Yes. Heapsort.

Not mergesort? Linear extra space. ← in-place merge possible, not practical

Not quicksort? Quadratic time in worst case. ← O(N log N) worst-case quicksort possible, not practical.

Heapsort is optimal for both time and space, but:

- inner loop longer than quicksort's.
- makes poor use of cache memory.

Sorting algorithms: summary

	inplace	stable	worst	average	best	remarks
selection	x		N^2 / 2	N^2 / 2	N^2 / 2	N exchanges
insertion	x	x	N^2 / 2	N^2 / 4	N	use for small N or partly ordered
shell	x				N	tight code
quick	x		N^2 / 2	2N ln N	N lg N	N log N probabilistic guarantee fastest in practice
merge		x	N lg N	N lg N	N lg N	N log N guarantee, stable
heap	x		2N lg N	2N lg N	N lg N	N log N guarantee, in-place

- API
- elementary implementations
- binary heaps
- heapsort
- **event-driven simulation**

Review

Bouncing balls (COS 126)

```
public class BouncingBalls
{
    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        Ball balls[] = new Ball[N];
        for (int i = 0; i < N; i++)
            balls[i] = new Ball();
        while(true)
        {
            StdDraw.clear();
            for (int i = 0; i < N; i++)
            {
                balls[i].move();
                balls[i].draw();
            }
            StdDraw.show(50);
        }
    }
}
```

Review

Bouncing balls (COS 126)

```
public class Ball
{
    private double rx, ry;   // position
    private double vx, vy;   // velocity
    private double radius;   // radius
    public Ball()
    { ... initialize position and velocity ... }
    public void move()
    {
        if ((rx + vx < radius) || (rx + vx > 1.0 - radius)) { vx = -vx; }
        if ((ry + vy < radius) || (ry + vy > 1.0 - radius)) { vy = -vy; }
        rx = rx + vx;
        ry = ry + vy;
    }
    public void draw()
    {  StdDraw.filledCircle(rx, ry, radius);  }
}
```

checks for colliding with walls

Missing: check for balls colliding with each other

- physics problems: when? what effect?
- CS problems: what object does the checks? too many checks?

Molecular dynamics simulation of hard spheres

Goal. Simulate the motion of N moving particles that behave according to the laws of elastic collision.

Hard sphere model.

- Moving particles interact via elastic collisions with each other, and with fixed walls.
- Each particle is a sphere with known position, velocity, mass, and radius.
- No other forces are exerted.

temperature, pressure, diffusion constant

motion of individual atoms and molecules

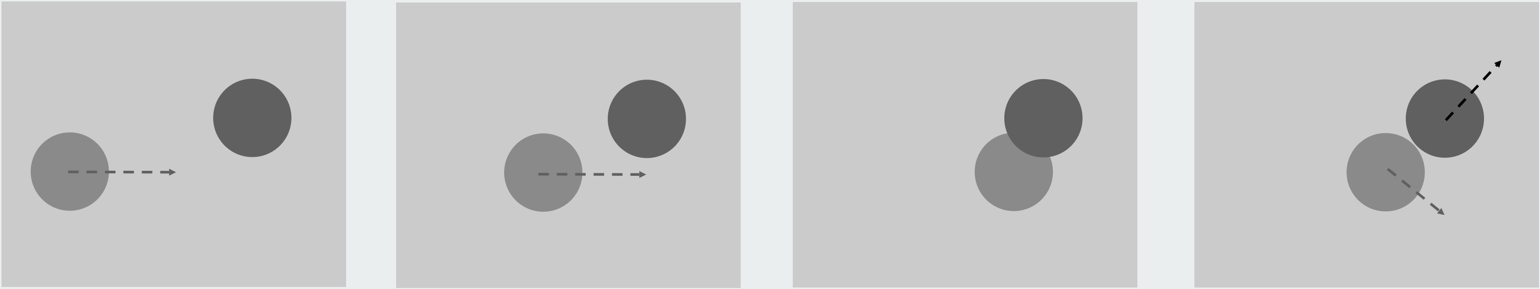
Significance. Relates macroscopic observables to microscopic dynamics.

- Maxwell and Boltzmann: derive distribution of speeds of interacting molecules as a function of temperature.
- Einstein: explain Brownian motion of pollen grains.

Time-driven simulation

Time-driven simulation.

- Discretize time in quanta of size dt.
- Update the position of each particle after every dt units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.



t

t + dt

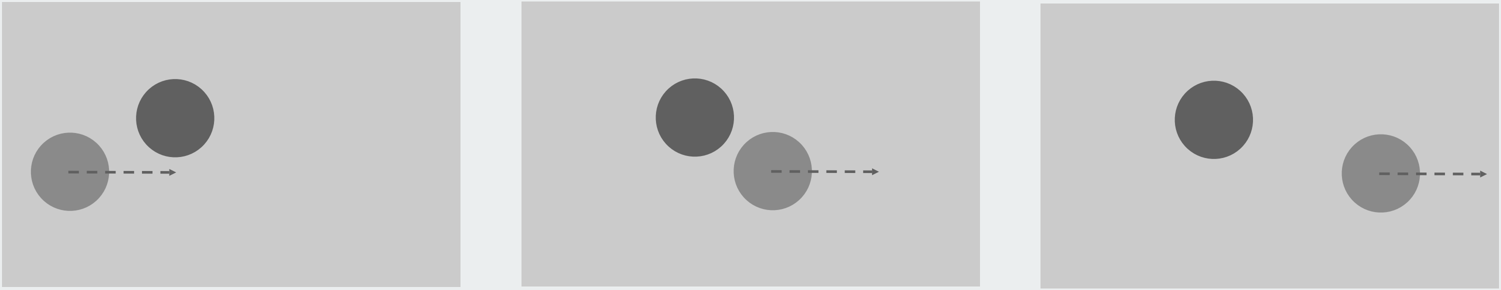
t + 2 dt
(collision detected)

t + Δt
(roll back clock)

Time-driven simulation

Main drawbacks.

- N^2 overlap checks per time quantum.
- May miss collisions if dt is too large and colliding particles fail to overlap when we are looking.
- Simulation is too slow if dt is very small.



t

t + dt

t + 2 dt

Event-driven simulation

Change state only when something happens.

- Between collisions, particles move in straight-line trajectories.
- Focus only on times when collisions occur.
- Maintain priority queue of collision events, prioritized by time.
- Remove the minimum = get next collision.

Collision prediction. Given position, velocity, and radius of a particle, when will it collide next with a wall or another particle?

Collision resolution. If collision occurs, update colliding particle(s) according to laws of elastic collisions.

Note: Same approach works for a broad variety of systems

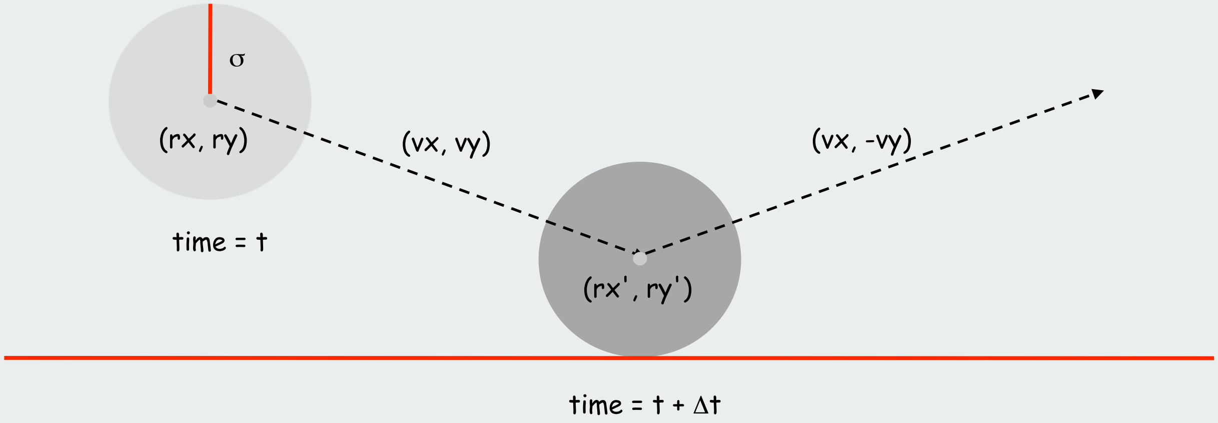
Particle-wall collision

Collision prediction.

- Particle of radius σ at position (rx, ry).
- Particle moving in unit box with velocity (vx, vy).
- Will it collide with a horizontal wall? If so, when?

$$\Delta t \;=\; \begin{cases} \infty & \text{if } vy = 0 \\ (\sigma - ry)/vy & \text{if } vy < 0 \\ (1 - \sigma - ry)/vy & \text{if } vy > 0 \end{cases}$$

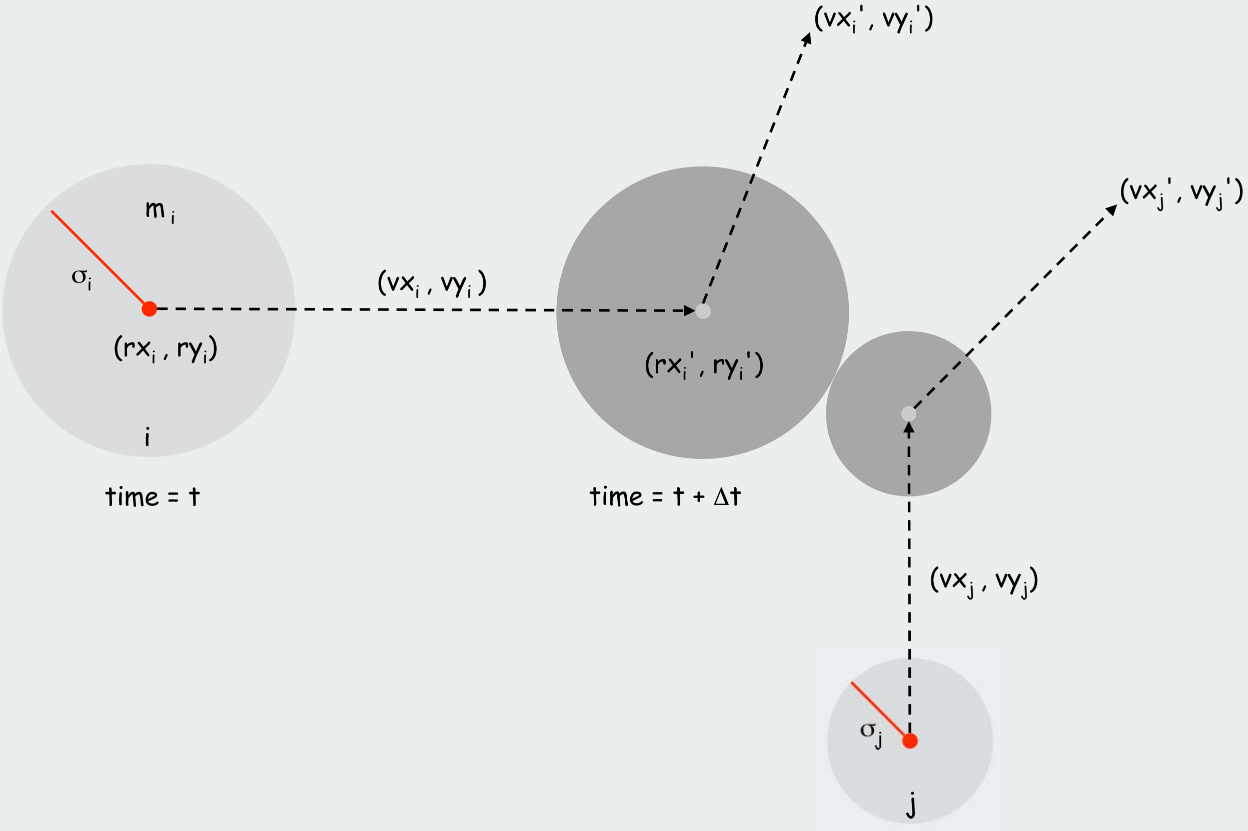
Collision resolution. (vx', vy') = (vx, -vy).



Particle-particle collision prediction

Collision prediction.

- Particle i: radius σ_i, position (rx_i, ry_i), velocity (vx_i, vy_i).
- Particle j: radius σ_j, position (rx_j, ry_j), velocity (vx_j, vy_j).
- Will particles i and j collide? If so, when?



Particle-particle collision prediction

Collision prediction.

- Particle i: radius σ_i, position (rx_i, ry_i), velocity (vx_i, vy_i).
- Particle j: radius σ_j, position (rx_j, ry_j), velocity (vx_j, vy_j).
- Will particles i and j collide? If so, when?

$$\Delta t = \begin{cases} \infty & \text{if } \Delta v \cdot \Delta r \geq 0 \\ \infty & \text{if } d < 0 \\ -\dfrac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise} \end{cases}$$

$$d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v)\ (\Delta r \cdot \Delta r - \sigma^2) \qquad \sigma = \sigma_i + \sigma_j$$

$$\Delta v = (\Delta vx,\ \Delta vy) = (vx_i - vx_j,\ vy_i - vy_j)$$
$$\Delta r = (\Delta rx,\ \Delta ry) = (rx_i - rx_j,\ ry_i - ry_j)$$

$$\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2$$
$$\Delta r \cdot \Delta r = (\Delta rx)^2 + (\Delta ry)^2$$
$$\Delta v \cdot \Delta r = (\Delta vx)(\Delta rx) + (\Delta vy)(\Delta ry)$$

Particle-particle collision prediction implementation

Particle has method to predict collision with another particle

```
public double dt(Particle b)
{
    Particle a = this;
    if (a == b) return INFINITY;
    double dx  = b.rx - a.rx;
    double dy  = b.ry - a.ry;
    double dvx = b.vx - a.vx;
    double dvy = b.vy - a.vy;
    double dvdr = dx*dvx + dy*dvy;
    if(dvdr > 0) return INFINITY;
    double dvdv = dvx*dvx + dvy*dvy;
    double drdr = dx*dx + dy*dy;
    double sigma = a.radius + b.radius;
    double d = (dvdr*dvdr) - dvdv * (drdr - sigma*sigma);
    if (d < 0) return INFINITY;
    return -(dvdr + Math.sqrt(d)) / dvdv;
}
```

and methods **`dtX()`** and **`dtY()`** to predict collisions with walls

Particle-particle collision prediction implementation

CollisionSystem has method to predict all collisions

```
private void predict(Particle a, double limit)
{
    if (a == null) return;
    for(int i = 0; i < N; i++)
    {
        double dt = a.dt(particles[i]);
        if(t + dt <= limit)
            pq.insert(new Event(t + dt, a, particles[i]));
    }
    double dtX = a.dtX();
    double dtY = a.dtY();
    if (t + dtX <= limit)
       pq.insert(new Event(t + dtX, a, null));
    if (t + dtY <= limit)
       pq.insert(new Event(t + dtY, null, a));
 }
```

Particle-particle collision resolution

Collision resolution. When two particles collide, how does velocity change?

$$
\begin{aligned}
vx_i' &= vx_i + Jx \,/\, m_i \\
vy_i' &= vy_i + Jy \,/\, m_i \\
vx_j' &= vx_j - Jx \,/\, m_j \\
vy_j' &= vx_j - Jy \,/\, m_j
\end{aligned}
$$

Newton's second law
(momentum form)

$$Jx = \frac{J\,\Delta rx}{\sigma}, \quad Jy = \frac{J\,\Delta ry}{\sigma}, \quad J = \frac{2\,m_i\,m_j\,(\Delta v \cdot \Delta r)}{\sigma(m_i + m_j)}$$

impulse due to normal force
(conservation of energy, conservation of momentum)

Particle-particle collision resolution implementation

Particle has method to resolve collision with another particle

```
public void bounce(Particle b)
{
     Particle a = this;
     double dx  = b.rx - a.rx;
     double dy  = b.ry - a.ry;
     double dvx = b.vx - a.vx;
     double dvy = b.vy - a.vy;
     double dvdr = dx*dvx + dy*dvy;
     double dist = a.radius + b.radius;
     double J = 2 * a.mass * b.mass * dvdr / ((a.mass + b.mass) * dist);
     double Jx = J * dx / dist;
     double Jy = J * dy / dist;
     a.vx += Jx / a.mass;
     a.vy += Jy / a.mass;
     b.vx -= Jx / b.mass;
     b.vy -= Jy / b.mass;
     a.count++;
     b.count++;
 }
```

and methods **bounceX()** and **bounceY()** to resolve collisions with walls

Collision system: event-driven simulation main loop

Initialization.

- Fill PQ with all potential particle-wall collisions
- Fill PQ with all potential particle-particle collisions.

"potential" since collision may not happen if
some other collision intervenes

Main loop.

- Delete the impending event from PQ (min priority = t).
- If the event in no longer valid, ignore it.
- Advance all particles to time t, on a straight-line trajectory.
- Update the velocities of the colliding particle(s).
- Predict future particle-wall and particle-particle collisions involving the colliding particle(s) and insert events onto PQ.

Collision system: main event-driven simulation loop implementation

```
public void simulate(double limit)
{
    pq = new MinPQ<Event>();
    for(int i = 0; i < N; i++)
        predict(particles[i], limit);
    pq.insert(new Event(0, null, null));
    while(!pq.isEmpty())
    {
        Event e = pq.delMin();
        if(!e.isValid()) continue;
        Particle a = e.a();
        Particle b = e.b();

        for(int i = 0; i < N; i++)
            particles[i].move(e.time() - t);
        t = e.time();

        if      (a != null && b != null) a.bounce(b);
        else if (a != null && b == null) a.bounceX()
        else if (a == null && b != null) b.bounceY();
        else if (a == null && b == null)
        {
            StdDraw.clear(StdDraw.WHITE);
            for(int i = 0; i < N; i++) particles[i].draw();
            StdDraw.show(20);
            if (t < limit)
              pq.insert(new Event(t + 1.0 / Hz, null, null));
        }
        predict(a, limit);
        predict(b, limit);
     }
 }
```

initialize PQ with collision events and redraw event

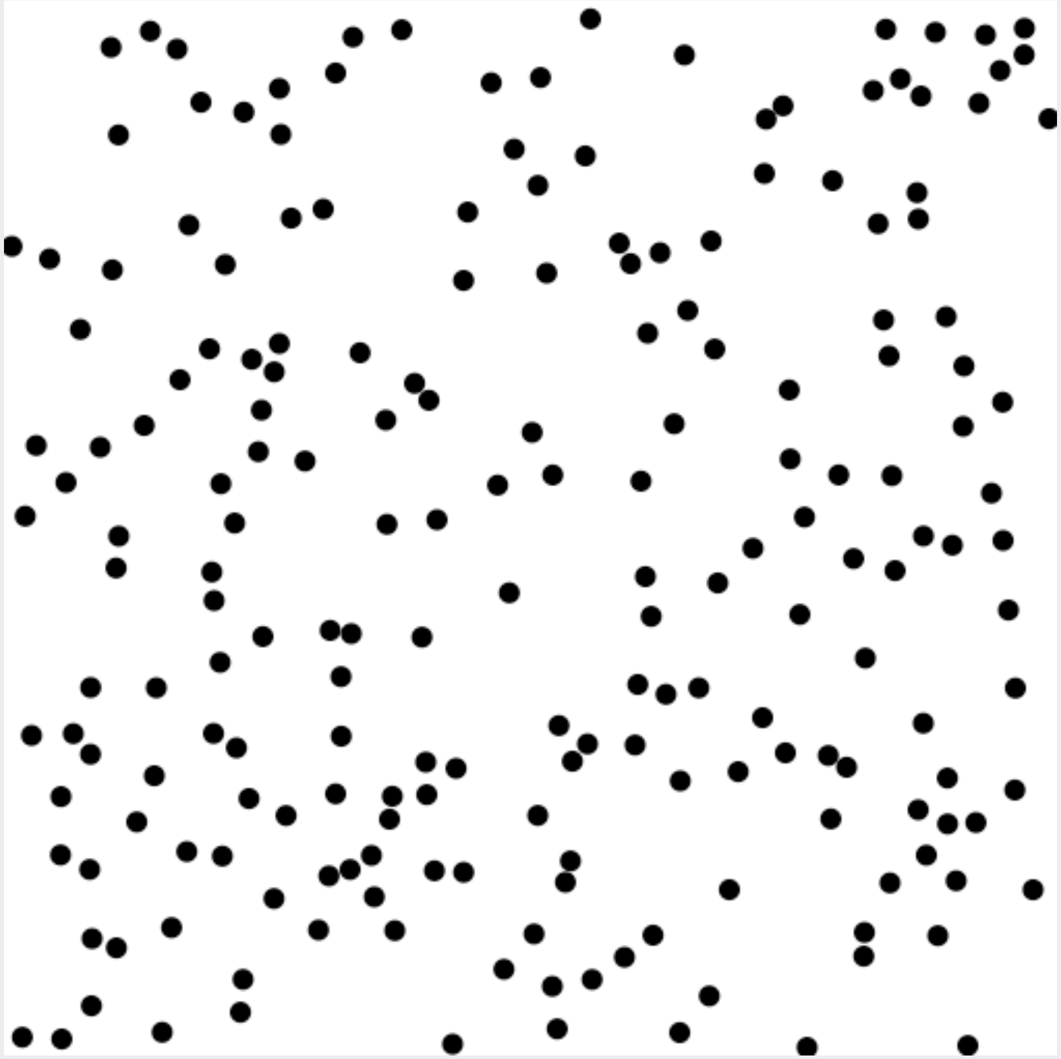
main event-driven simulation loop

update positions and time

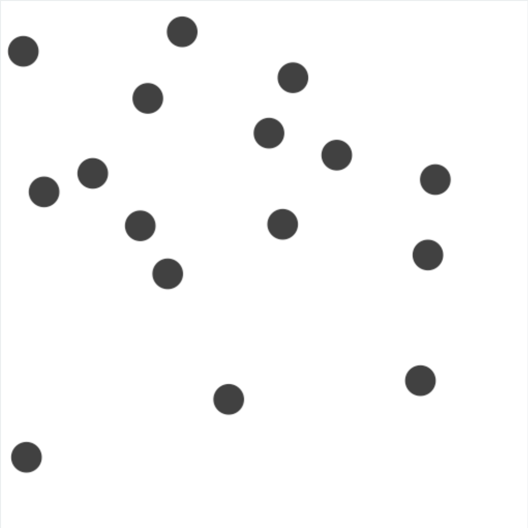
process event

predict new events based on changes

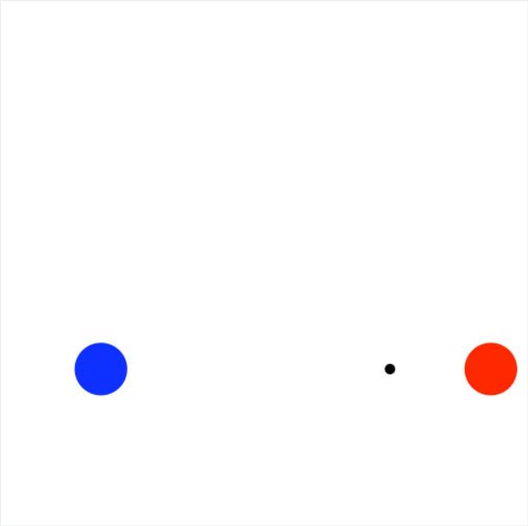
```
java CollisionSystem 200
```



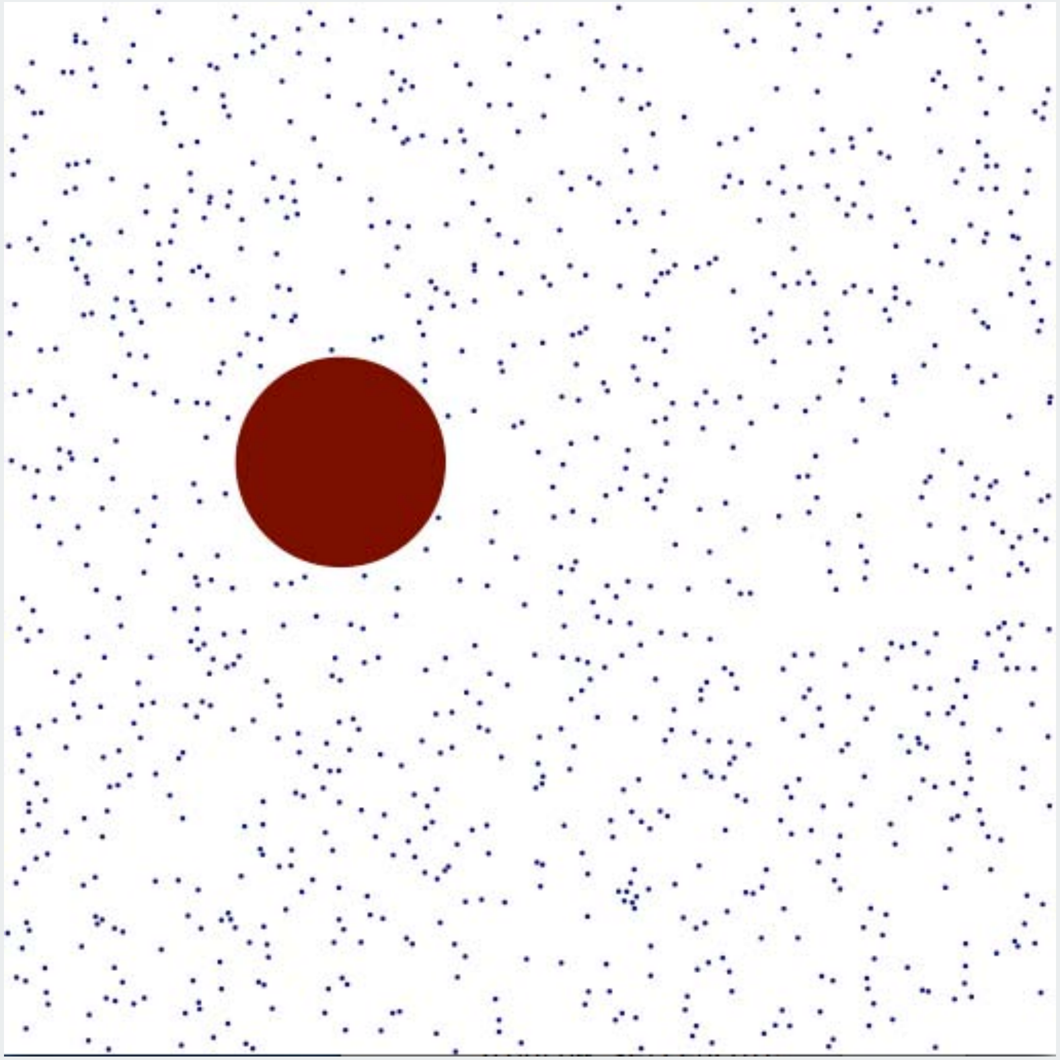
```
java CollisionSystem < billiards5.txt
```



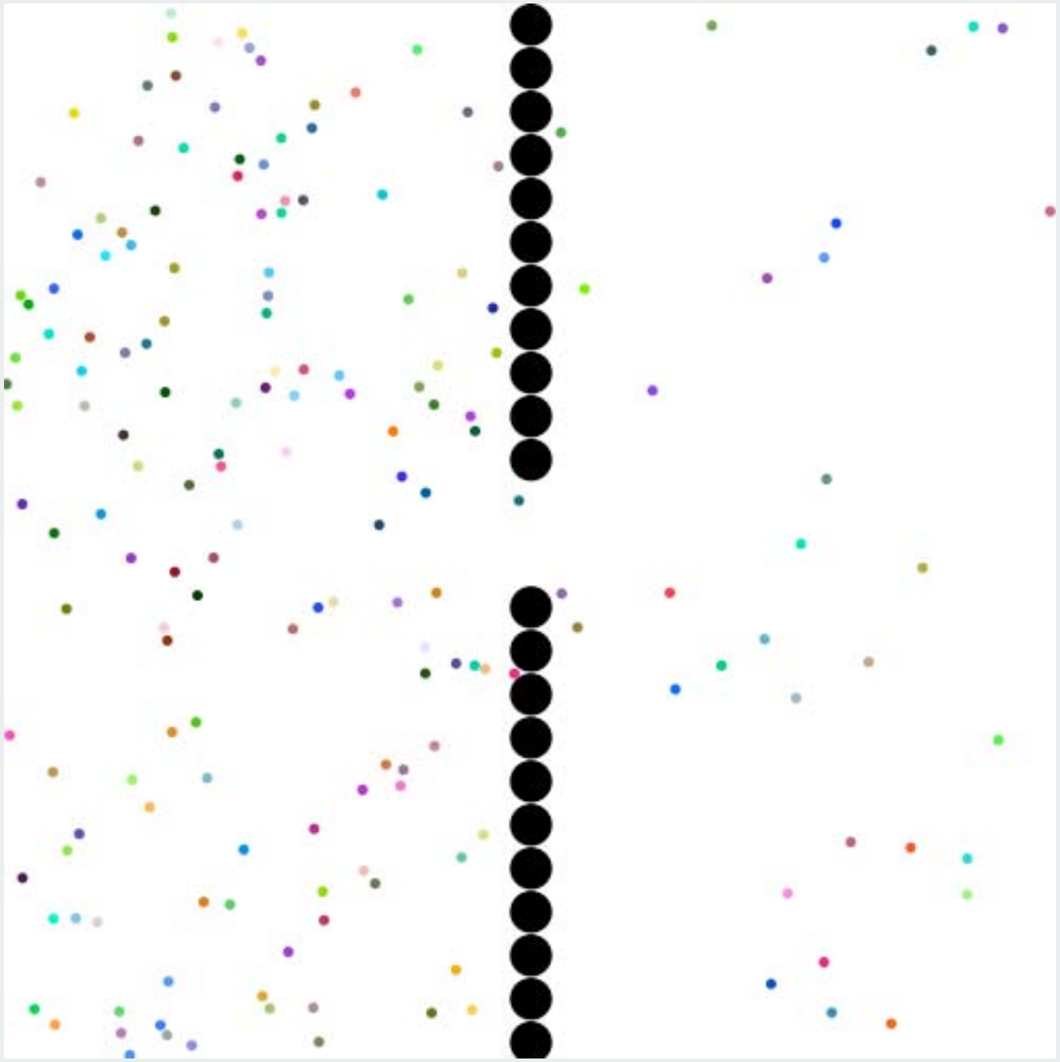
```
java CollisionSystem < squeeze2.txt
```



```
java CollisionSystem < brownianmotion.txt
```



```
java CollisionSystem < diffusion.txt
```



Symbol Tables

- API
- basic implementations
- iterators
- Comparable keys
- challenges

References:

Algorithms in Java, Chapter 12
Intro to Programming, Section 4.4
http://www.cs.princeton.edu/introalgsds/41st

- API
- basic implementations
- iterators
- Comparable keys
- challenges

Symbol Tables

Key-value pair abstraction.

- Insert a value with specified key.
- Given a key, search for the corresponding value.

Example: DNS lookup.

- Insert URL with specified IP address.
- Given URL, find corresponding IP address

URL	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

key value

Can interchange roles: given IP address find corresponding URL

Symbol Table Applications

Application	Purpose	Key	Value
Phone book	Look up phone number	Name	Phone number
Bank	Process transaction	Account number	Transaction details
File share	Find song to download	Name of song	Computer ID
File system	Find file on disk	Filename	Location on disk
Dictionary	Look up word	Word	Definition
Web search	Find relevant documents	Keyword	List of documents
Book index	Find relevant pages	Keyword	List of pages
Web cache	Download	Filename	File contents
Genomics	Find markers	DNA string	Known positions
DNS	Find IP address given URL	URL	IP address
Reverse DNS	Find URL given IP address	IP address	URL
Compiler	Find properties of variable	Variable name	Value and type
Routing table	Route Internet packets	Destination	Best route

Symbol Table API

Associative array abstraction: Unique value associated with each key.

`public class *ST<Key extends Comparable<Key>, Value>`

	`*ST()`	create a symbol table	
`void`	`put(Key key, Value val)`	put key-value pair into the table	← insert
`Value`	`get(Key key)`	return value paired with **key** (null if **key** not in table)	← search
`boolean`	`contains(Key key)`	is there a value paired with **key**?	
`void`	`remove(Key key)`	remove key-value pair from table	← stay tuned
`Iterator<Key>`	`iterator()`	iterator through keys in table	← stay tuned

Our conventions:

1. Values are not `null`.
2. Method `get()` returns `null` if key not present

enables this code in all implementations:

```
public boolean contains(Key key)
{  return get(key) != null;  }
```

3. Method `put()` overwrites old value with new value.

```
a[key] = val;
```

← Some languages (not Java) allow this notation

ST client: make a dictionary and process lookup requests

Command line arguments

- a comma-separated value (CSV) file
- key field
- value field

Example 1: DNS lookup

URL is key IP is value

```
% java Lookup ip.csv 0 1
adobe.com
192.150.18.60
www.princeton.edu
128.112.128.15
ebay.edu
Not found
```

IP is key URL is value

```
% java Lookup ip.csv 1 0
128.112.128.15
www.princeton.edu
999.999.999.99
Not found
```

```
% more ip.csv
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.math.princeton.edu,128.112.18.11
www.cs.harvard.edu,140.247.50.127
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.econ.yale.edu,128.36.236.74
www.cs.yale.edu,128.36.229.30
espn.com,199.181.135.201
yahoo.com,66.94.234.13
msn.com,207.68.172.246
google.com,64.233.167.99
baidu.com,202.108.22.33
yahoo.co.jp,202.93.91.141
sina.com.cn,202.108.33.32
ebay.com,66.135.192.87
adobe.com,192.150.18.60
163.com,220.181.29.154
passport.net,65.54.179.226
tom.com,61.135.158.237
nate.com,203.226.253.11
cnn.com,64.236.16.20
daum.net,211.115.77.211
blogger.com,66.102.15.100
fastclick.com,205.180.86.4
wikipedia.org,66.230.200.100
rakuten.co.jp,202.72.51.22
...
```

ST client: make a dictionary and process lookup requests

```
public class Lookup
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int keyField = Integer.parseInt(args[1]);
      int valField = Integer.parseInt(args[2]);
      String[] database = in.readAll().split("\\n");

      ST<String, String> st = new ST<String, String>();
      for (int i = 0; i < database.length; i++)
      {
         String[] tokens = database[i].split(",");
         String key = tokens[keyField];
         String val = tokens[valField];
         st.put(key, val);
      }

      while (!StdIn.isEmpty())
      {
         String s = StdIn.readString();
         if (!st.contains(s)) StdOut.println("Not found");
         else                 StdOut.println(st.get(s));
      }
   }
}
```

← process input file

← build symbol table

← process lookups with standard I/O

ST client: make a dictionary and process lookup requests

Command line arguments

- a comma-separated value (CSV) file
- key field
- value field

Example 2: Amino acids

codon is key

name is value

```
% % java Lookup amino.csv 0 3
ACT
Threonine
TAG
Stop
CAT
Histidine
```

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
TAC,Tyr,Y,Tyrosine
TAA,Stop,Stop,Stop
TAG,Stop,Stop,Stop
TGT,Cys,C,Cysteine
TGC,Cys,C,Cysteine
TGA,Stop,Stop,Stop
TGG,Trp,W,Tryptophan
CTT,Leu,L,Leucine
CTC,Leu,L,Leucine
CTA,Leu,L,Leucine
CTG,Leu,L,Leucine
CCT,Pro,P,Proline
CCC,Pro,P,Proline
CCA,Pro,P,Proline
CCG,Pro,P,Proline
CAT,His,H,Histidine
CAC,His,H,Histidine
CAA,Gln,Q,Glutamine
CAG,Gln,Q,Glutamine
CGT,Arg,R,Arginine
CGC,Arg,R,Arginine
CGA,Arg,R,Arginine
CGG,Arg,R,Arginine
ATT,Ile,I,Isoleucine
ATC,Ile,I,Isoleucine
ATA,Ile,I,Isoleucine
ATG,Met,M,Methionine
...
```

ST client: make a dictionary and process lookup requests

Command line arguments

- a comma-separated value (CSV) file
- key field
- value field

```
% more classlist.csv
10,Bo Ling,P03,bling
10,Steven A Ross,P01,saross
10,Thomas Oliver Horton
Conway,P03,oconway
08,Michael R. Corces
Zimmerman,P01A,mcorces
09,Bruce David Halperin,P02,bhalperi
09,Glenn Charles Snyders Jr.,P03,gsnyders
09,Siyu Yang,P01A,siyuyang
08,Taofik O. Kolade,P01,tkolade
09,Katharine Paris
Klosterman,P01A,kkloster
SP,Daniel Gopstein,P01,dgtwo
10,Sauhard Sahi,P01,ssahi
10,Eric Daniel Cohen,P01A,edcohen
09,Brian Anthony Geistwhite,P02,bgeistwh
09,Boris Pivtorak,P01A,pivtorak
09,Jonathan Patrick
Zebrowski,P01A,jzebrows
09,Dexter James Doyle,P01A,ddoyle
09,Michael Weiyang Ye,P03,ye
08,Delwin Uy Olivan,P02,dolivan
08,Edward George Conbeer,P01A,econbeer
09,Mark Daniel Stefanski,P01,mstefans
09,Carter Adams Cleveland,P03,cclevela
10,Jacob Stephen Lewellen,P02,jlewelle
10,Ilya Trubov,P02,itrubov
09,Kenton William Murray,P03,kwmurray
07,Daniel Steven Marks,P02,dmarks
09,Vittal Kadapakkam,P01,vkadapak
10,Eric Ruben Domb,P01A,edomb
07,Jie Wu,P03,jiewu
08,Pritha Ghosh,P02,prithag
10,Minh Quang Anh Do,P01,mqdo
...
```

Example 3: Class lists

login is key name is value

```
% java Lookup classlist.csv 3 1
jsh
Jeffrey Scott Harris
dgtwo
Daniel Gopstein
ye
Michael Weiyang Ye
```

login is key precept is value

```
% java Lookup classlist.csv 3 2
jsh
P01A
dgtwo
P01
```

Keys and Values

Associative array abstraction.

```
a[key] = val;
```

- Unique value associated with each key
- If client presents duplicate key, overwrite to change value.

Key type: several possibilities

1. Assume keys are any generic type, use `equals()` to test equality.
2. Assume keys are `Comparable`, use `compareTo()`.
3. Use `equals()` to test equality and `hashCode()` to scramble key.

Value type. Any generic type.

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: `String`, `Integer`, `BigInteger`.
- Mutable in Java: `Date`, `GregorianCalendar`, `StringBuilder`.

Elementary ST implementations

Unordered array
Ordered array
Unordered linked list
Ordered linked list

Why study elementary implementations?

- API details need to be worked out
- performance benchmarks
- method of choice can be one of these in many situations
- basis for advanced implementations

Always good practice to study elementary implementations

- API
- **basic implementations**
- iterators
- Comparable keys
- challenges

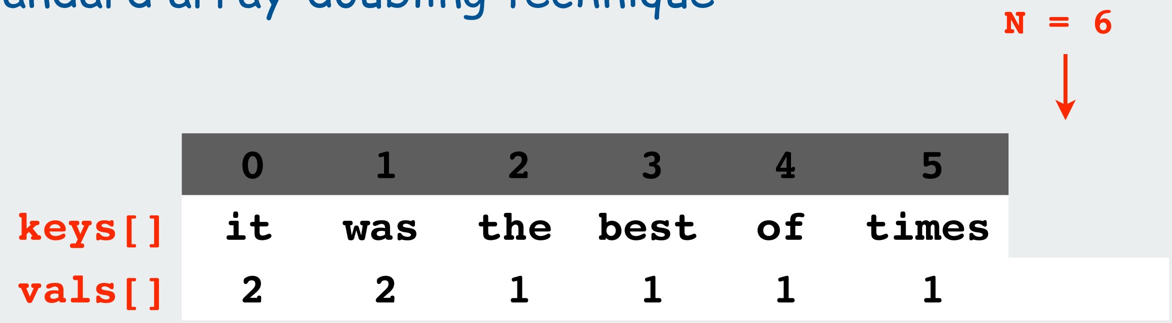
Unordered array ST implementation

Maintain parallel arrays of keys and values.

Instance variables

- array **keys[]** holds the keys.
- array **vals[]** holds the values.
- integer **N** holds the number of entries.

Need to use standard array-doubling technique



N = 6

	0	1	2	3	4	5
keys[]	it	was	the	best	of	times
vals[]	2	2	1	1	1	1

Alternative: define inner type for entries

- space overhead for entry objects
- more complicated code

Unordered array ST implementation (skeleton)

```
public class UnorderedST<Key, Value>
{
   private Value[] vals;
   private Key[] keys;
   private int N = 0;

   public UnorderedST(int maxN)
   {
      keys = (Key[]) new Object[maxN];
      vals = (Value[]) new Object[maxN];
   }

   public boolean isEmpty()
   {  return N == 0; }

   public void put(Key key, Value val)
   // see next slide

   public Value get(Key key)
   // see next slide
}
```

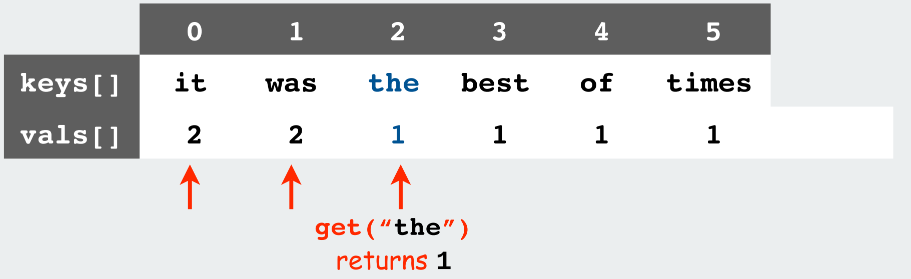
parallel arrays lead to cleaner code than defining a type for entries

standard array doubling code omitted

standard ugly casts

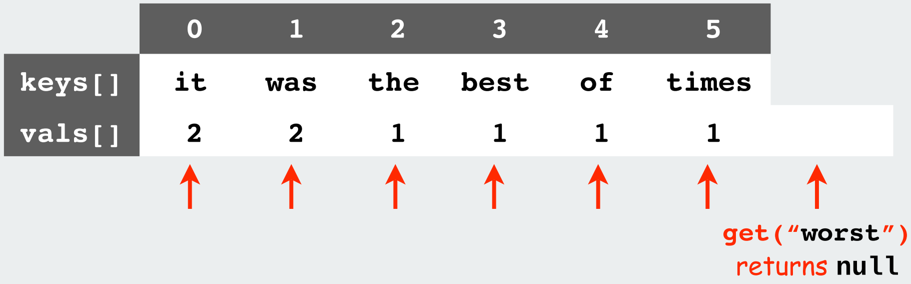
Unordered array ST implementation (search)

```
public Value get(Key key)
{
   for (int i = 0; i < N; i++)
      if (keys[i].equals(key))
         return vals[i];
   return null;
}
```



	0	1	2	3	4	5
keys[]	it	was	the	best	of	times
vals[]	2	2	1	1	1	1

get("the")
returns 1



	0	1	2	3	4	5
keys[]	it	was	the	best	of	times
vals[]	2	2	1	1	1	1

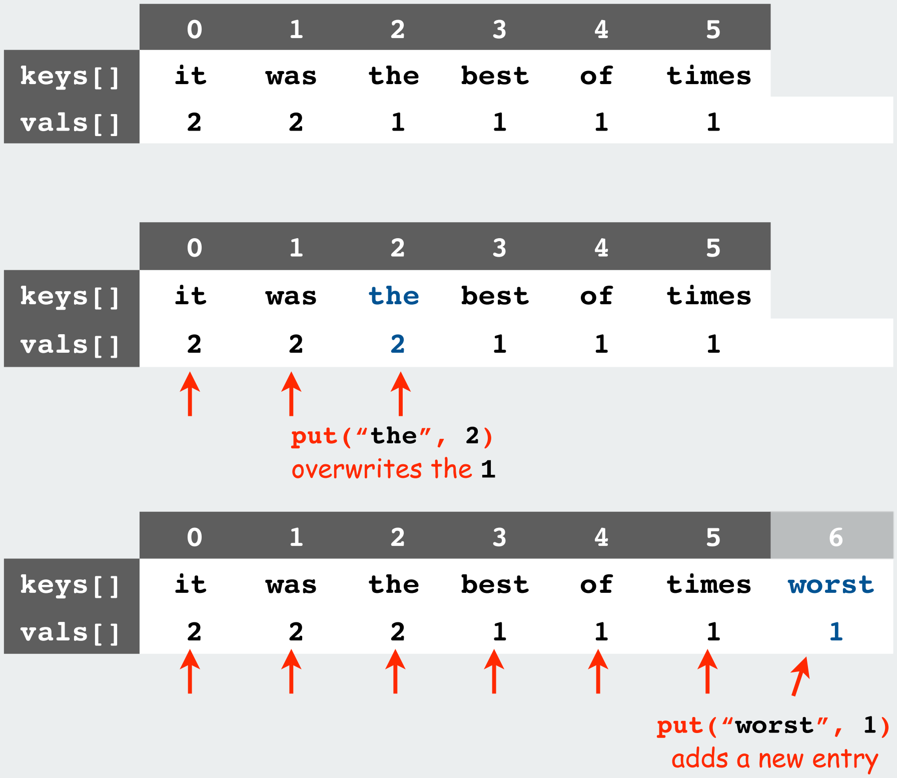
get("worst")
returns null

Key, Value are generic and can be any type

Java convention: all objects implement equals()

Unordered array ST implementation (insert)

```
public void put(Key key, Value val)
{
   int i;
   for (i = 0; i < N; i++)
      if (key.equals(keys[i]))
          break;
   vals[i] = val;
   keys[i] = key;
   if (i == N) N++;
}
```



	0	1	2	3	4	5
keys[]	it	was	the	best	of	times
vals[]	2	2	1	1	1	1

	0	1	2	3	4	5
keys[]	it	was	the	best	of	times
vals[]	2	2	2	1	1	1

put("the", 2)
overwrites the 1

	0	1	2	3	4	5	6
keys[]	it	was	the	best	of	times	worst
vals[]	2	2	2	1	1	1	1

put("worst", 1)
adds a new entry

Associative array abstraction

- must search for key and overwrite with new value if it is there
- otherwise, add new key, value at the end (as in stack)

Java conventions for `equals()`

All objects implement `equals()` but default implementation is `(x == y)`

is the object referred to by x the same object that is referred to by y?

Customized implementations.

`String`, `URL`, `Integer`.

User-defined implementations.

Some care needed (example: type of argument must be `Object`)

Equivalence relation. For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is `true`.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: If `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is `false`.
- Consistency: Multiple calls to `x.equals(y)` return same value.

Implementing `equals()`

Seems easy

```
public       class PhoneNumber
{
   private       int area, exch, ext;

   ...

   public boolean equals(PhoneNumber y)
   {






      PhoneNumber a = this;
      PhoneNumber b = (PhoneNumber) y;
      return (a.area == b.area)
             && (a.exch == b.exch)
             && (a.ext == b.ext);
   }
}
```

Implementing equals()

Seems easy, but requires some care

```
public final class PhoneNumber
{
   private final int area, exch, ext;

   ...

   public boolean equals(   Object    y)
   {
      if (y == this) return true;

      if (y == null) return false;

      if (y.getClass() != this.getClass())
         return false;

      PhoneNumber a = this;
      PhoneNumber b = (PhoneNumber) y;
      return (a.area == b.area)
             && (a.exch == b.exch)
             && (a.ext == b.ext);
   }
}
```

no safe way to use with inheritance

Must be **Object**.
Why? Experts still debate.

← Optimize for true object equality

← If I'm executing this code,
I'm not null.

← Objects must be in the same class.

Linked list ST implementation

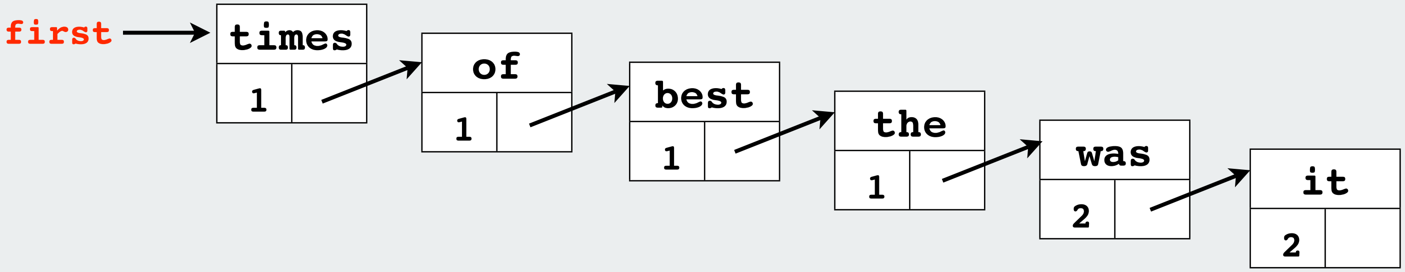
Maintain a linked list with keys and values.

inner **`Node`** class

- instance variable **`key`** holds the key
- instance variable **`val`** holds the value

instance variable

- **`Node first`** refers to the first node in the list



Linked list ST implementation (skeleton)

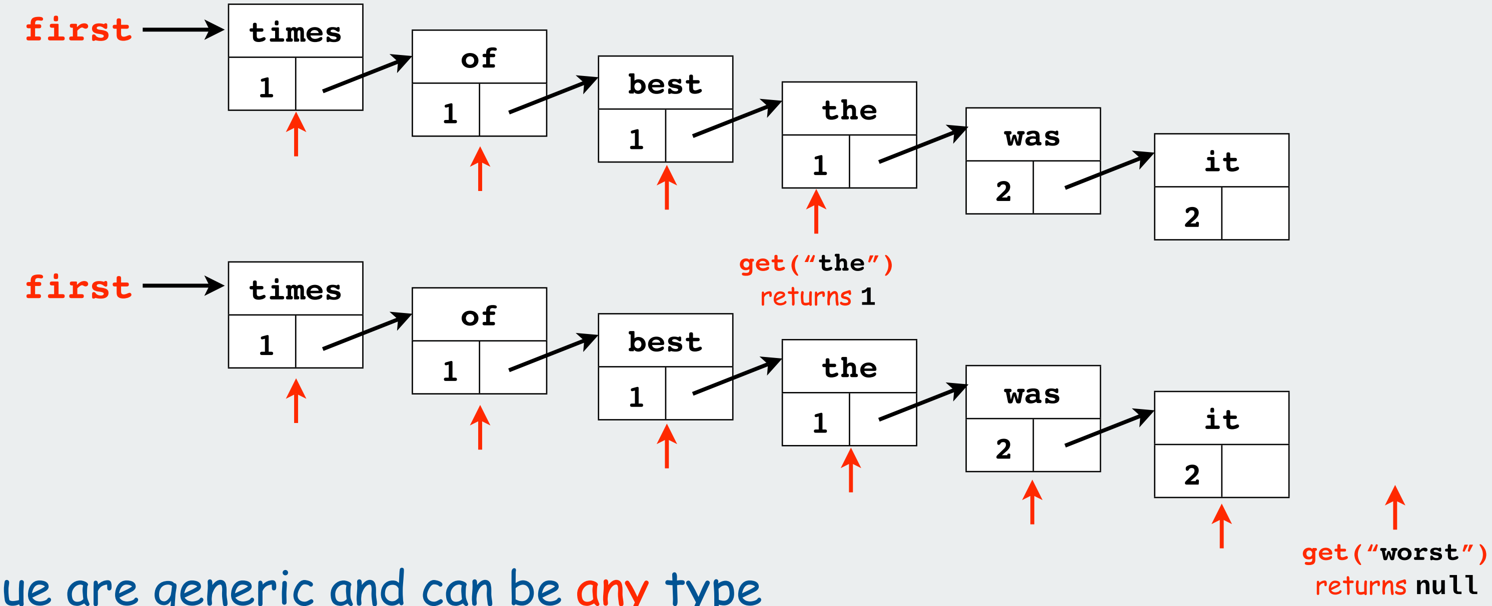
```
public class LinkedListST<Key, Value>
{
    private Node first;

    private class Node
    {
        Key key;
        Value val;
        Node next;
        Node(Key key, Value val, Node next)
        {
            this.key    = key;
            this.val = val;
            this.next   = next;
        }
    }

   public void put(Key key, Value val)
   // see next slides

   public Val get(Key key)
   // see next slides
}
```

← instance variable

← inner class

Linked list ST implementation (search)

```
public Value get(Key key)
{
   for (Node x = first; x != null; x = x.next))
      if (key.equals(x.key))
         return vals[i];
   return null;
}
```



get("the")
returns 1

get("worst")
returns null

Key, Value are generic and can be any type

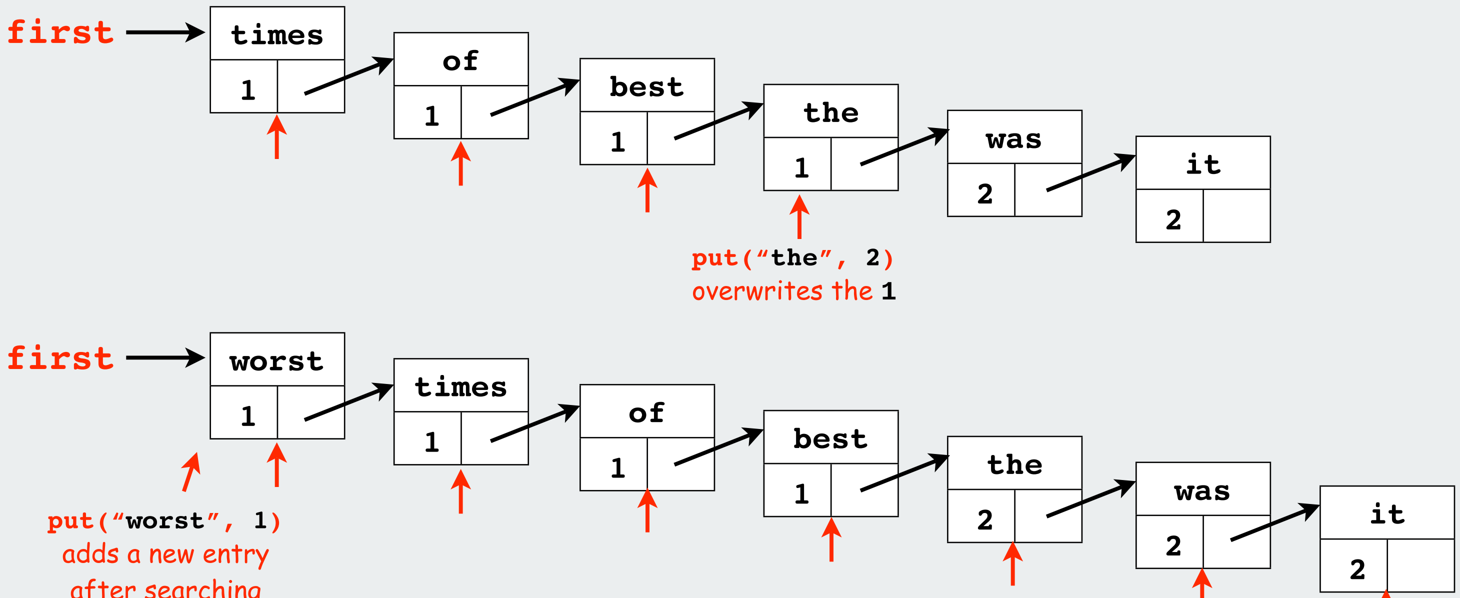
Java convention: all objects implement equals()

Linked list ST implementation (insert)

```
public void put(Key key, Value val)
{
   for (Node x = first; x != null; x = x.next)
      if (key.equals(x.key))
         { x.value = value; return; }
   first = new Node(key, value, first);
}
```

Associative array abstraction

- must search for key and, if it is there, overwrite with new value
- otherwise, add new key, value at the beginning (as in stack)



put("the", 2) overwrites the 1

put("worst", 1) adds a new entry after searching the entire list

- API
- basic implementations
- iterators
- Comparable keys
- challenges

Iterators

Symbol tables should be `Iterable`

Q. What is `Iterable`?

A. Implements `iterator()`

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

Q. What is an `Iterator`?

A. Implements `hasNext()` and `next()`.

`java.util.Iterator`

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();
}
```

← optional in Java
use at your own risk

Q. Why should symbol tables be iterable?

A. Java language supports elegant client code for iterators

"foreach" statement

```
for (String s: st)
   StdOut.println(st.get(s));
```

equivalent code

```
Iterator<String> i = st.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(st.get(s));
}
```

Iterable ST client: count frequencies of occurrence of input strings

Standard input: A file (of strings)

Standard output: All the distinct strings in the file with frequency

```
% more tiny.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness


% java FrequencyCount < tiny.txt
2 age
1 best
1 foolishness
4 it
4 of
4 the
2 times
4 was
1 wisdom
1 worst
```

tiny example
24 words
10 distinct

real example
137177 words
9888 distinct

```
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair
we had everything before us
we had nothing before us
...
% java FrequencyCount < tale.txt
2941 a
1 aback
1 abandon
10 abandoned
1 abandoning
1 abandonment
1 abashed
1 abate
1 abated
5 abbaye
2 abed
1 abhorrence
1 abided
1 abiding
1 abilities
2 ability
1 abject
1 ablaze
17 able
1 abnegating
```

Iterable ST client: count frequencies of occurrence of input strings

```
public class FrequencyCount
{
   public static void main(String[] args)
   {
      ST<String, Integer> st;
      st = new ST<String, Integer>();

      while (!StdIn.isEmpty())
      {
         String key = StdIn.readString();

         if (!st.contains(key))
            st.put(key, 1);
         else
            st.put(key, st.get(key) + 1);
      }

      for (String s: st)
         StdOut.println(st.get(s) + " " + s);

   }
}
```

← read a string

← insert

← increment

← print all strings

Note: Only slightly more work required to build an index of all of the places where each key occurs in the text.

Iterators for array, linked list ST implementations

```
import java.util.Iterator;
public class UnorderedST<Key, Value>
             implements Iterable<Key>
{
    ...

    public Iterator<Key> iterator()
    { return new ArrayIterator(); }

    private class ArrayIterator
       implements Iterator<Key>
    {
        private int i = 0;

        public boolean hasNext()
        {  return i < N;  }

        public void remove() { }

        public Key next()
        {  return keys[i++];  }
    }
}
```

```
import java.util.Iterator;
public class LinkedListST<Key, Value>
             implements Iterable<Key>
{
    ...

    public Iterator<Key> iterator()
    { return new ListIterator();  }

    private class ListIterator
             implements Iterator<Key>
    {
        private Node current = first;

        public boolean hasNext()
        {  return current != null;  }

        public void remove() { }

        public Key next()
        {
            Key key = current.key;
            current = current.next;
            return key;
        }
    }
}
```

Iterable ST client: A problem?

Use **UnorderedST** in **FrequencyCount**

```
% more tiny.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness


% java FrequencyCount < tiny.txt
4 it
4 was
4 the
1 best
4 of
2 times
1 worst
2 age
1 wisdom
1 foolishness
```

Use **LinkedListST** in **FrequencyCount**

```
% more tiny.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness


% java FrequencyCount < tiny.txt
1 foolishness
1 wisdom
2 age
1 worst
2 times
4 of
1 best
4 the
4 was
4 it
```

Clients who use **Comparable** keys might expect ordered iteration

- not a requirement for some clients
- not a problem if postprocessing, e.g. with sort or grep
- not in API

- API
- basic implementations
- iterators
- **Comparable keys**
- challenges

Ordered array ST implementation

Assume that keys are **Comparable**

Maintain parallel arrays with keys and values that are sorted by key.

Instance variables

- **keys[i]** holds the **i**th smallest key
- **vals[i]** holds the value associated with the **i**th smallest key
- integer **N** holds the number of entries.

Note: no duplicate keys

N = 6 ↓

	0	1	2	3	4	5
keys[]	best	it	of	the	times	was
vals[]	1	2	1	1	1	2

Need to use standard array-doubling technique

Two reasons to consider using ordered arrays

- provides ordered iteration (for free)
- can use binary search to significantly speed up search

Ordered array ST implementation (skeleton)

```
public class OrderedST
          <Key extends Comparable<Key>, Value>
          implements Iterable<Key>
{
   private Value[] vals;
   private Key[] keys;
   private int N = 0;
   public OrderedST(int maxN)
   {
     keys = (Key[]) new Object[maxN];
     vals = (Value[]) new Object[maxN];
   }

   public boolean isEmpty()
   {  return N == 0; }


   public void put(Key key, Value val)
   // see next slides

   public Val get(Key key)
   // see next slides
}
```

← standard array iterator code omitted

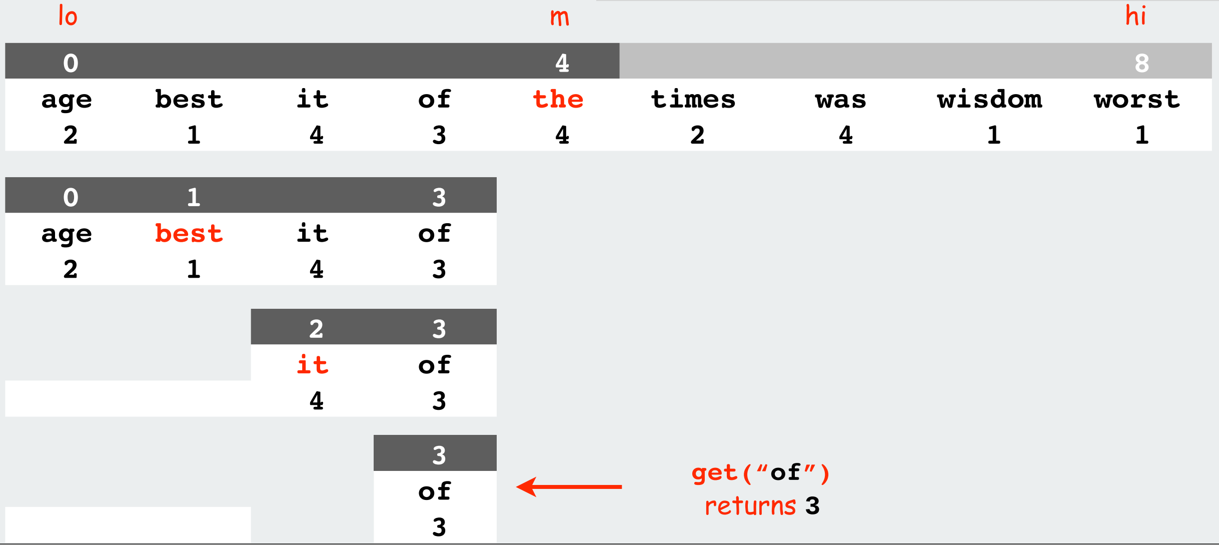
← standard array doubling code omitted

Ordered array ST implementation (search)

Keeping array in order enables binary search algorithm

```
public Value get(Key key)
 {
      int i = bsearch(key);
      if (i == -1) return null;
      return vals[i];
 }
```

```
private int bsearch(Key key)
{
    int lo = 0, hi = N-1;
    while (lo <= hi)
    {
        int m = lo + (hi - lo) / 2;
        int cmp = key.compareTo(keys[m]);
        if      (cmp < 0) hi = m - 1;
        else if (cmp > 0) lo = m + 1;
        else return m;
    }
   return -1;
}
```



lo				m				hi
0				4				8
age	best	it	of	the	times	was	wisdom	worst
2	1	4	3	4	2	4	1	1

0	1		3
age	best	it	of
2	1	4	3

2	3
it	of
4	3

3
of
3

get("of")
returns 3

Binary search analysis: Comparison count

Def. $T(N) \equiv$ number of comparisons to binary search in an ST of size N

$$= \underset{\uparrow \atop \text{left or right half}}{T(N/2)} + \underset{\uparrow \atop \text{middle}}{1}$$

Binary search recurrence $T(N) = T(N/2) + 1$

for $N > 1$, with $T(1) = 0$

- not quite right for odd N
- same recurrence holds for many algorithms
- same number of comparisons for any input of size N.

Solution of binary search recurrence $T(N) \sim \lg N$

- true for all N
- easy to prove when N is a power of 2.

can then use induction for general N
(see COS 340)

Binary search recurrence: Proof by telescoping

$$T(N) \ = \ T(N/2) \ + \ 1$$

for $N > 1$, with $T(1) = 0$

(assume that N is a power of 2)

Pf.

$$\begin{aligned}
T(N) &= T(N/2) + 1 && \text{given} \\
&= T(N/4) + 1 + 1 && \text{telescope (apply to first term)} \\
&= T(N/8) + 1 + 1 + 1 && \text{telescope again} \\
&\ \ \vdots && \\
&= T(N/N) + 1 + 1 + \ldots + 1 && \text{stop telescoping, } T(1) = 0 \\
&= \lg N &&
\end{aligned}$$

$$T(N) \ = \ \lg N$$

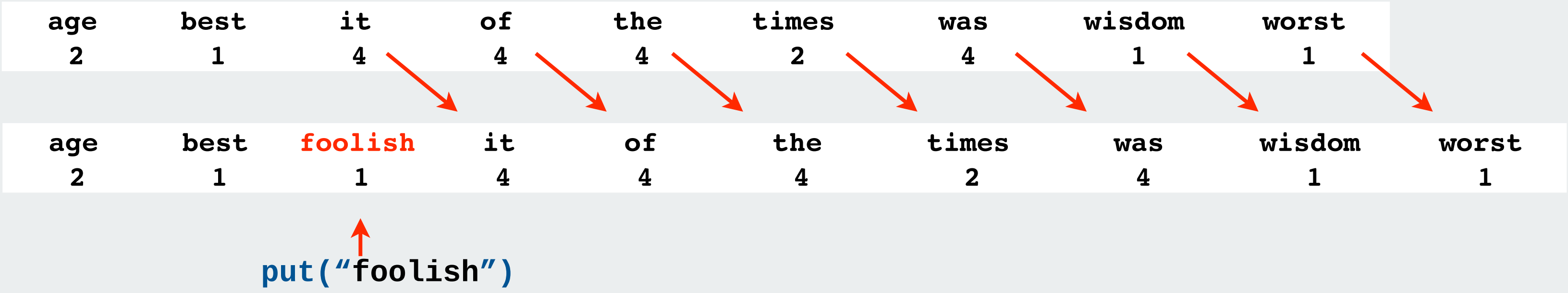
Ordered array ST implementation (insert)

Binary search is little help for `put()`: still need to move larger keys

```
public Val put(Key key, Value val)
 {
     int i = bsearch(key);
     if (i != -1)
     {  vals[i] = val; return;  }

     for ( i = N; i > 0; i-- )
     {
         if key.compareTo(keys[i-1] < 0) break;
         keys[i] = keys[i-1];
         vals[i] = vals[i-1];
     }
     vals[i] = val;
     keys[i] = key;
     N++;

 }
```

overwrite with new value if key in table

move larger keys to make room if key not in table



age	best	it	of	the	times	was	wisdom	worst	
2	1	4	4	4	2	4	1	1	
age	best	foolish	it	of	the	times	was	wisdom	worst
2	1	1	4	4	4	2	4	1	1

put("foolish")

Ordered array ST implementation: an important special case

Test whether key is equal to or greater than largest key

```
public Val put(Key key, Value val)
{
   if (key.compareTo(keys[N-1]) == 0)
   {  vals[N-1] = val; return;   }

   if (key.compareTo(keys[N-1] > 0)
   {
      vals[N] = val;
      keys[N] = key;
      N++;
      return;
   }
}
```

If either test succeeds, constant-time insert!

Method of choice for some clients:

- sort database by key
- insert N key-value pairs in order by key
- support searches that never use more than lg N compares
- support occasional (expensive) inserts

Ordered linked-list ST implementation

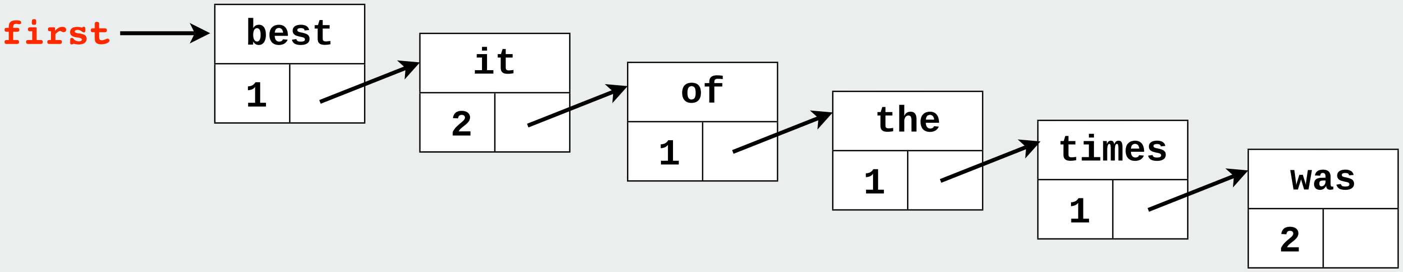
Binary search depends on array indexing for efficiency.

Jump to the middle of a linked list?

Advantages of keeping linked list in order for **`Comparable`** keys:

- support ordered iterator (for free)
- cuts search/insert time in half (on average) for random search/insert

[code omitted]



- API
- basic implementations
- iterators
- Comparable keys
- challenges

Searching challenge 1A:

Problem: maintain symbol table of song names for an iPod

Assumption A: hundreds of songs

Which searching method to use?

1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough

Searching challenge 1B:

Problem: maintain symbol table of song names for an iPod
Assumption B: thousands of songs

Which searching method to use?

1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough

Searching challenge 2A:

Problem: IP lookups in a web monitoring device

Assumption A: billions of lookups, millions of distinct addresses

Which searching method to use?

1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough

Searching challenge 2B:

Problem: IP lookups in a web monitoring device
Assumption B: billions of lookups, thousands of distinct addresses

Which searching method to use?

1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough

Searching challenge 3:

Problem: Frequency counts in “Tale of Two Cities”

Assumptions: book has 135,000+ words
about 10,000 distinct words

Which searching method to use?

1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn’t matter much, all fast enough

Searching challenge 4:

Problem: Spell checking for a book

Assumptions: dictionary has 25,000 words
book has 100,000+ words

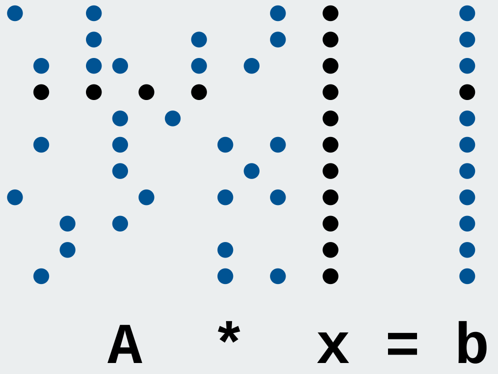
Which searching method to use?

1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough

Searching challenge 5:

Problem: Sparse matrix-vector multiplication

Assumptions: matrix dimension is billions by billions
average number of nonzero entries/row is ~10

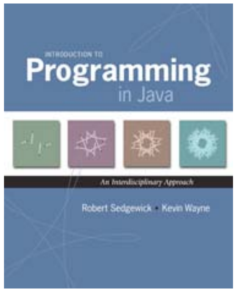


Which searching method to use?

1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough

Summary and roadmap

- basic algorithmics
- no generics
- more code
- more analysis
- equal keys in ST (not associative arrays)



- iterators
- ST as associative array (all keys distinct)
- BST implementations
- applications

Symbol Tables

- API
- basic implementations
- iterators
- Comparable keys
- challenges

References:
Algorithms in Java, Chapter 12
Intro to Programming, Section 4.4
http://www.cs.princeton.edu/introalgsds/41st

- distinguish algs by operations on keys
- ST as associative array (all keys distinct)
- important special case for binary search
- challenges

Elementary implementations: summary

studying STs for the midterm? Start here.

implementation	worst case		average case		ordered iteration?	operations on keys
	search	insert	search	insert		
unordered array	N	N	N/2	N/2	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
unordered list	N	N	N/2	N	no	`equals()`
ordered list	N	N	N/2	N/2	yes	`compareTo()`

Next challenge.

Efficient implementations of search and insert and ordered iteration for arbitrary sequences of operations.

(ordered array meets challenge if keys arrive approximately in order)

Binary Search Trees

- basic implementations
- randomized BSTs
- deletion in BSTs

References:
Algorithms in Java, Chapter 12
Intro to Programming, Section 4.4
http://www.cs.princeton.edu/introalgsds/43bst

Elementary implementations: summary

implementation	worst case		average case		ordered iteration?	operations on keys
	search	insert	search	insert		
unordered array	N	N	N/2	N/2	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
unordered list	N	N	N/2	N	no	`equals()`
ordered list	N	N	N/2	N/2	yes	`compareTo()`

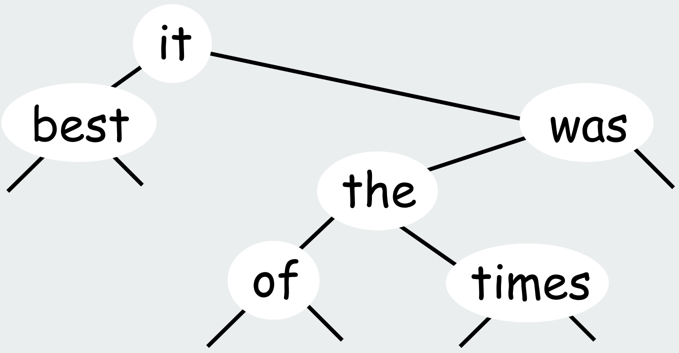
Challenge:

Efficient implementations of `get()` and `put()` and ordered iteration.

- basic implementations
- randomized BSTs
- deletion in BSTs

Binary Search Trees (BSTs)

Def. A BINARY SEARCH TREE is a binary tree in symmetric order.



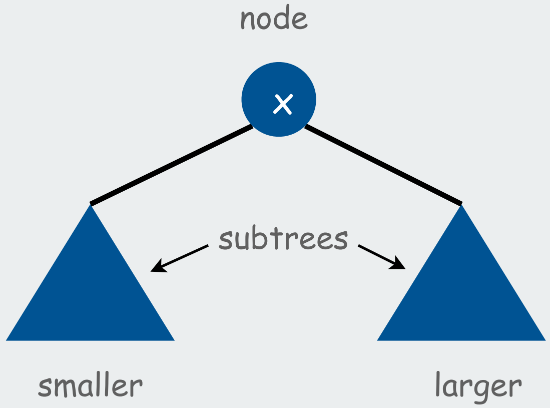
A binary tree is either:

- empty
- a key-value pair and two binary trees [neither of which contain that key]

equal keys ruled out to facilitate associative array implementations

Symmetric order means that:

- every node has a key
- every node's key is
  - larger than all keys in its left subtree
  - smaller than all keys in its right subtree



BST representation

A **BST** is a reference to a **Node**.

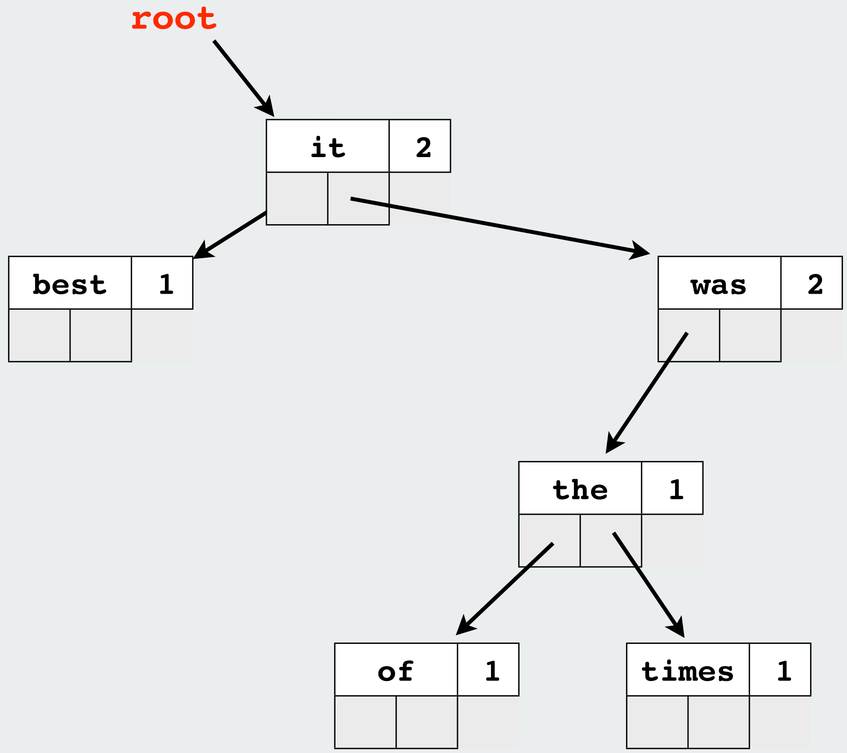
A **Node** is comprised of four fields:

- A key and a value.
- A reference to the left and right subtree.

smaller keys   larger keys

```
private class Node
{
   Key key;
   Value val;
   Node left, right;
}
```

Key and **Value** are generic types;
Key is **Comparable**



BST implementation (skeleton)

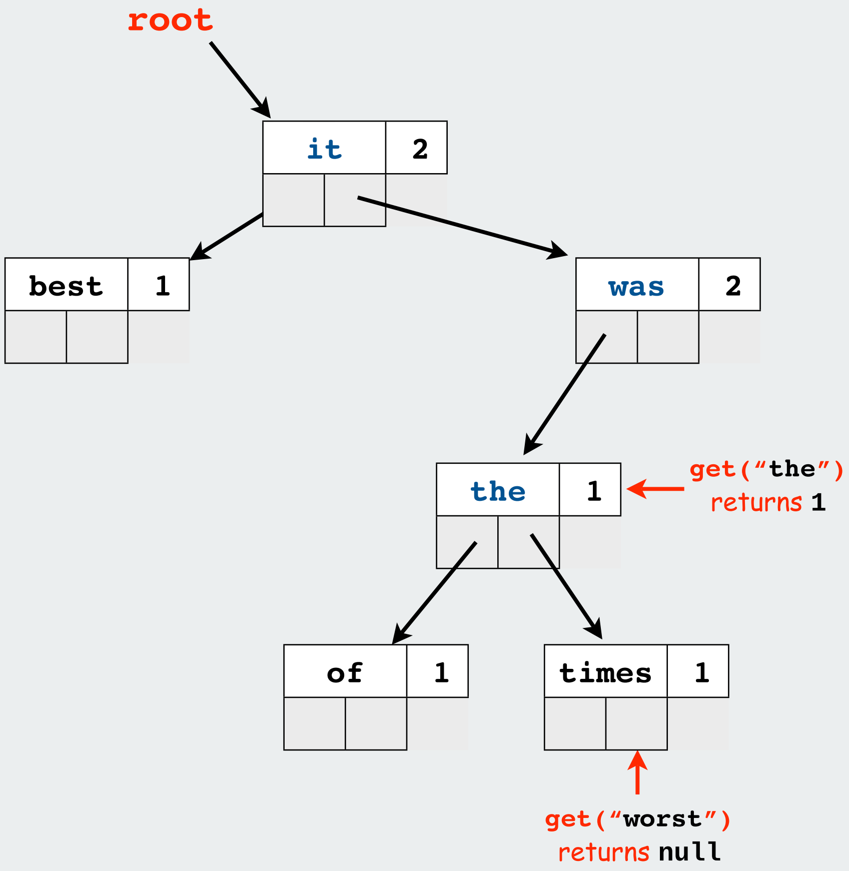
```
public class BST<Key extends Comparable<Key>, Value>
             implements Iterable<Key>
{
   private Node root;

   private class Node
   {
       Key key;
       Value val;
       Node left, right;
       Node(Key key, Value val)
       {
           this.key = key;
           this.val = val;
       }
   }


  public void put(Key key, Value val)
  // see next slides

  public Val get(Key key)
  // see next slides
}
```

← instance variable

← inner class

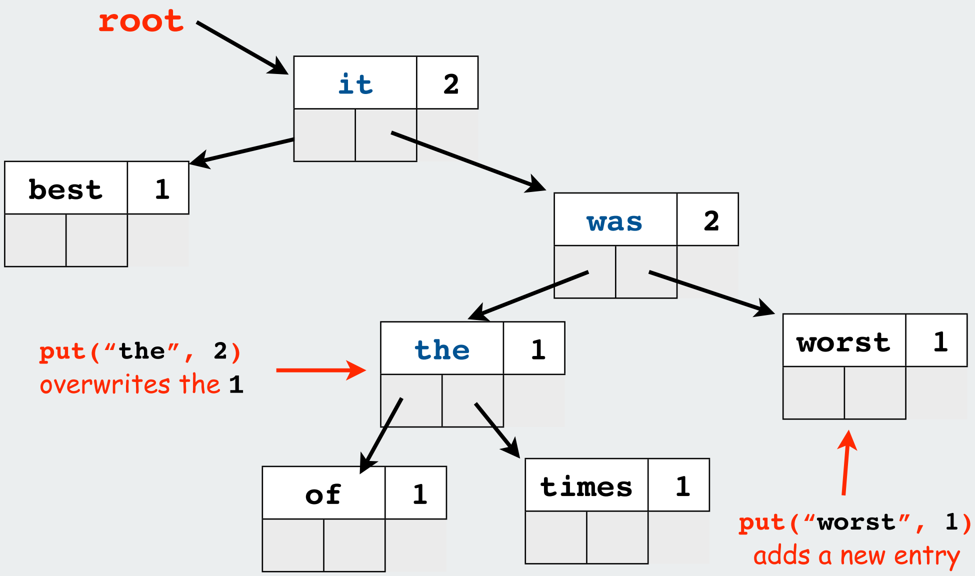
BST implementation (search)

```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if (cmp == 0)      return x.val;
      else if (cmp < 0) x = x.left;
      else if (cmp > 0) x = x.right;
   }
   return null;
}
```



BST implementation (insert)

```
public void put(Key key, Value val)
{  root = put(root, key, val);  }
```



put("the", 2)
overwrites the 1

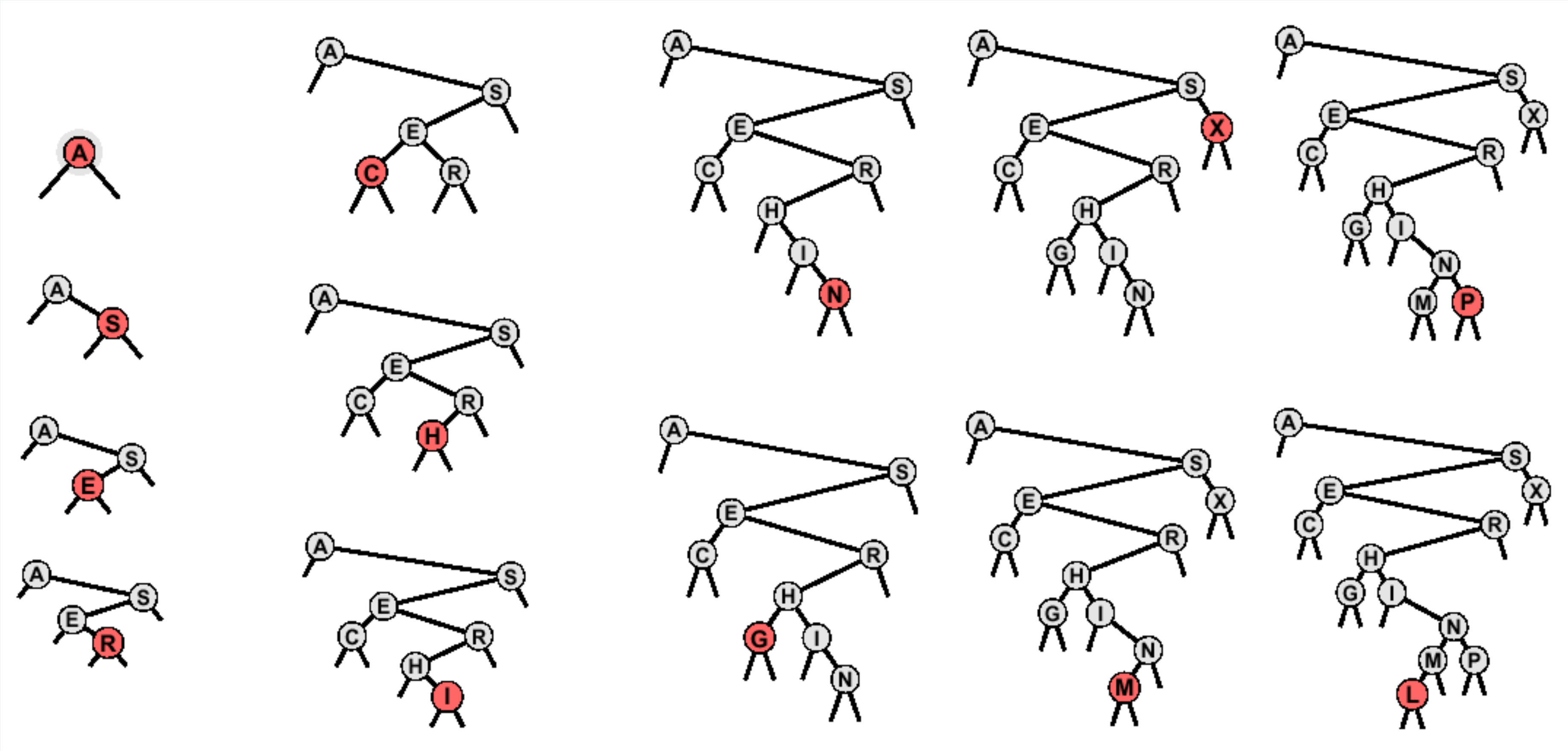
put("worst", 1)
adds a new entry

Caution: tricky recursive code.
Read carefully!

```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if (cmp == 0)     x.val = val;
   else if (cmp < 0) x.left  = put(x.left,  key, val);
   else if (cmp > 0) x.right = put(x.right, key, val);
   return x;
}
```

BST: Construction

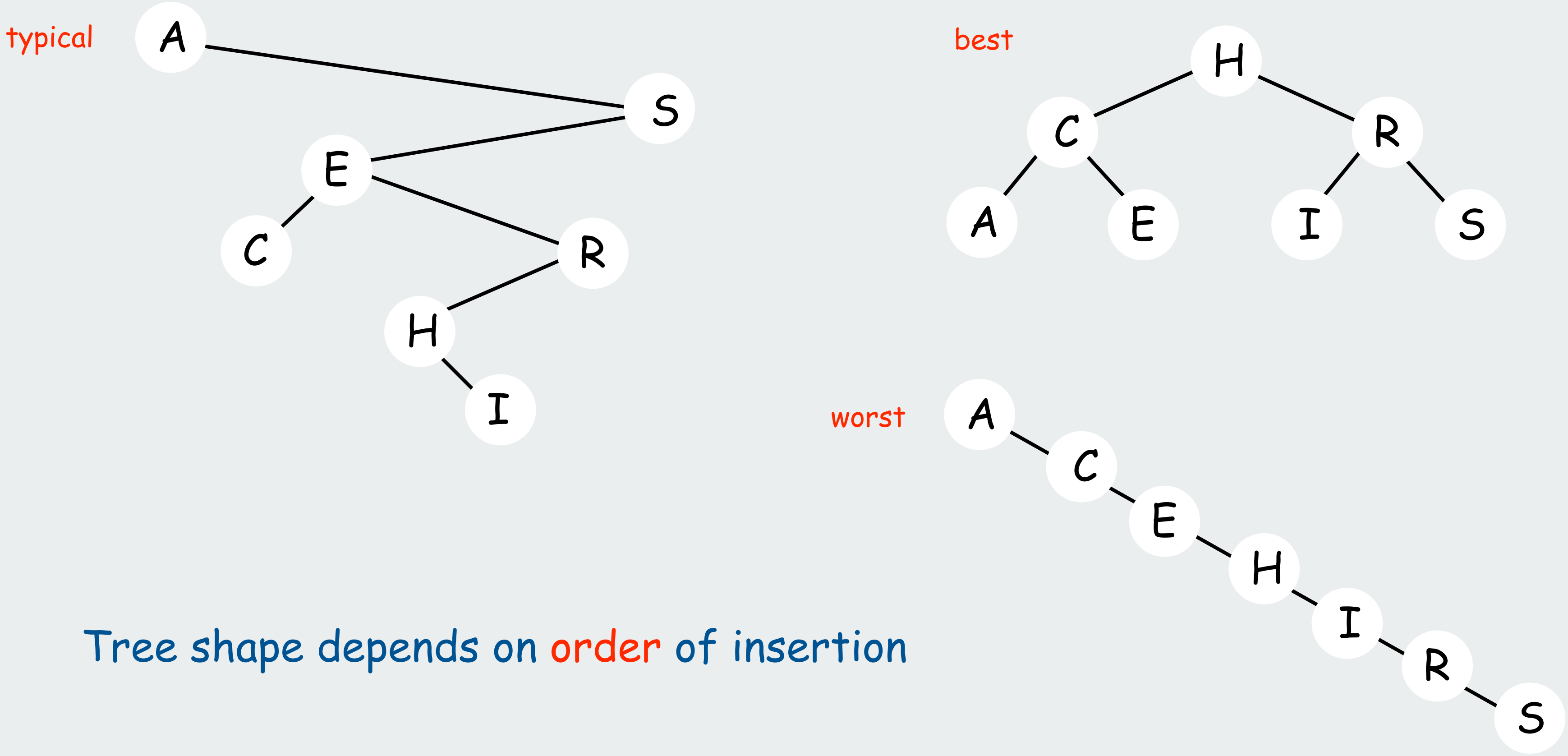
Insert the following keys into BST. **A S E R C H I N G X M P L**



Tree Shape

Tree shape.

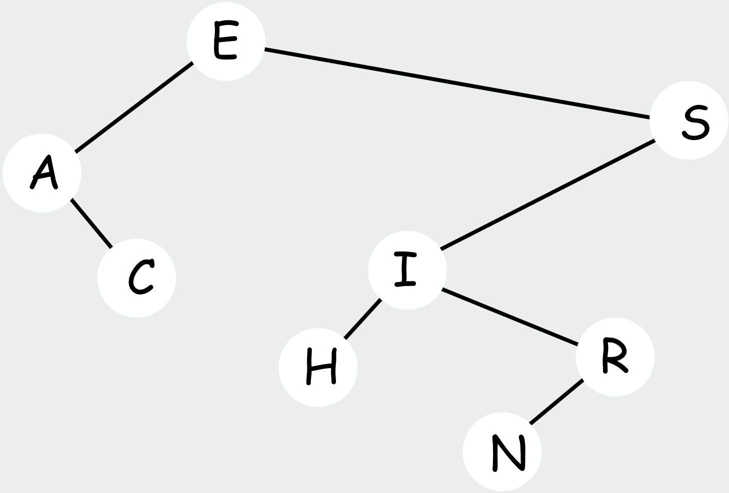
- Many BSTs correspond to same input data.
- Cost of search/insert is proportional to depth of node.



Tree shape depends on order of insertion

BST implementation: iterator?

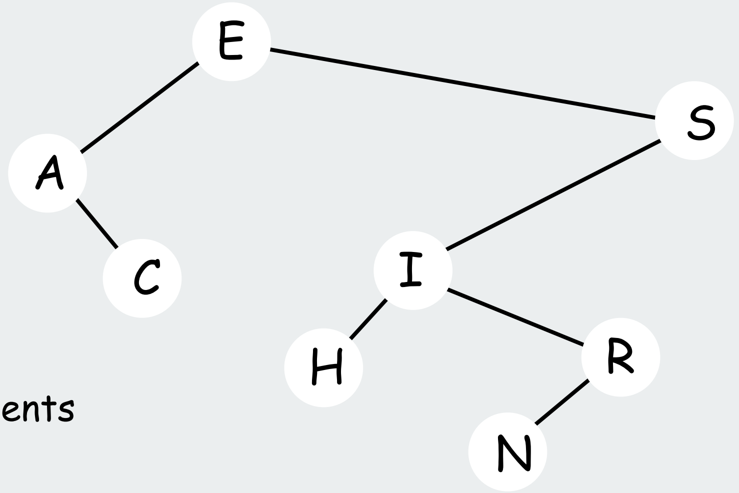
```
public Iterator<Key> iterator()
{ return new BSTIterator(); }

private class BSTIterator
              implements Iterator<Key>
{
   BSTIterator()
   {   }

   public boolean hasNext()
   {   }

   public Key next()
   {   }
}
```



BST implementation: iterator?

Approach: mimic recursive inorder traversal

```
public void visit(Node x)
{
   if (x == null) return;
   visit(x.left)
   StdOut.println(x.key);
   visit(x.right);
}
```



Stack contents

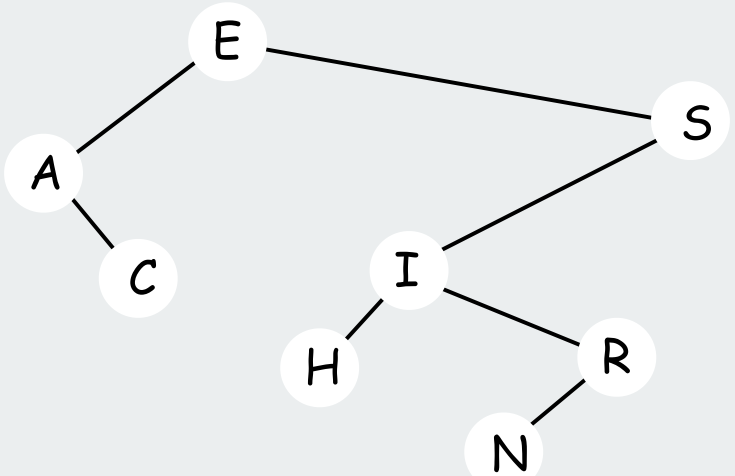
```
visit(E)                  E
  visit(A)                A  E
    print A        A      E
    visit(C)              C  E
      print C      C      E
  print E          E
  visit(S)                S
    visit(I)              I  S
      visit(H)     H      H  I  S
        print H           I  S
      print I      I      S
      visit(R)            R  S
        visit(N)          N  R  S
          print N  N      R  S
        print R    R      S
    print S        S
```

To process a node

- follow left links until empty (pushing onto stack)
- pop and process
- process node at right link

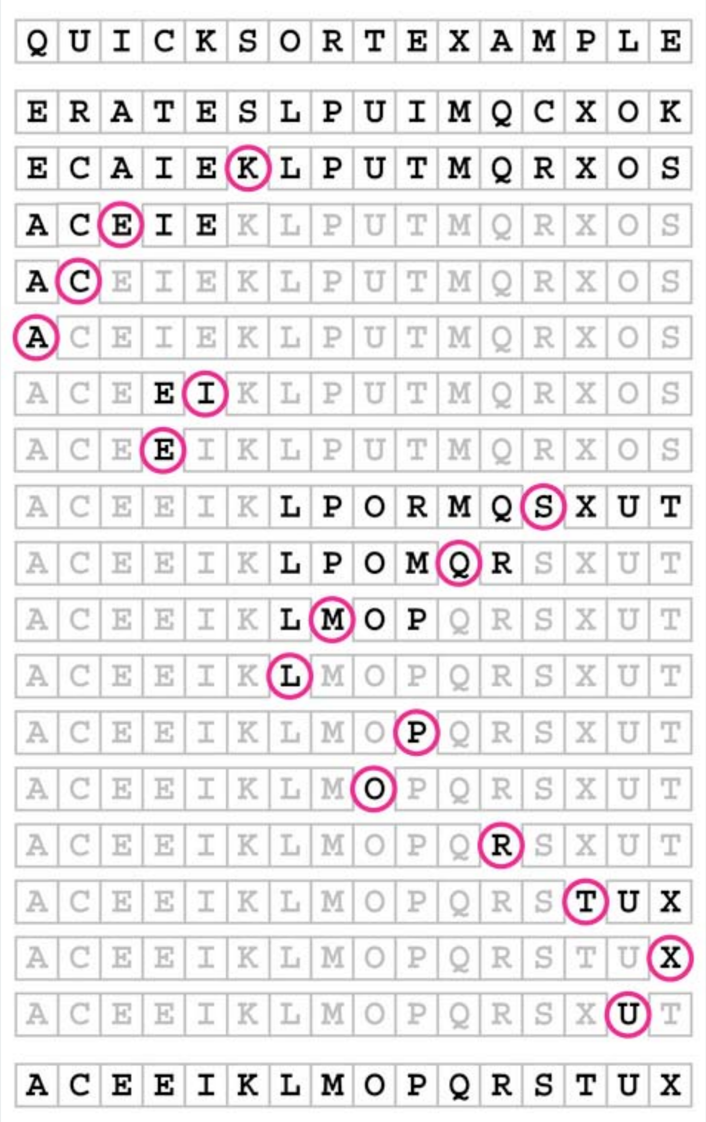
BST implementation: iterator

```
public Iterator<Key> iterator()
{ return new BSTIterator(); }

private class BSTIterator
              implements Iterator<Key>
{
   private Stack<Node>
             stack = new Stack<Node>();

   private void pushLeft(Node x)
   {
       while (x != null)
       {  stack.push(x); x = x.left; }
   }

   BSTIterator()
   {  pushLeft(root); }

   public boolean hasNext()
   {  return !stack.isEmpty(); }

   public Key next()
   {
      Node x = stack.pop();
      pushLeft(x.right);
      return x.key;
   }
}
```

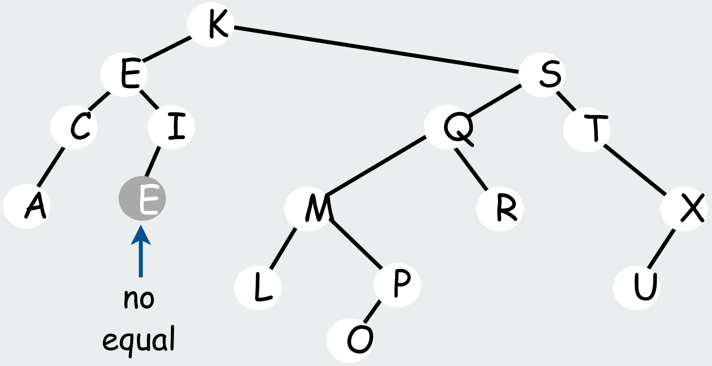


	A	E	
A	C	E	
C	E		
E	H	I	S
H	I	S	
I	N	R	S
N	R	S	
R	S		
S			

1-1 correspondence between BSTs and Quicksort partitioning



```
Q U I C K S O R T E X A M P L E
E R A T E S L P U I M Q C X O K
E C A I E K L P U T M Q R X O S
A C E I E K L P U T M Q R X O S
A C E I E K L P U T M Q R X O S
A C E I E K L P U T M Q R X O S
A C E E I K L P U T M Q R X O S
A C E E I K L P U T M Q R X O S
A C E E I K L P O R M Q S X U T
A C E E I K L P O M Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S T U X
A C E E I K L M O P Q R S T U X
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S T U X
```



no equal keys

BSTs: analysis

Theorem. If keys are inserted in random order, the expected number of comparisons for a search/insert is about 2 ln N.

≈ 1.38 lg N, variance = O(1)

Proof: 1-1 correspondence with quicksort partitioning

Theorem. If keys are inserted in random order, height of tree is proportional to lg N, except with exponentially small probability.

mean ≈ 6.22 lg N, variance = O(1)

But… Worst-case for search/insert/height is N.

e.g., keys inserted in ascending order

Searching challenge 3 (revisited):

Problem: Frequency counts in “Tale of Two Cities”

Assumptions: book has 135,000+ words
about 10,000 distinct words

Which searching method to use?

1) unordered array
2) unordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn’t matter much, all fast enough
6) BSTs

insertion cost < 10000 * 1.38 * lg 10000 < .2 million
lookup cost < 135000 * 1.38 * lg 10000 < 2.5 million

Elementary implementations: summary

implementation	guarantee		average case		ordered iteration?	operations on keys
	search	insert	search	insert		
unordered array	N	N	N/2	N/2	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
unordered list	N	N	N/2	N	no	`equals()`
ordered list	N	N	N/2	N/2	yes	`compareTo()`
BST	N	N	1.38 lg N	1.38 lg N	yes	`compareTo()`

Next challenge:

Guaranteed efficiency for **`get()`** and **`put()`** and ordered iteration.

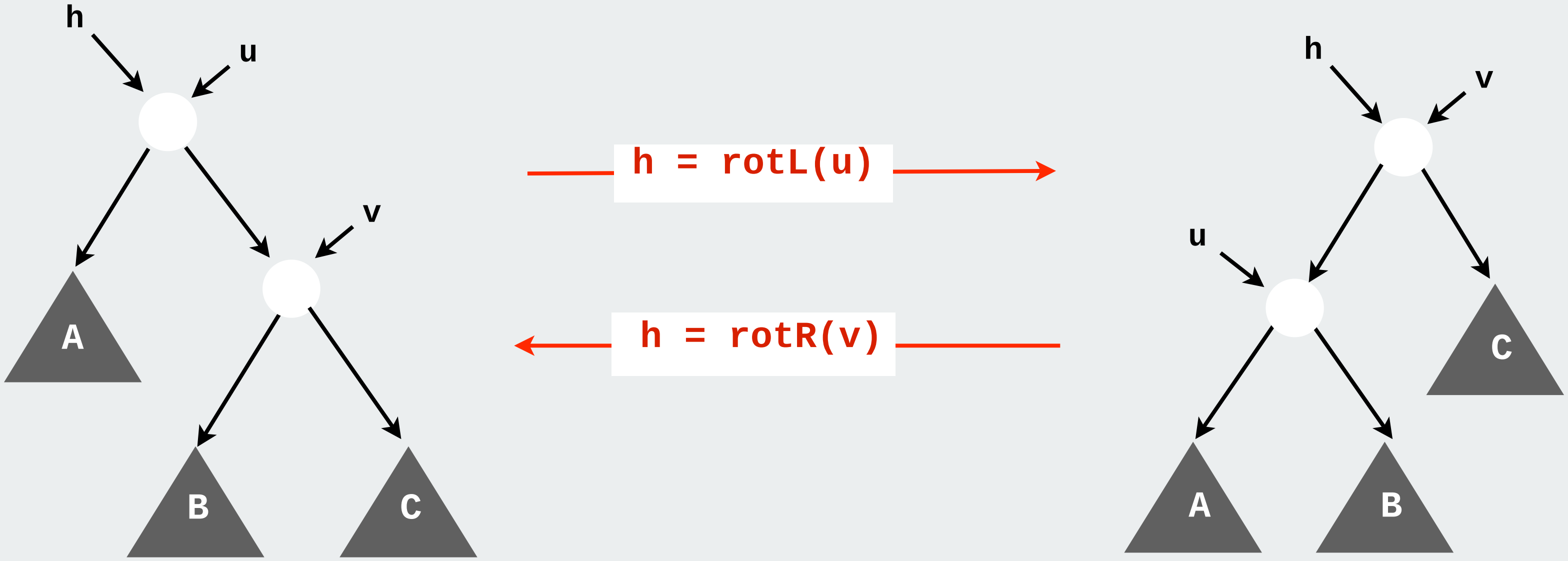
- basic implementations
- randomized BSTs
- deletion in BSTs

Rotation in BSTs

Two fundamental operations to rearrange nodes in a tree.

- maintain symmetric order.
- local transformations (change just 3 pointers).
- basis for advanced BST algorithms

Strategy: use rotations on insert to adjust tree shape to be more balanced



`h = rotL(u)`

`h = rotR(v)`

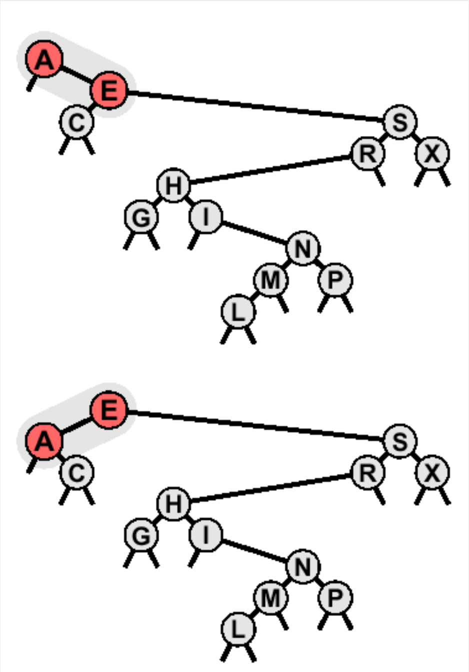
Key point: no change in search code (!)

Rotation

Fundamental operation to rearrange nodes in a tree.

- easier done than said
- raise some nodes, lowers some others

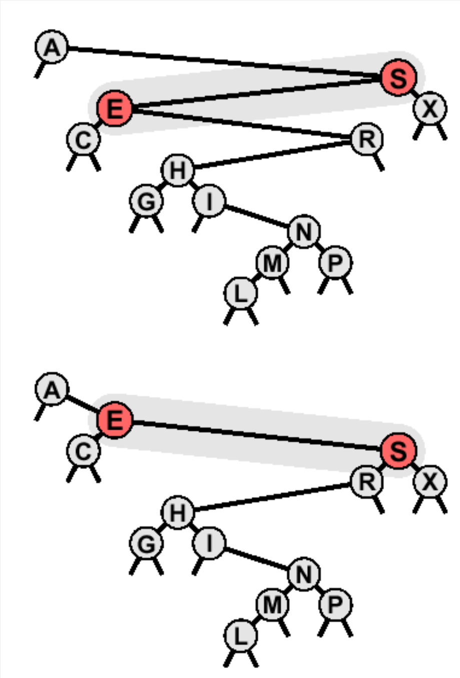
root = rotL(A)



```
private Node rotL(Node h)
{
   Node v = h.r;
   h.r = v.l;
   v.l = h;
   return v;
}
```

```
private Node rotR(Node h)
{
   Node u = h.l;
   h.l = u.r;
   u.r = h;
   return u;
}
```

A.left = rotR(S)



Recursive BST Root Insertion

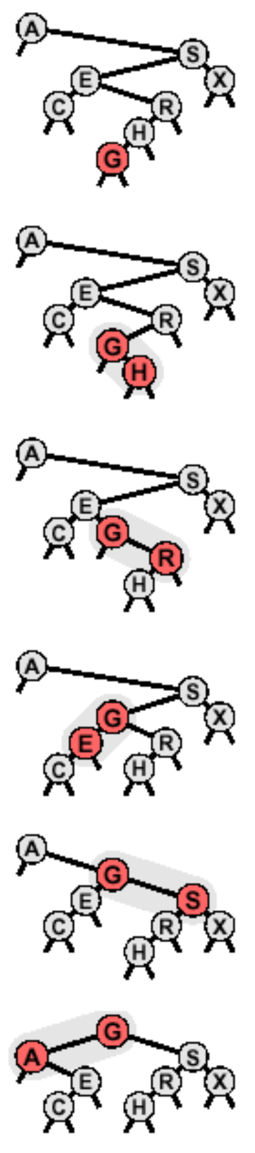
Root insertion: insert a node and make it the new root.

- Insert as in standard BST.
- Rotate inserted node to the root.
- Easy recursive implementation

Caution: very tricky recursive code.
Read very carefully!

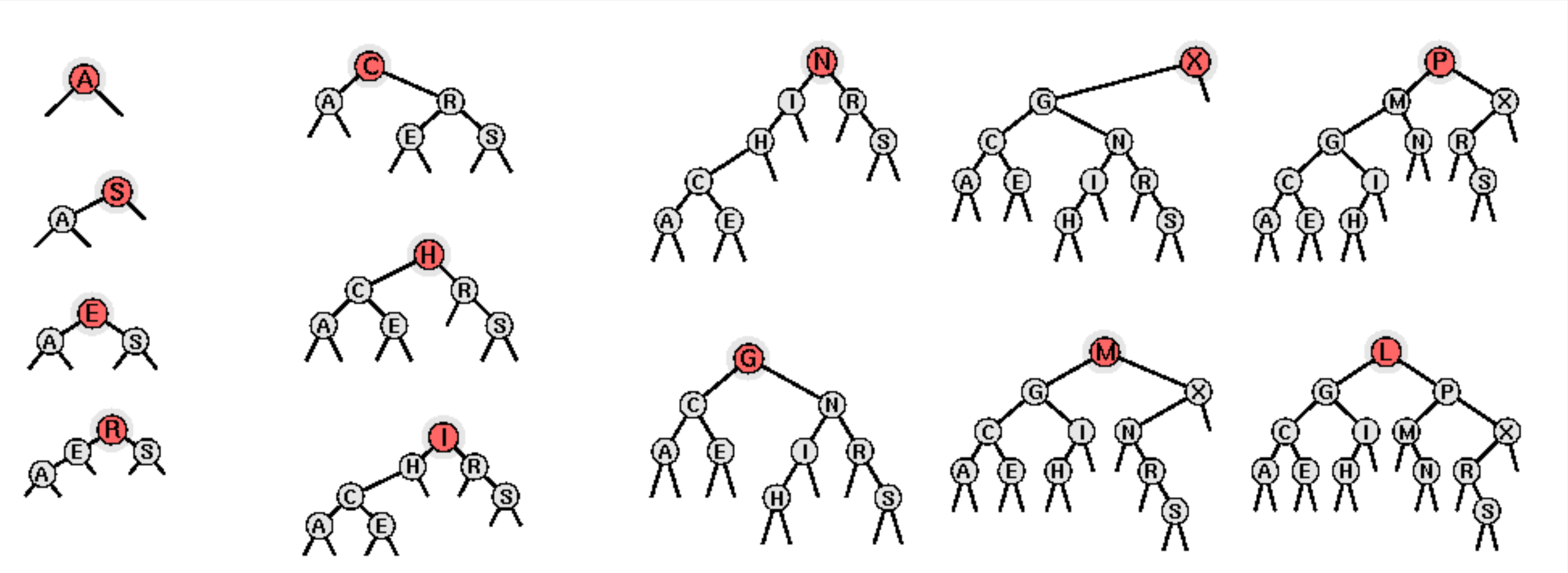
```
private Node putRoot(Node x, Key key, Val val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if (cmp == 0)  x.val = val;
   else if (cmp < 0)
   {  x.left  = putRoot(x.left,  key, val); x = rotR(x);  }
   else if (cmp > 0)
   {  x.right = putRoot(x.right, key, val); x = rotL(x);  }
   return x;
}
```

insert G



Constructing a BST with root insertion

Ex. **A S E R C H I N G X M P L**



Why bother?

- Recently inserted keys are near the top (better for some clients).
- Basis for advanced algorithms.

Randomized BSTs (Roura, 1996)

Intuition. If tree is random, height is logarithmic.
Fact. Each node in a random tree is equally likely to be the root.

Idea. Since new node should be the root with probability 1/(N+1), make it the root (via root insertion) with probability 1/(N+1).

```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if (cmp == 0) { x.val = val; return x; }
   if (StdRandom.bernoulli(1.0 / (x.N + 1.0)))
      return putRoot(h, key, val);
   if       (cmp < 0) x.left  = put(x.left,  key, val);
   else if (cmp > 0) x.right = put(x.right, key, val);
   x.N++;
   return x;
}
```

need to maintain count of nodes in tree rooted at x

Constructing a randomized BST

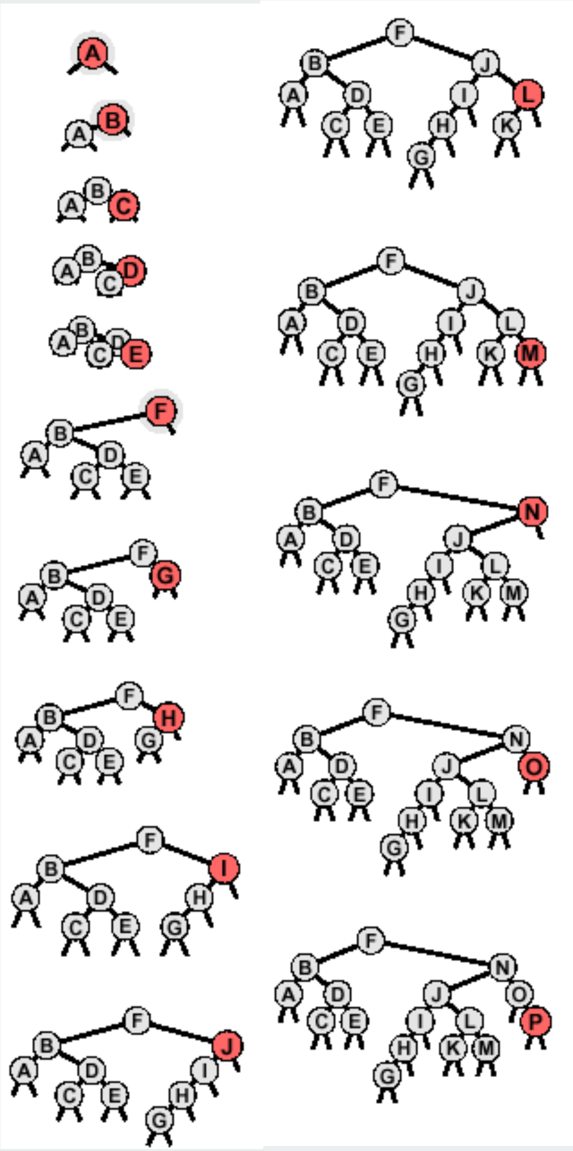
Ex: Insert distinct keys in ascending order.

Surprising fact:

Tree has same shape as if keys were inserted in random order.

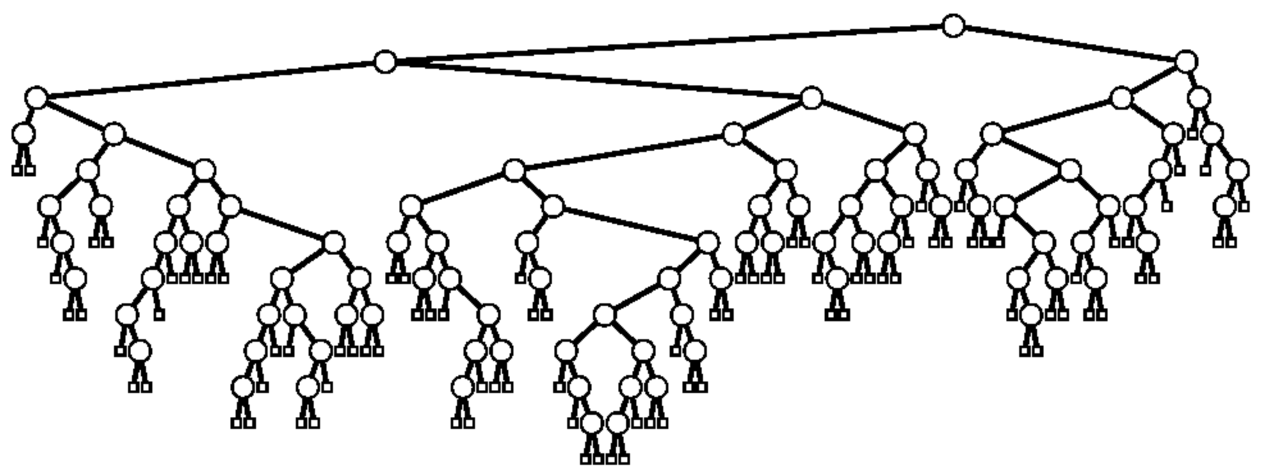
Random trees result from any insert order

Note: to maintain associative array abstraction need to check whether key is in table and replace value without rotations if that is the case.



Randomized BST

Property. Randomized BSTs have the same distribution as BSTs under random insertion order, no matter in what order keys are inserted.



- Expected height is ~6.22 lg N
- Average search cost is ~1.38 lg N.
- Exponentially small chance of bad balance.

Implementation cost. Need to maintain subtree size in each node.

Summary of symbol-table implementations

implementation	guarantee		average case		ordered iteration?	operations on keys
	search	insert	search	insert		
unordered array	N	N	N/2	N/2	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
unordered list	N	N	N/2	N	no	`equals()`
ordered list	N	N	N/2	N/2	yes	`compareTo()`
BST	N	N	1.38 lg N	1.38 lg N	yes	`compareTo()`
randomized BST	7 lg N	7 lg N	1.38 lg N	1.38 lg N	yes	`compareTo()`

Randomized BSTs provide the desired guarantee

probabilistic, with
exponentially small
chance of quadratic time

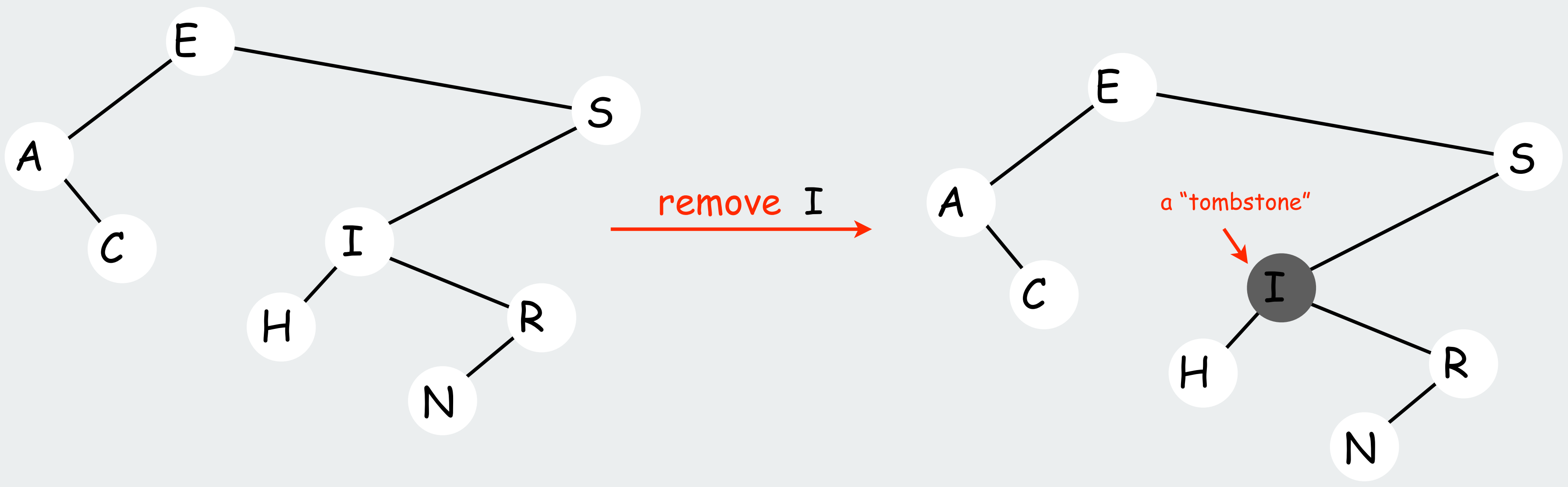
Bonus (next): Randomized BSTs also support delete (!)

- basic implementations
- randomized BSTs
- **deletion in BSTs**

BST delete: lazy approach

To remove a node with a given key

- set its value to **null**
- leave key in tree to guide searches
  [but do not consider it equal to any search key]



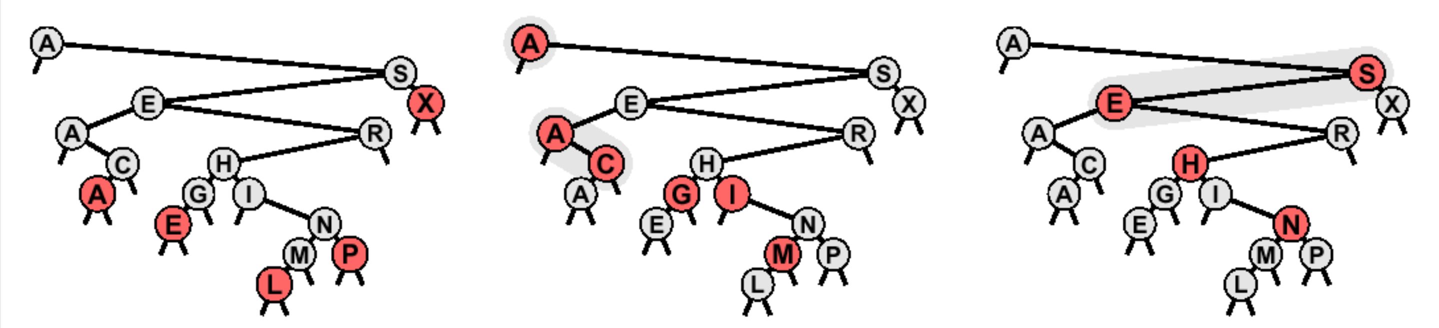
Cost. O(log N') per insert, search, and delete, where N' is the number of elements ever inserted in the BST.

Unsatisfactory solution: Can get overloaded with tombstones.

BST delete: first approach

To remove a node from a BST. [Hibbard, 1960s]

- Zero children: just remove it.
- One child: pass the child up.
- Two children: find the next largest node using right-left*
  swap with next largest
  remove as above.



zero children

one child

two children

Unsatisfactory solution. Not symmetric, code is clumsy.

Surprising consequence. Trees not random (!) ⇒ sqrt(N) per op.

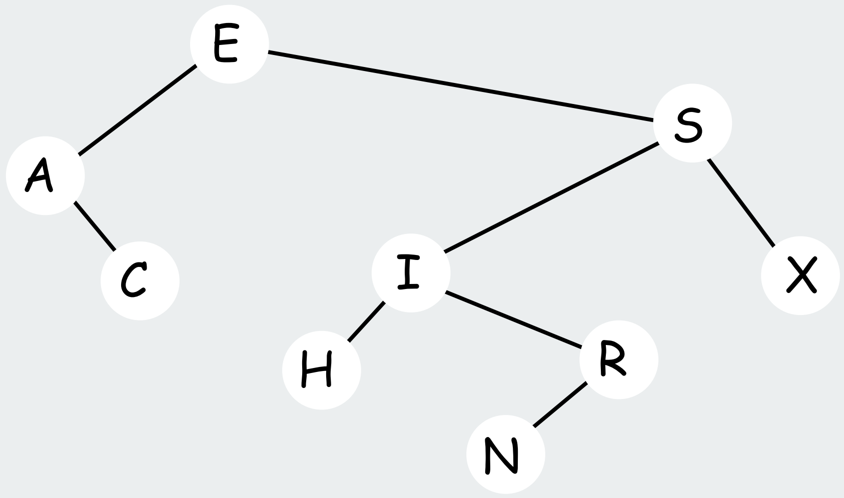
Longstanding open problem: simple and efficient delete for BSTs

Deletion in randomized BSTs

To delete a node containing a given key

- remove the node
- join the two remaining subtrees to make a tree

Ex. Delete S in



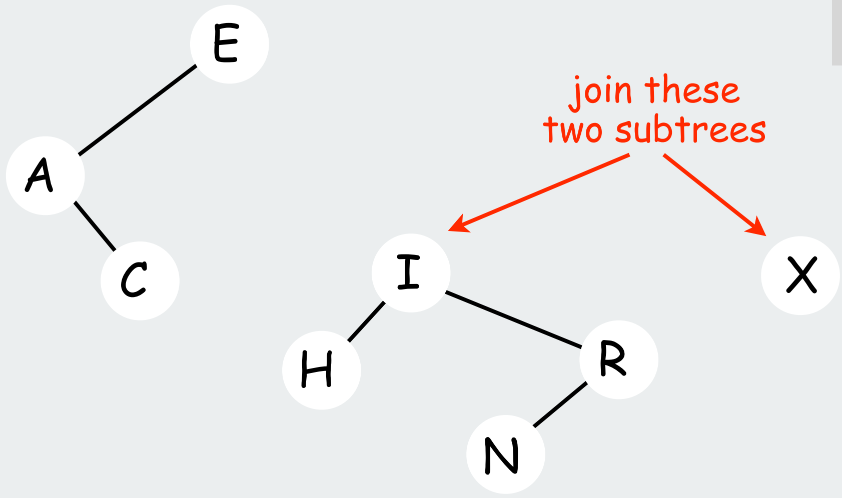
Deletion in randomized BSTs

To delete a node containing a given key

- remove the node
- join its two subtrees

```
private Node remove(Node x, Key key)
{
   if (x == null)
      return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp == 0)
      return join(x.left, x.right);
   else if (cmp < 0)
      x.left  = remove(x.left,  key);
   else if (cmp > 0)
      x.right = remove(x.right, key);
   return x;
}
```

Ex. Delete S in



Join in randomized BSTs

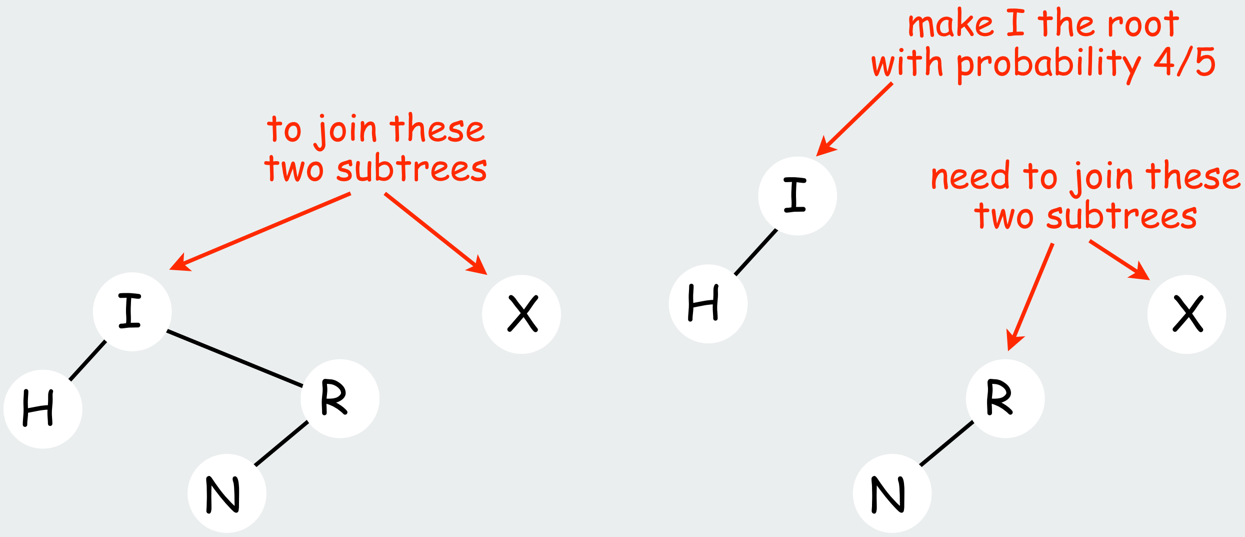
To join two subtrees with all keys in one less than all keys in the other

- maintain counts of nodes in subtrees (L and R)
- with probability L/(L+R)

  make the root of the left the root

  make its left subtree the left subtree of the root

  join its right subtree to R to make the right subtree of the root
- with probability L/(L+R) do the symmetric moves on the right



to join these two subtrees

make I the root with probability 4/5

need to join these two subtrees

Join in randomized BSTs

To join two subtrees with all keys in one less than all keys in the other

- maintain counts of nodes in subtrees (L and R)
- with probability L/(L+R)

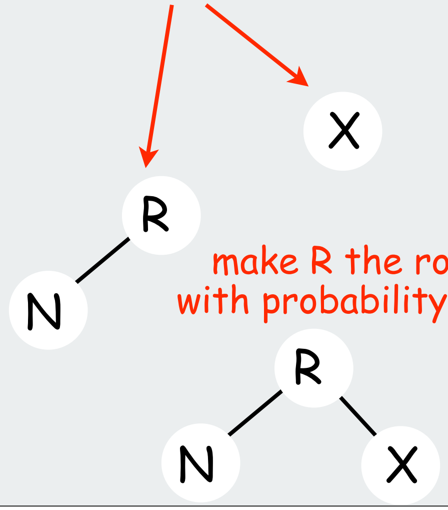
  make the root of the left the root

  make its left subtree the left subtree of the root

  join its right subtree to R to make the right subtree of the root
- with probability L/(L+R) do the symmetric moves on the right

```
private Node join(Node a, Node b)
{
   if (a == null) return a;
   if (b == null) return b;
   int cmp = key.compareTo(x.key);
   if (StdRandom.bernoulli((double)*a.N / (a.N + b.N))
     { a.right = join(a.right, b); return a; }
   else
     { b.left  = join(a, b.left ); return b; }
}
```

to join these two subtrees



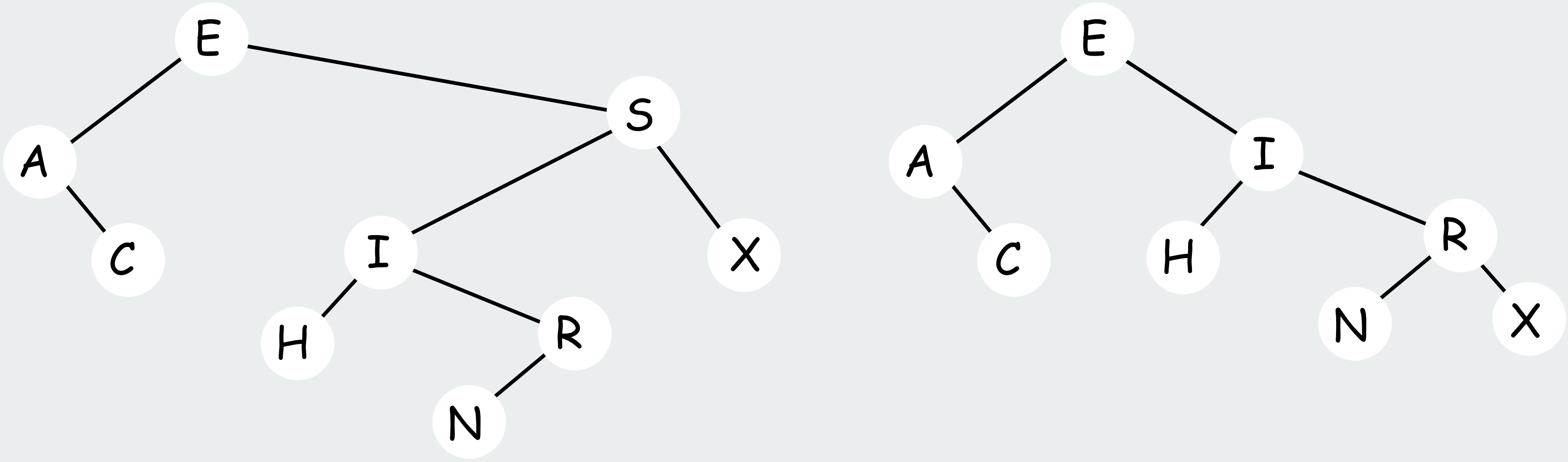
make R the root with probability 2/3

Deletion in randomized BSTs

To delete a node containing a given key

- remove the node
- join its two subtrees

Ex. Delete S in



Theorem. Tree still random after delete (!)

Bottom line. Logarithmic guarantee for search/insert/delete

Summary of symbol-table implementations

implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes

Randomized BSTs provide the desired guarantees

↑
probabilistic, with
exponentially small
chance of error

Next lecture: Can we do better?

Balanced Trees

- 2-3-4 trees
- red-black trees
- B-trees

References:

Algorithms in Java, Chapter 13

http://www.cs.princeton.edu/introalgsds/44balanced

Symbol Table Review

Symbol table: key-value pair abstraction.

- Insert a value with specified key.
- Search for value given key.
- Delete value with given key.

Randomized BST.

- Guarantee of ~c lg N time per operation (probabilistic).
- Need subtree count in each node.
- Need random numbers for each insert/delete op.

This lecture. 2-3-4 trees, left-leaning red-black trees, B-trees.

↑

new for Fall 2007!

Summary of symbol-table implementations

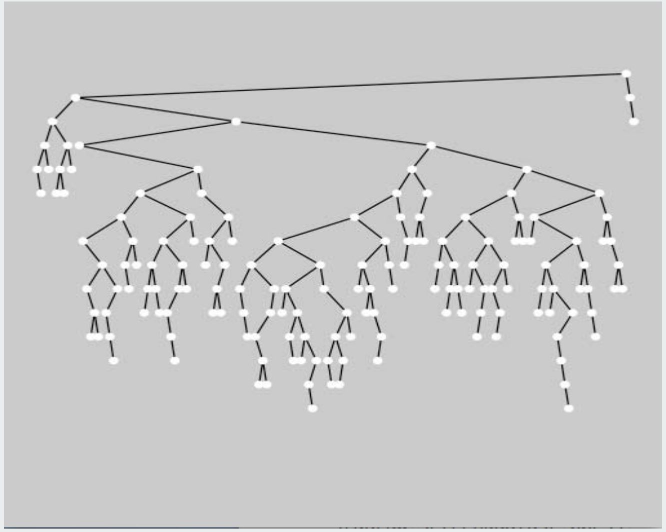
implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.39 lg N	1.39 lg N	1.39 lg N	yes

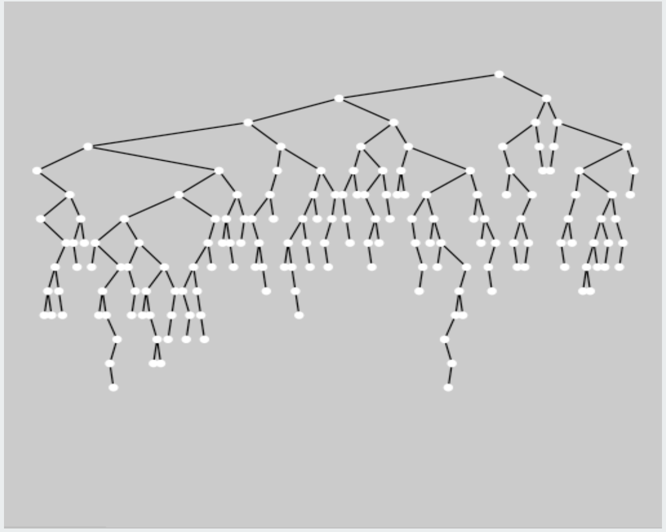
Randomized BSTs provide the desired guarantees

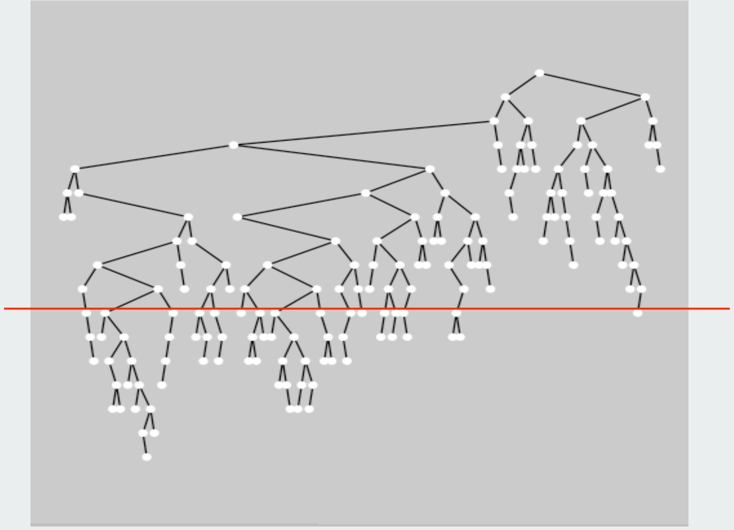
↑
probabilistic, with
exponentially small
chance of quadratic time

This lecture: Can we do better?

Typical random BSTs







$N = 250$

$\lg N \approx 8$

$1.39 \lg N \approx 11$

average node depth

- 2-3-4 trees
- red-black trees
- B-trees

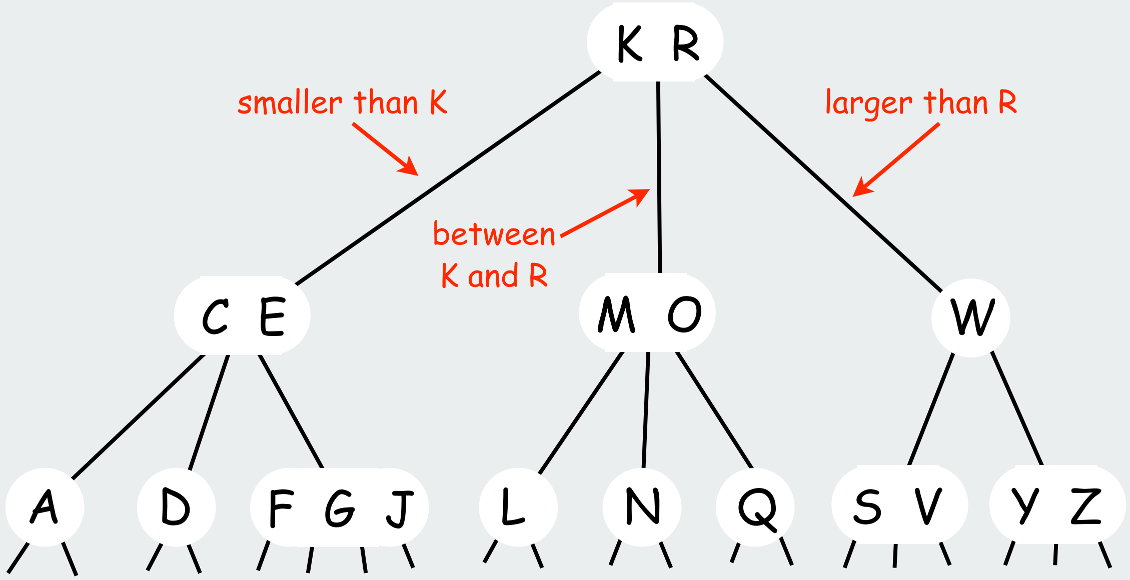
2-3-4 Tree

2-3-4 tree. Generalize node to allow multiple keys; keep tree balanced.

Perfect balance. Every path from root to leaf has same length.

Allow 1, 2, or 3 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.
- 4-node: three keys, four children.

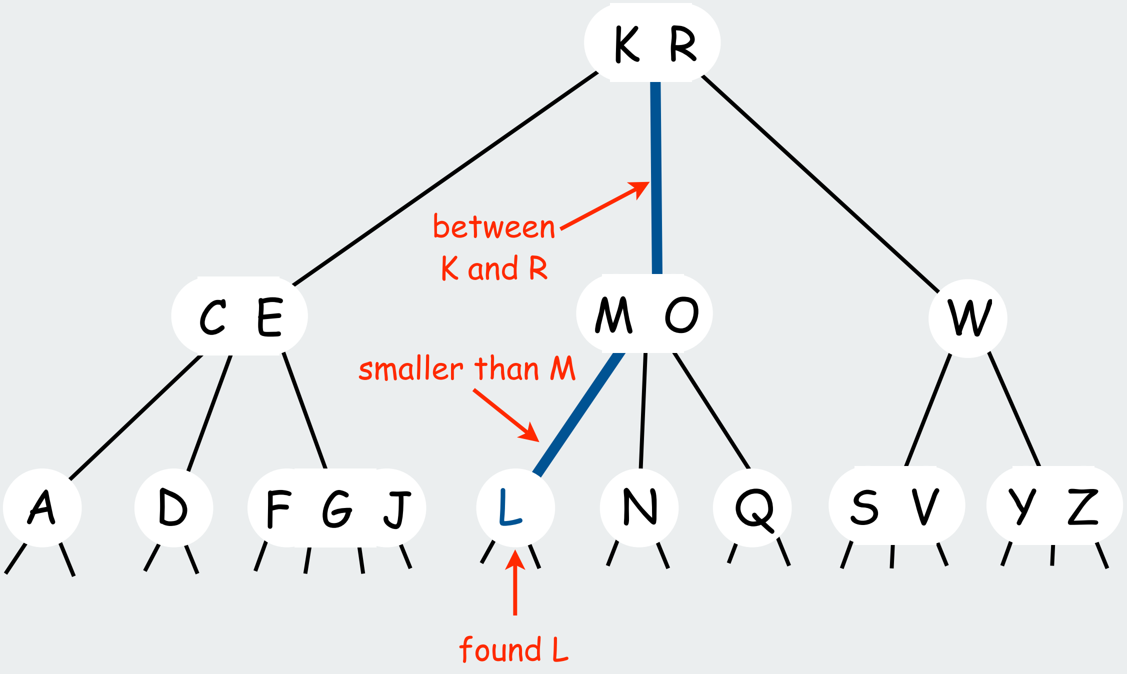


Searching in a 2-3-4 Tree

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

Ex. Search for L



between
K and R

smaller than M

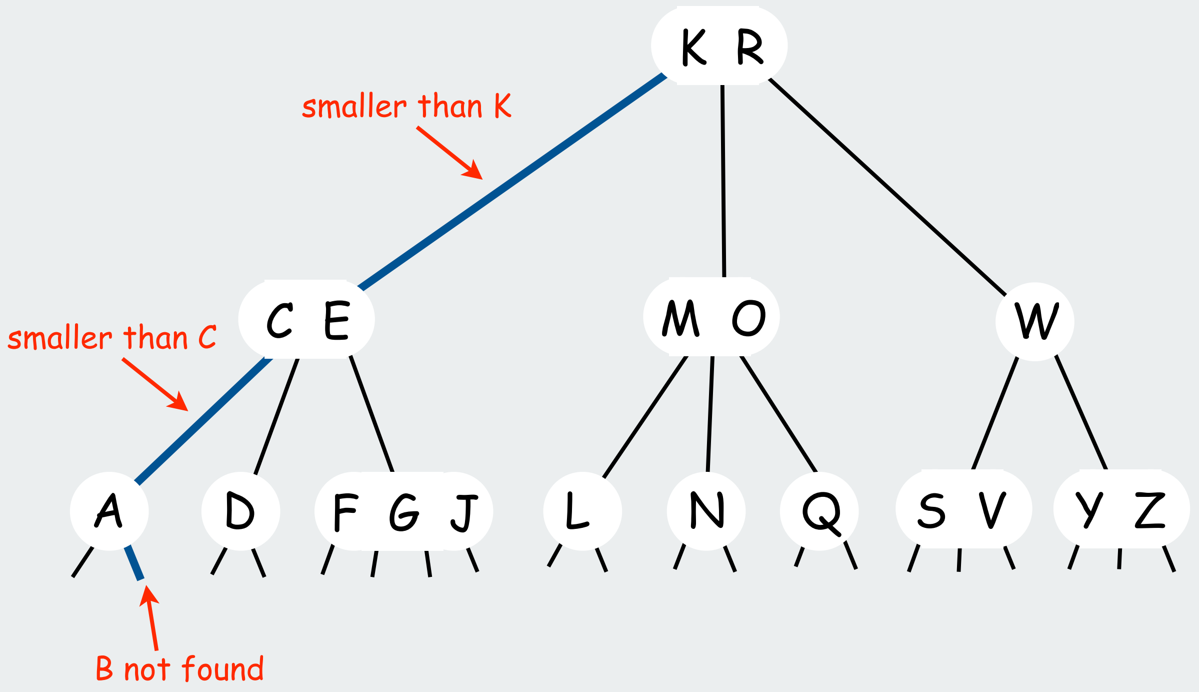
found L

Insertion in a 2-3-4 Tree

Insert.

- Search to bottom for key.

Ex. Insert B

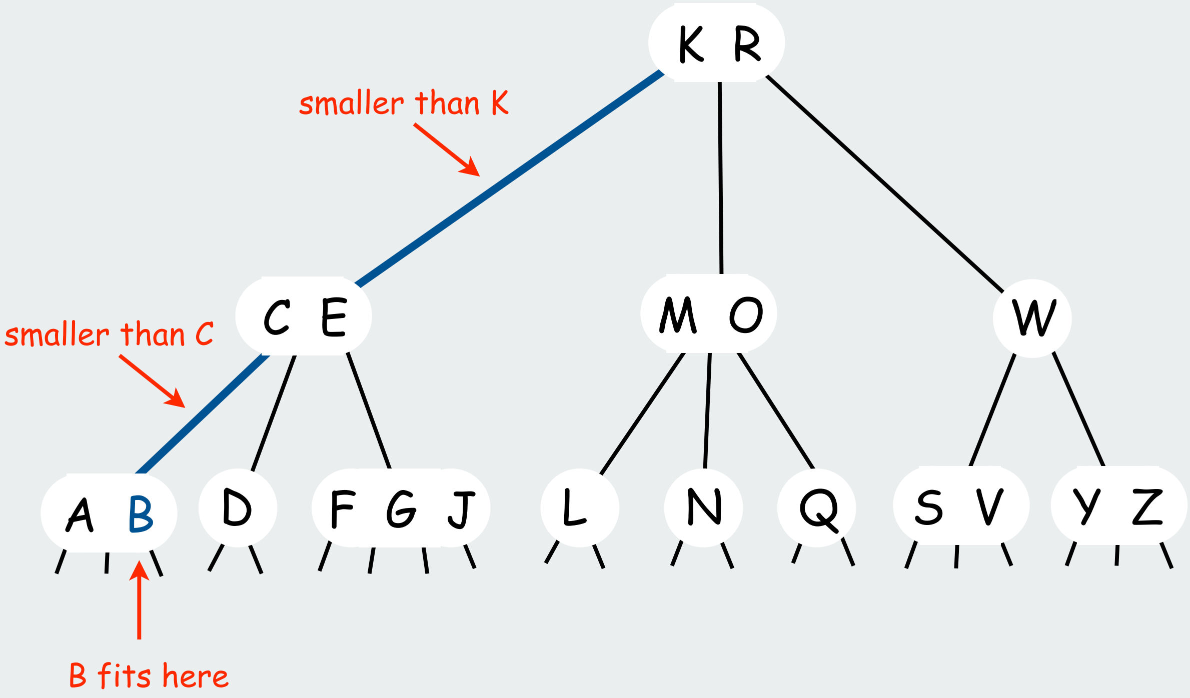


Insertion in a 2-3-4 Tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node.

Ex. Insert B

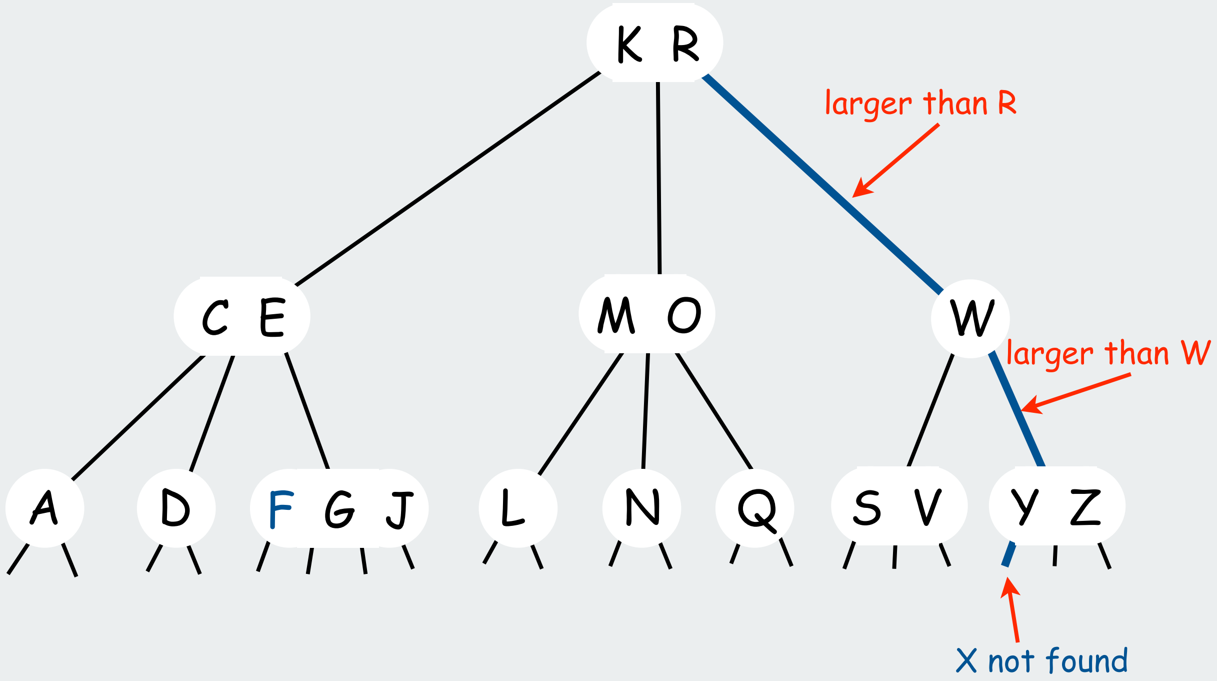


Insertion in a 2-3-4 Tree

Insert.

- Search to bottom for key.

Ex. Insert X

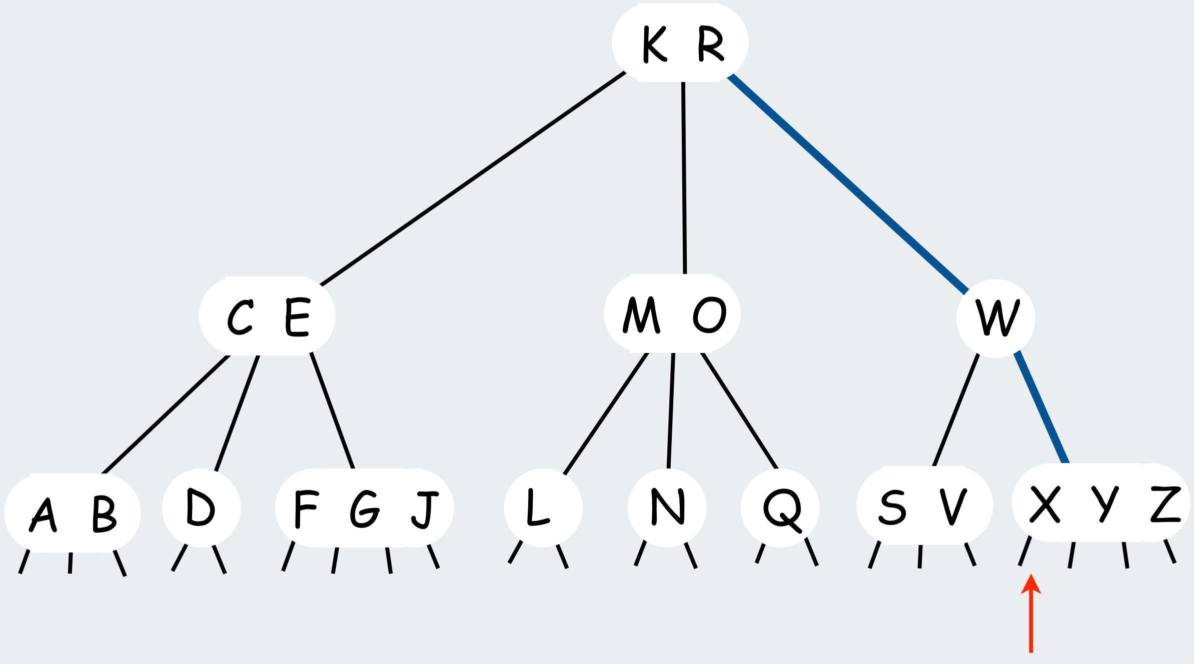


Insertion in a 2-3-4 Tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node.
- 3-node at bottom: convert to 4-node.

Ex. Insert X



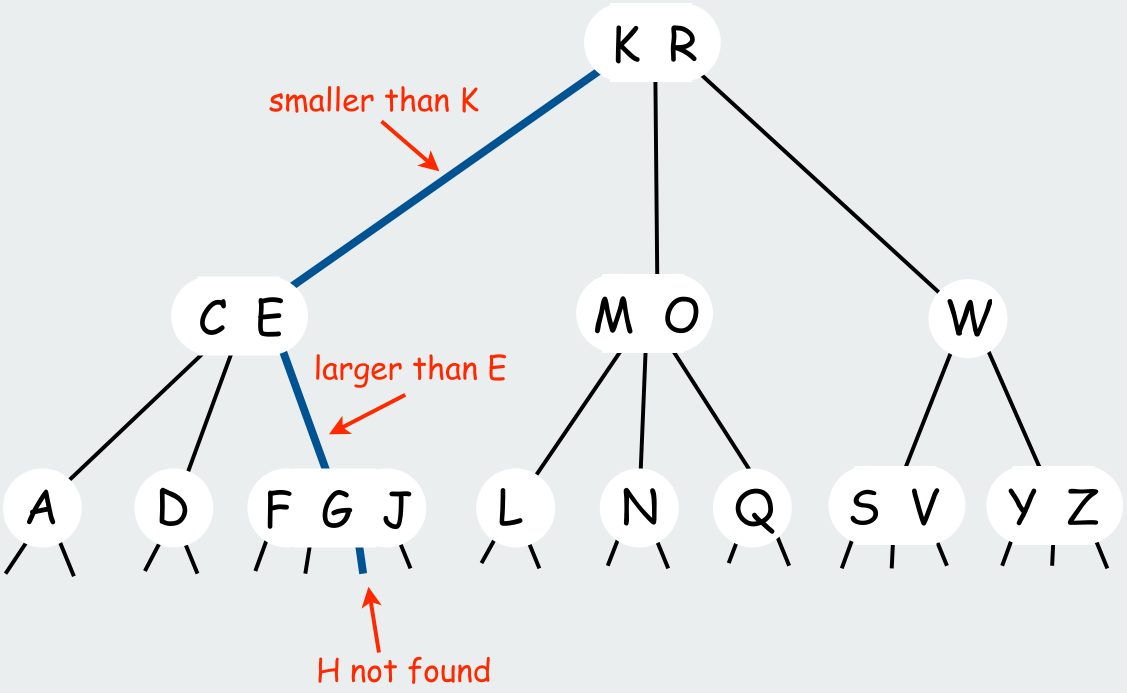
X fits here

Insertion in a 2-3-4 Tree

Insert.

- Search to bottom for key.

Ex. Insert H

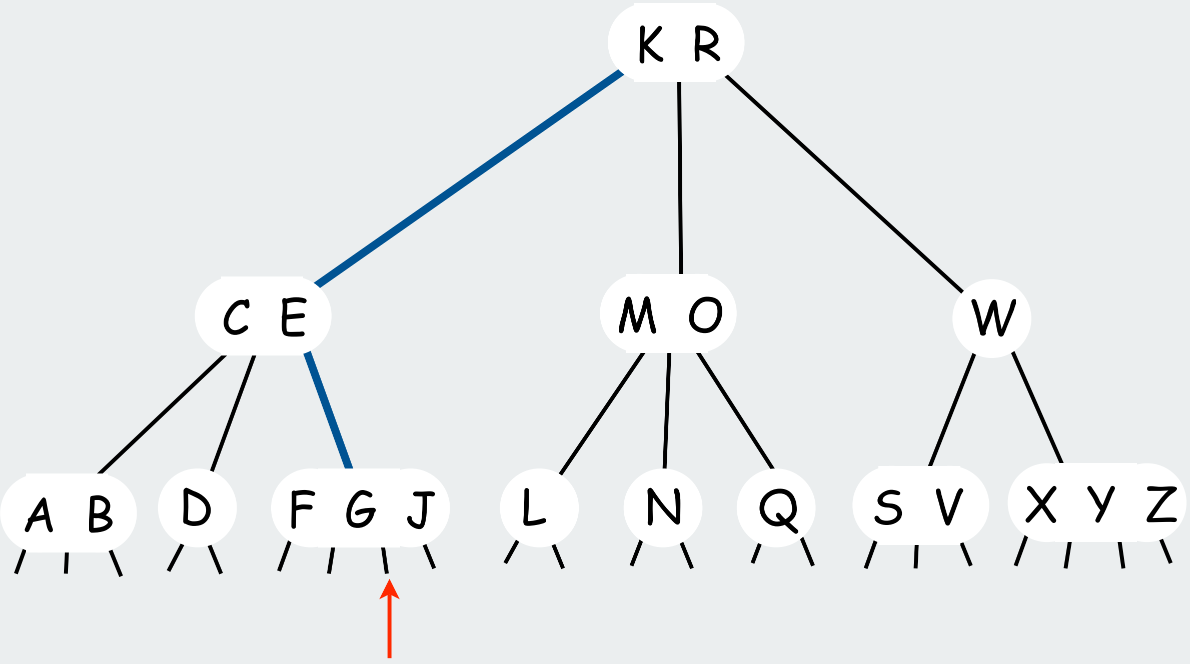


Insertion in a 2-3-4 Tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node.
- 3-node at bottom: convert to 4-node.
- 4-node at bottom: ??

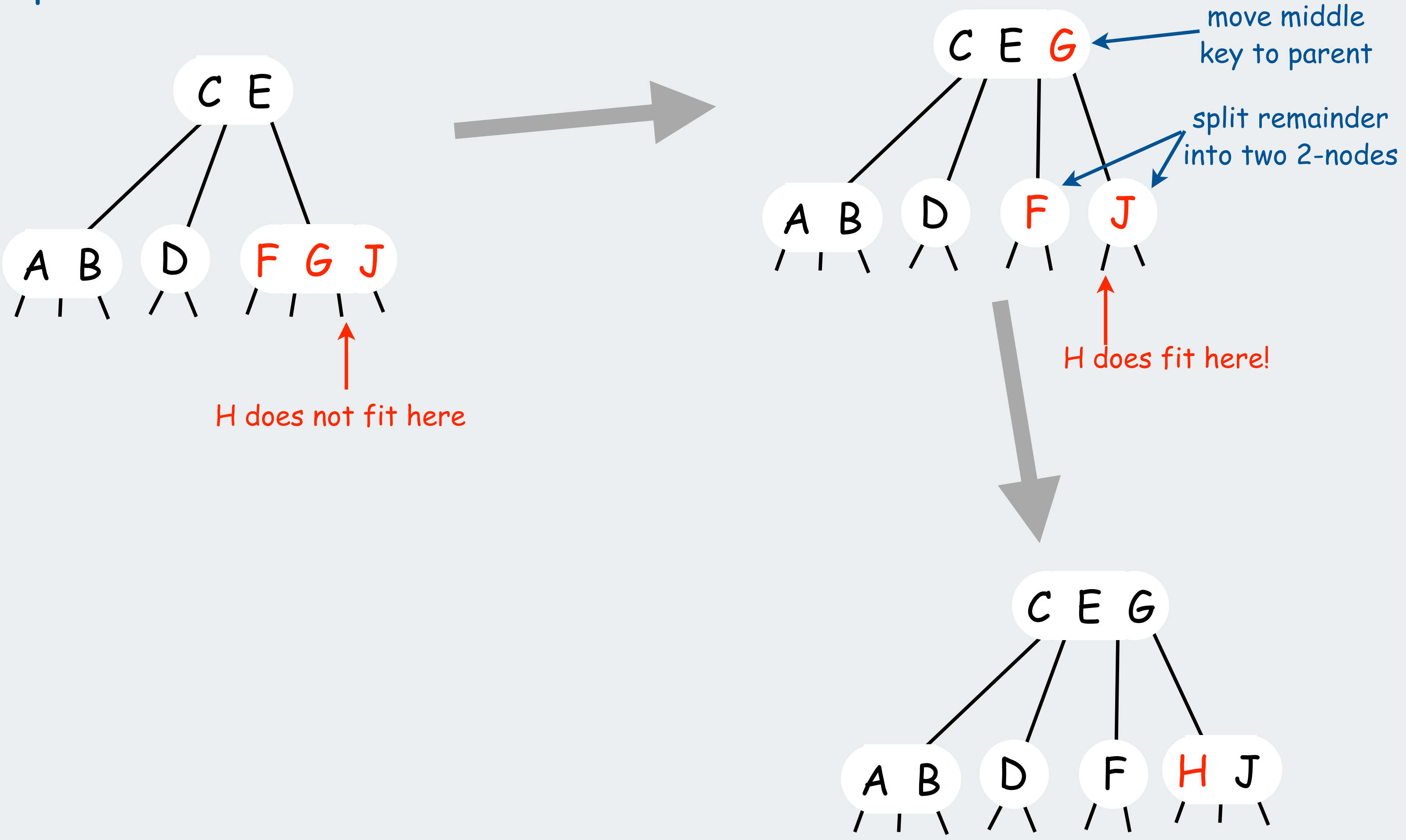
Ex. Insert H



H does not fit here!

Splitting a 4-node in a 2-3-4 tree

Idea: split the 4-node to make room



H does not fit here

move middle key to parent

split remainder into two 2-nodes

H does fit here!

Problem: Doesn't work if parent is a 4-node

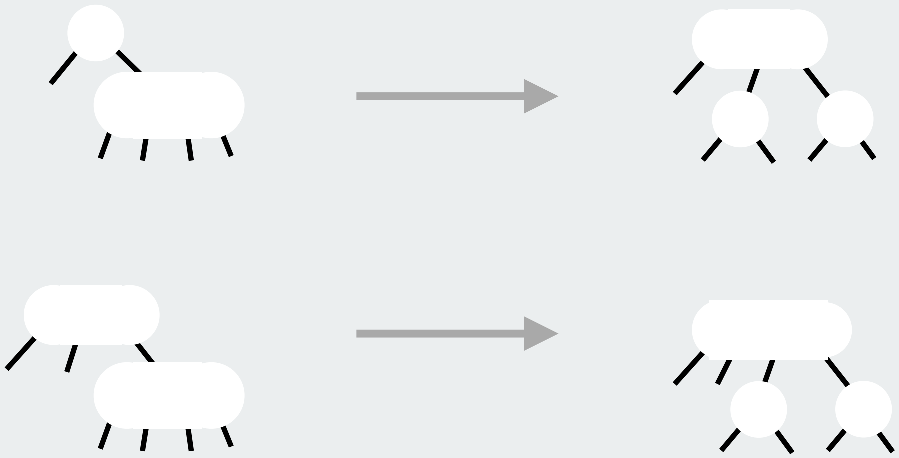
Solution 1: Split the parent (and continue splitting up while necessary).

Solution 2: Split 4-nodes on the way down.

Splitting 4-nodes in a 2-3-4 tree

Idea: split 4-nodes on the way down the tree.

- Ensures that most recently seen node is not a 4-node.
- Transformations to split 4-nodes:



local transformations
that work anywhere
in the tree

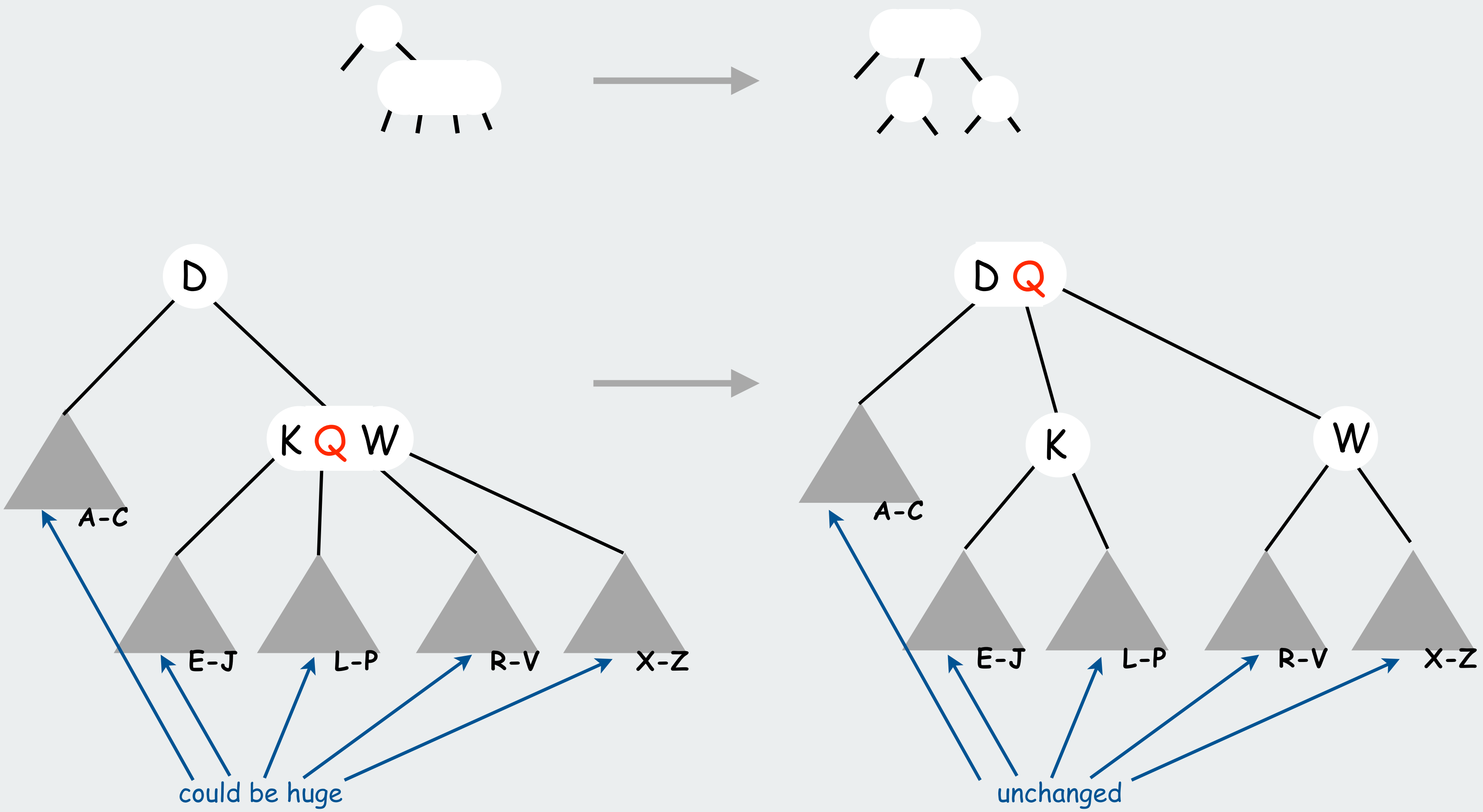
Invariant. Current node is not a 4-node.

Consequences

- 4-node below a 4-node case never happens
- insertion at bottom node is easy since it's not a 4-node.

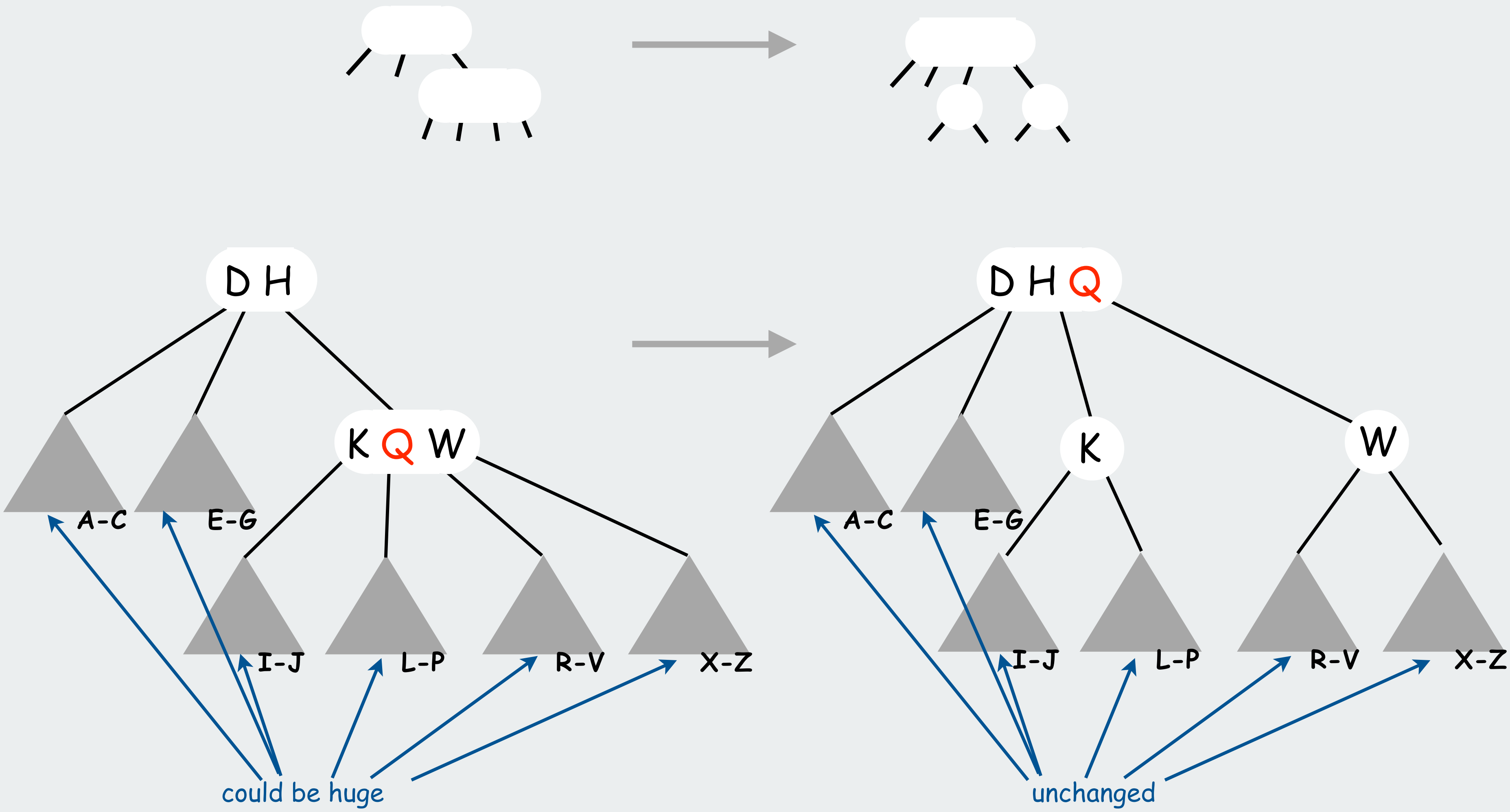
Splitting a 4-node below a 2-node in a 2-3-4 tree

A local transformation that works anywhere in the tree



Splitting a 4-node below a 3-node in a 2-3-4 tree

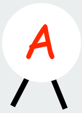
A local transformation that works anywhere in the tree



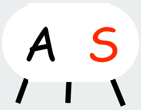
Growth of a 2-3-4 tree

Tree grows up from the bottom

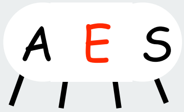
insert A



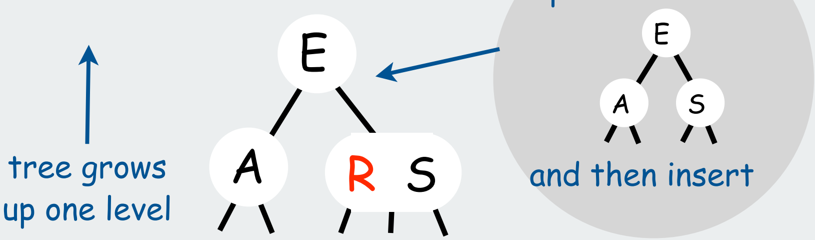
insert S



insert E



insert R

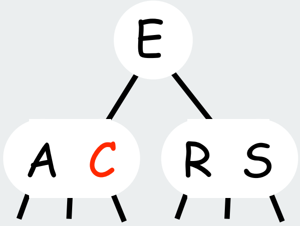


split 4-node to

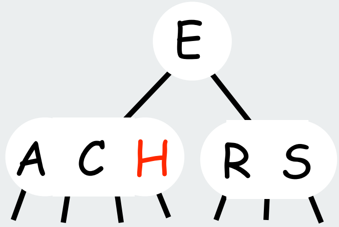
and then insert

tree grows up one level

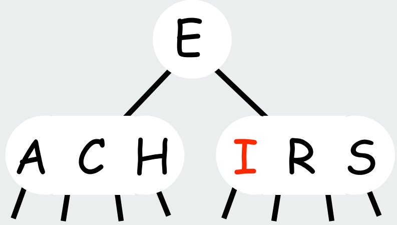
insert C



insert H

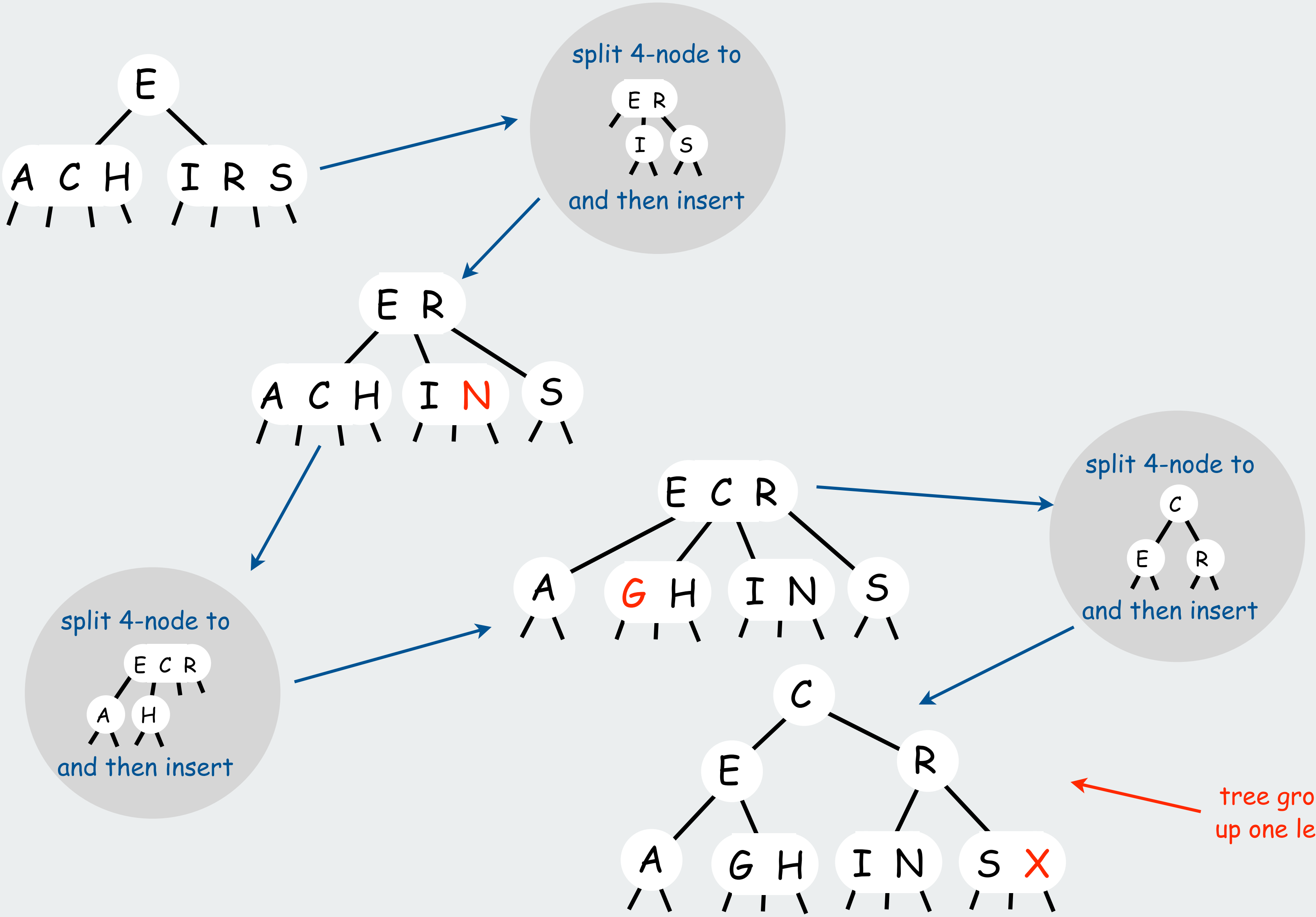


insert I



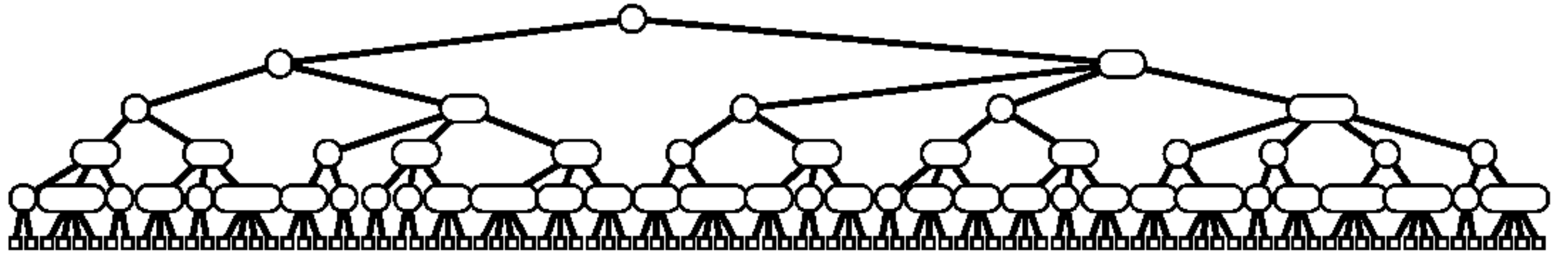
Growth of a 2-3-4 tree (continued)

Tree grows up from the bottom



Balance in 2-3-4 trees

Key property: All paths from root to leaf have same length.



Tree height.

- Worst case: lg N [all 2-nodes]
- Best case: $\log_4 N = 1/2 \lg N$ [all 4-nodes]
- Between 10 and 20 for a million nodes.
- Between 15 and 30 for a billion nodes.

2-3-4 Tree: Implementation?

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Implementation of `getChild()` involves multiple compares.
- Large number of cases for `split()`, `make3Node()`, and `make4Node()`.

```
private void insert(Key key, Val val)
{
   Node x = root;
   while (x.getChild(key) != null)
   {
      x = x.getChild(key);
      if (x.is4Node()) x.split();
   }
   if      (x.is2Node()) x.make3Node(key, val);
   else if (x.is3Node()) x.make4Node(key, val);
}
```

fantasy code

Bottom line: could do it, but stay tuned for an easier way.

Summary of symbol-table implementations

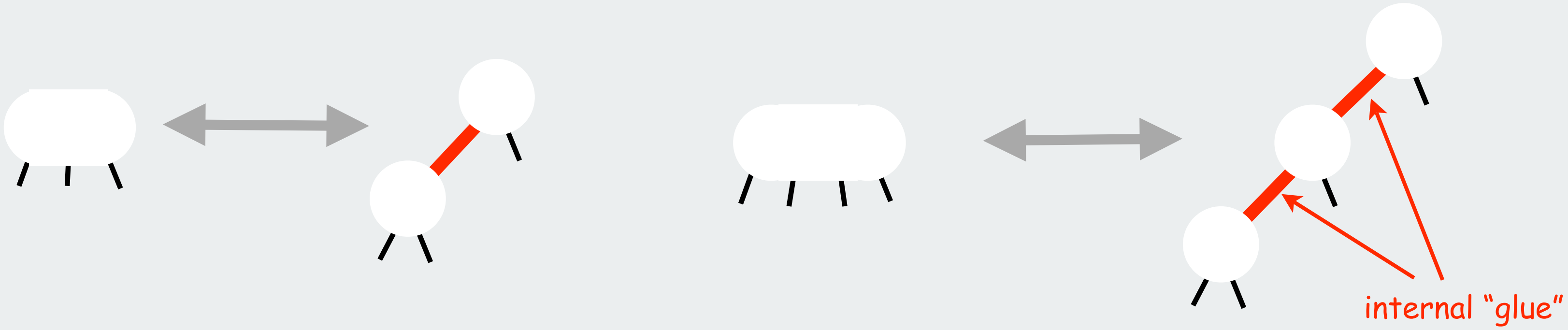
implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes
2-3-4 tree	c lg N	c lg N		c lg N	c lg N		yes

constants depend upon implementation

- 2-3-4 trees
- **red-black trees**
- B-trees

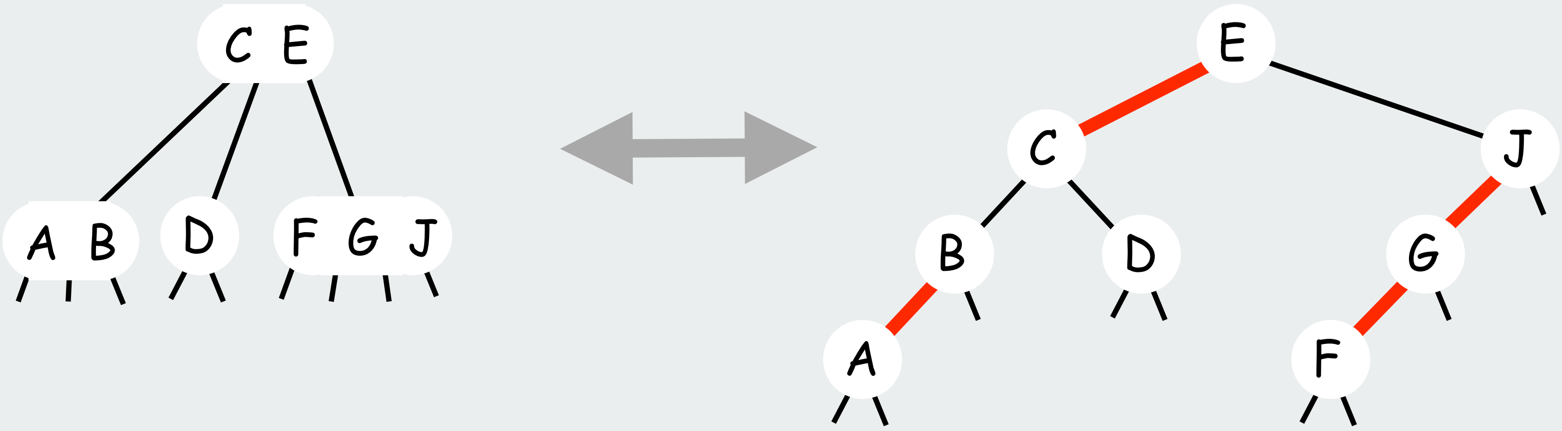
Left-leaning red-black trees (Guibas-Sedgewick, 1979 and Sedgewick, 2007)

1. Represent 2-3-4 tree as a BST.
2. Use "internal" left-leaning edges for 3- and 4- nodes.



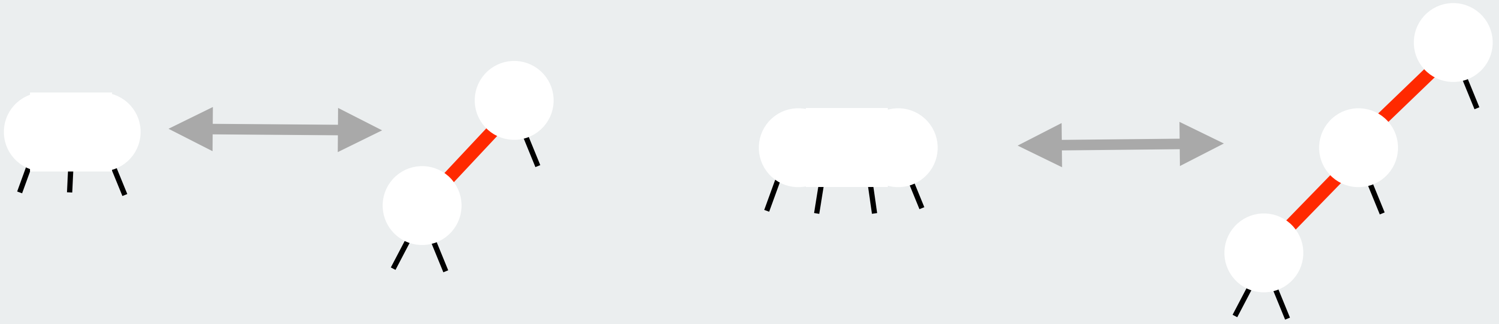
Key Properties

- elementary BST search works
- 1-1 correspondence between 2-3-4 and left-leaning red-black trees



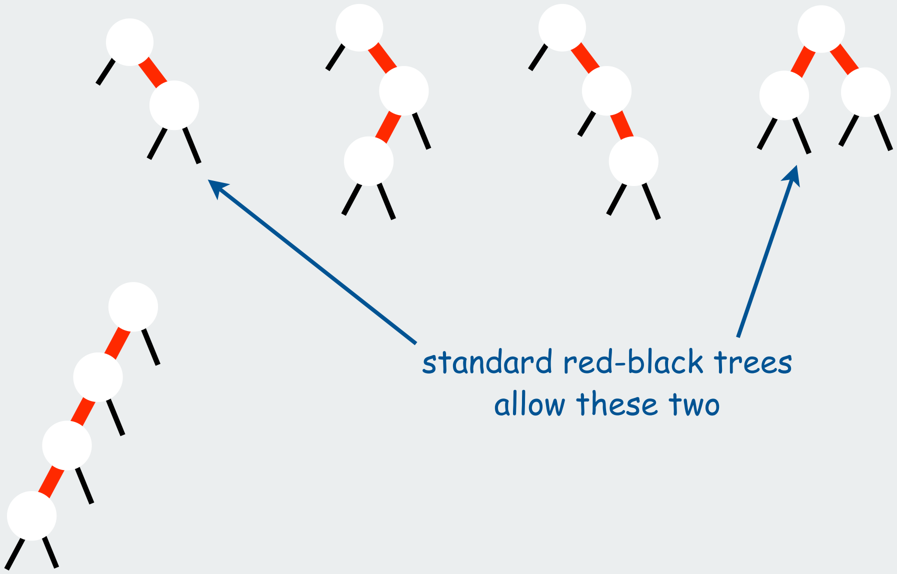
Left-leaning red-black trees

1. Represent 2-3-4 tree as a BST.
2. Use "internal" left-leaning edges for 3- and 4- nodes.



Disallowed:

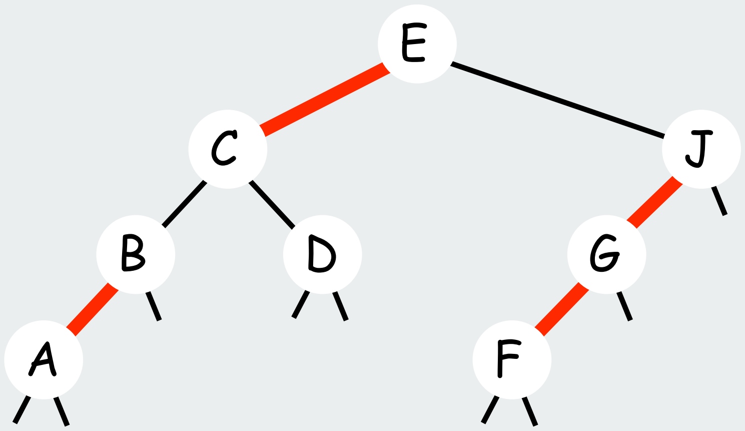
- right-leaning red edges
- three red edges in a row



standard red-black trees
allow these two

Search implementation for red-black trees

```
public Val get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if (cmp == 0)     return x.val;
      else if (cmp < 0) x = x.left;
      else if (cmp > 0) x = x.right;
   }
   return null;
}
```



Search code is the same as elementary BST (ignores the color)
[runs faster because of better balance in tree]

Note: iterator code is also the same.

Insert implementation for red-black trees (skeleton)

```
public class BST<Key extends Comparable<Key>, Value>
             implements Iterable<Key>
{
    private static final boolean RED   = true;
    private static final boolean BLACK = false;
    private Node root;

    private class Node
    {
        Key key;
        Value val;
        Node left, right;
        boolean color;
        Node(Key key, Value val, boolean color)
        {
            this.key   = key;
            this.val = val;
            this.color = color;
        }
    }


   public void put(Key key, Value val)
   {
      root = put(root, key, val);
      root.color = BLACK;
   }
}
```

color of incoming link

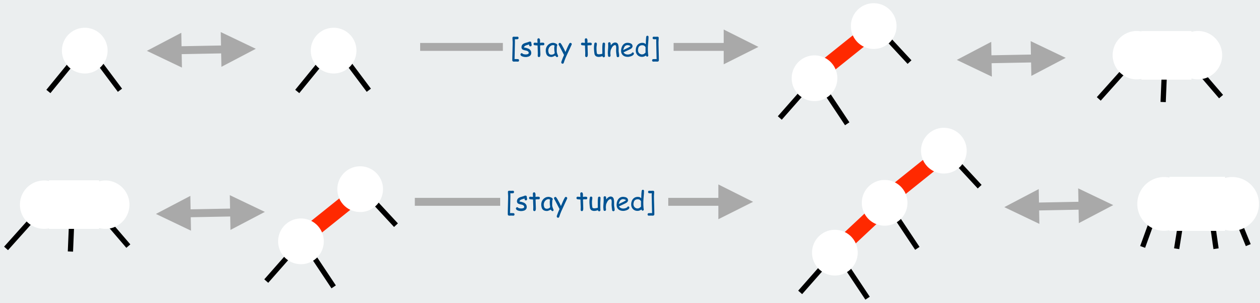
helper method to test node color

```
private boolean isRed(Node x)
{
   if (x == null) return false;
   return (x.color == RED);
}
```

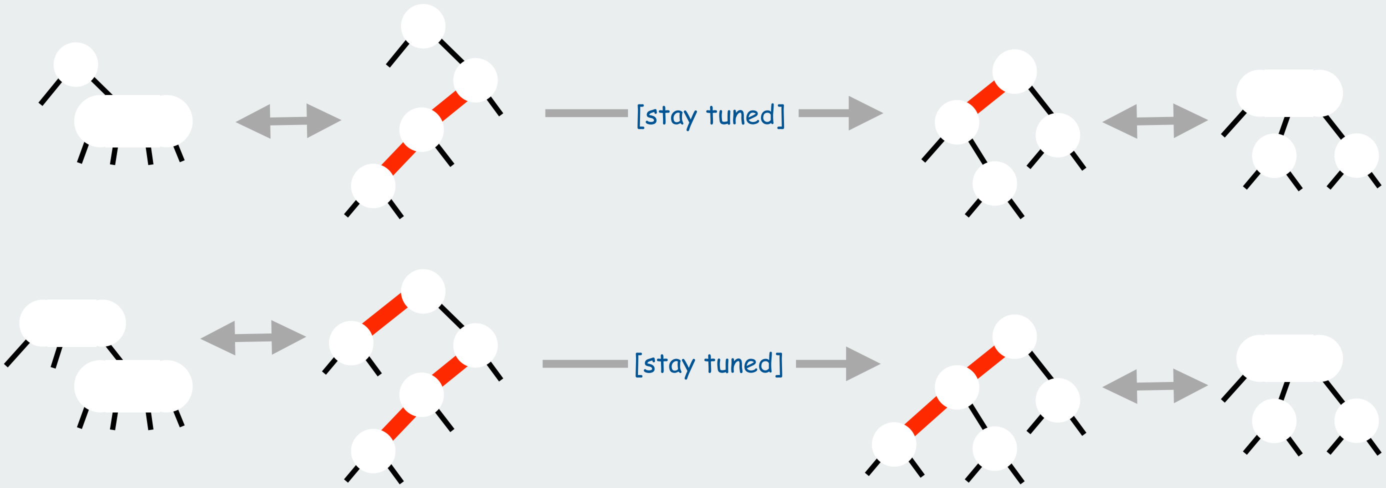
Insert implementation for left-leaning red-black trees (strategy)

Basic idea: maintain 1-1 correspondence with 2-3-4 trees

1. If key found on recursive search reset value, as usual
2. If key not found insert a new red node at the bottom



3. Split 4-nodes on the way DOWN the tree.

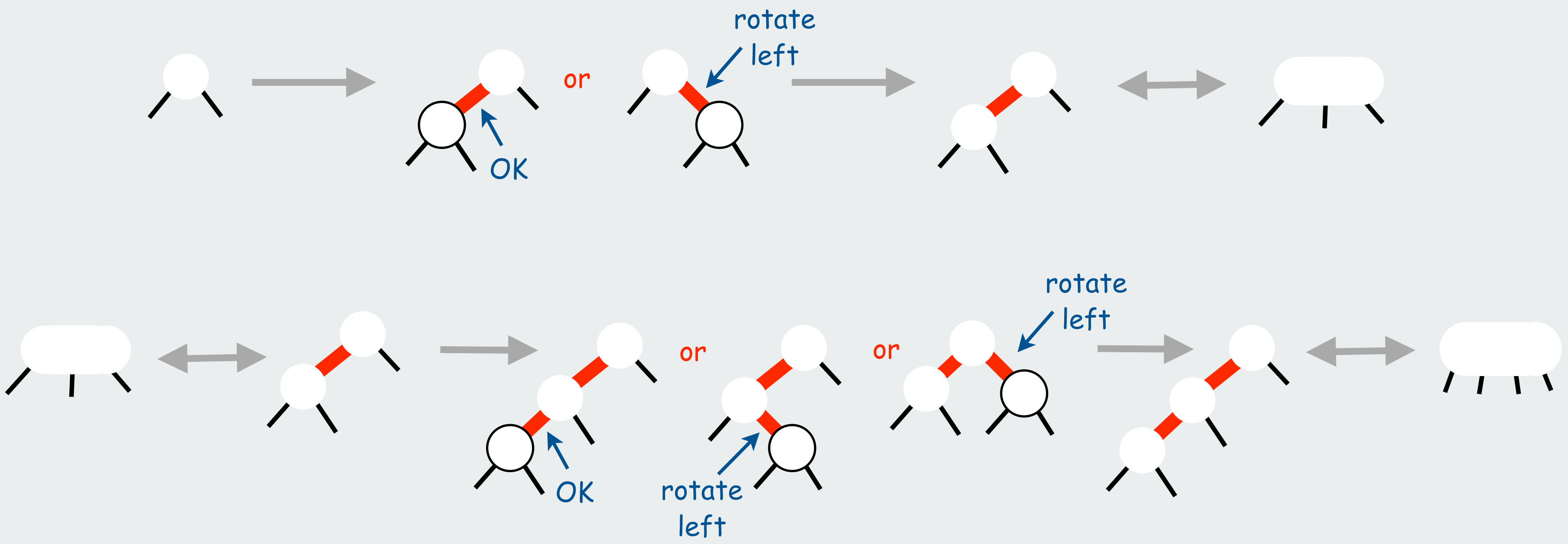


Inserting a new node at the bottom in a LLRB tree

Maintain 1-1 correspondence with 2-3-4 trees

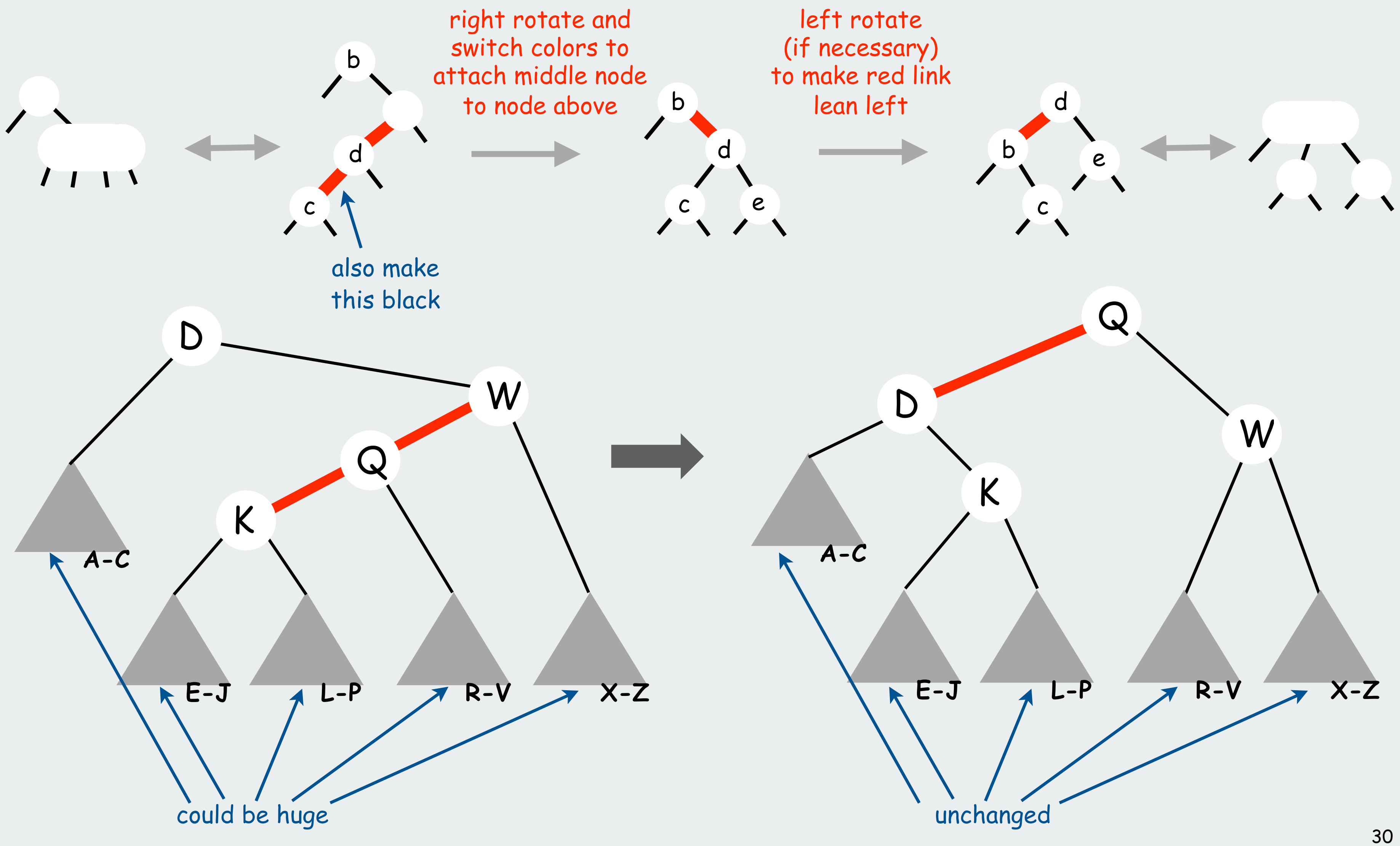
1. Add new node as usual, with red link to glue it to node above

2. Rotate left if necessary to make link lean left



Splitting a 4-node below a 2-node in a left-leaning red-black tree

Maintain correspondence with 2-3-4 trees



right rotate and switch colors to attach middle node to node above

left rotate (if necessary) to make red link lean left

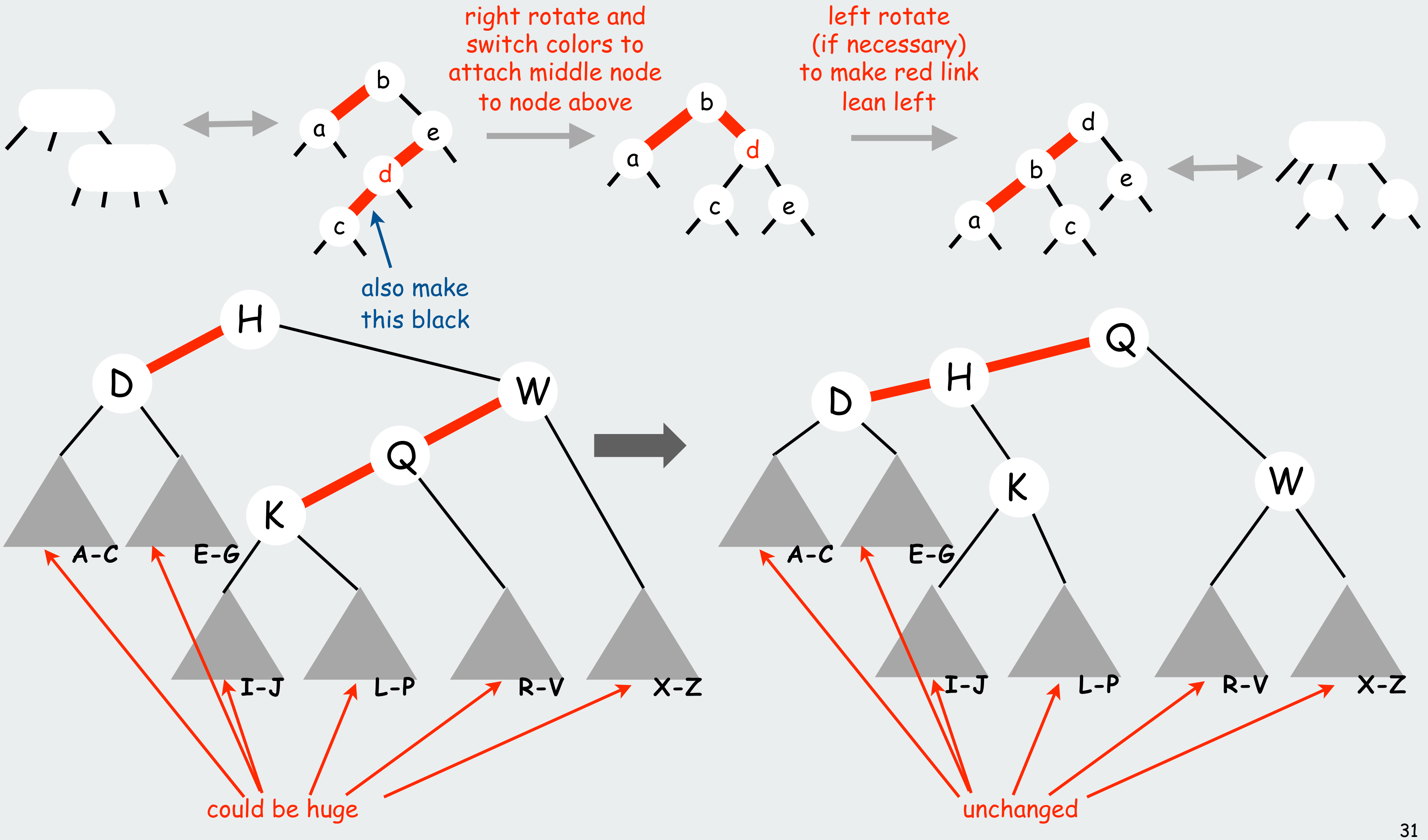
also make this black

could be huge

unchanged

Splitting a 4-node below a 3-node in a left-leaning red-black tree

Maintain correspondence with 2-3-4 trees



right rotate and switch colors to attach middle node to node above

left rotate (if necessary) to make red link lean left

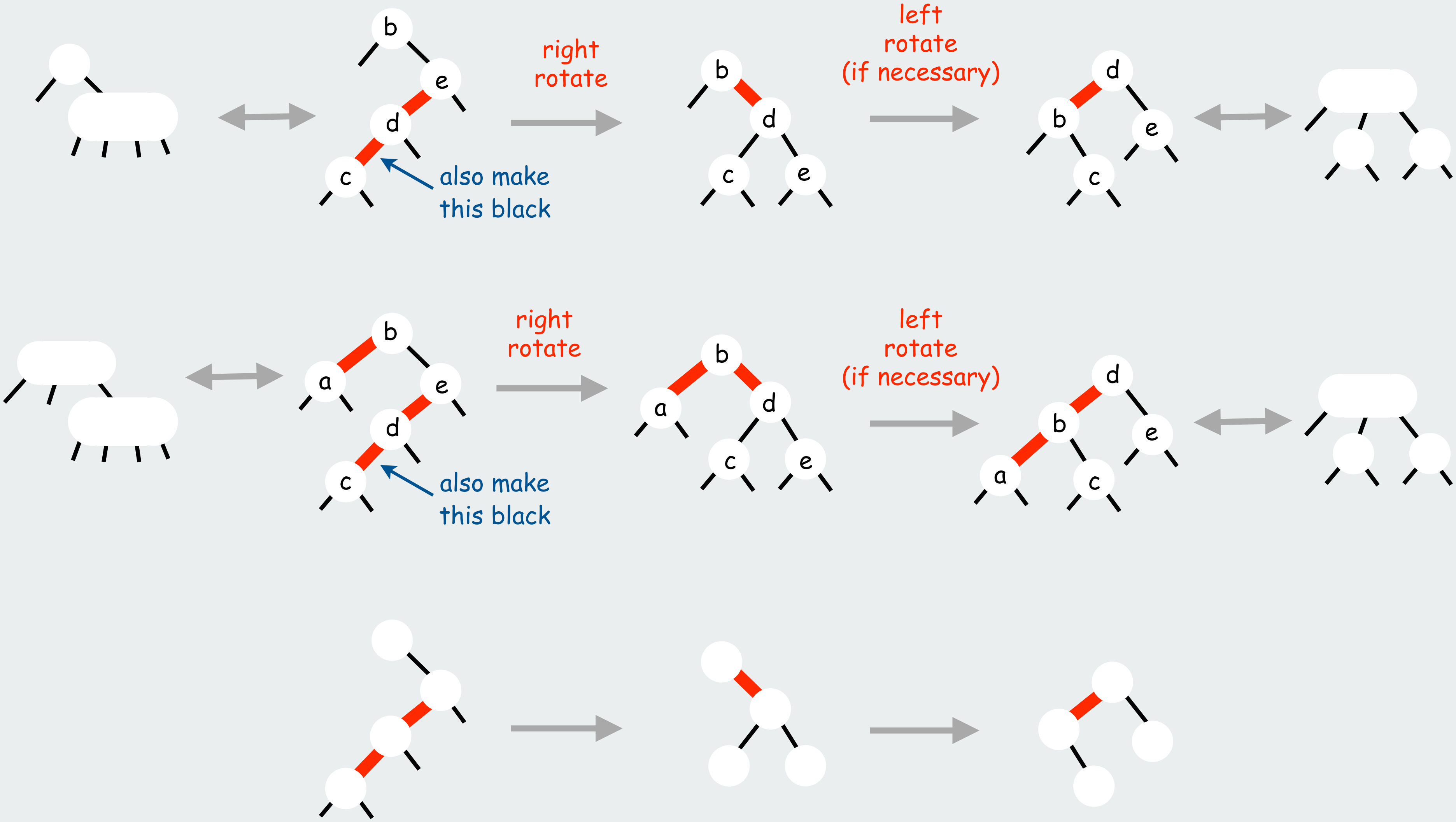
also make this black

could be huge

unchanged

Splitting 4-nodes a left-leaning red-black tree

The two transformations are the same



right rotate

left rotate (if necessary)

also make this black

right rotate

left rotate (if necessary)

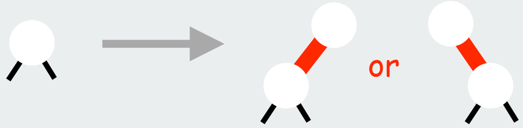
also make this black

Insert implementation for left-leaning red-black trees (strategy revisited)

Basic idea: maintain 1-1 correspondence with 2-3-4 trees

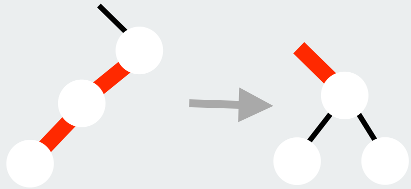
Search as usual

- if key found reset value, as usual
- if key not found insert a new red node at the bottom
  [might be right-leaning red link]



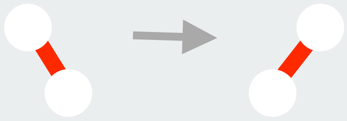
Split 4-nodes on the way DOWN the tree.

- right-rotate and flip color
- might leave right-leaning link higher up in the tree



NEW TRICK: enforce left-leaning condition on the way UP the tree.

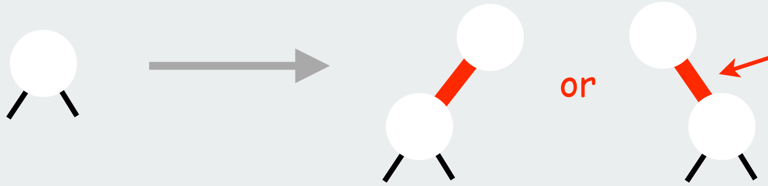
- left-rotate any right-leaning link on search path
- trivial with recursion (do it after recursive calls)
- no other right-leaning links elsewhere



Note: nonrecursive top-down implementation possible, but requires keeping track of great-grandparent on search path (!) and lots of cases.

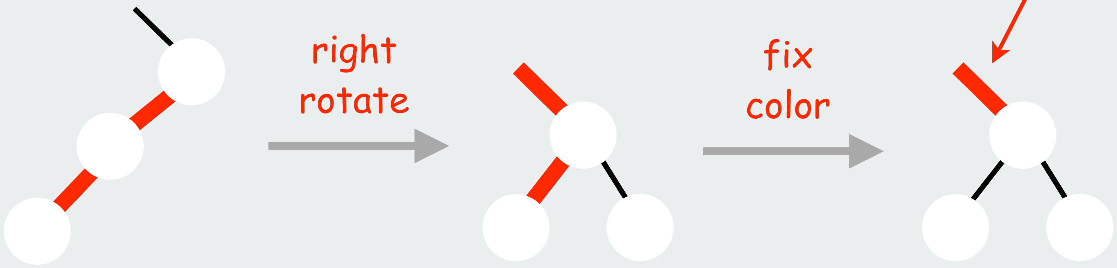
Insert implementation for left-leaning red-black trees (basic operations)

Insert a new node at bottom

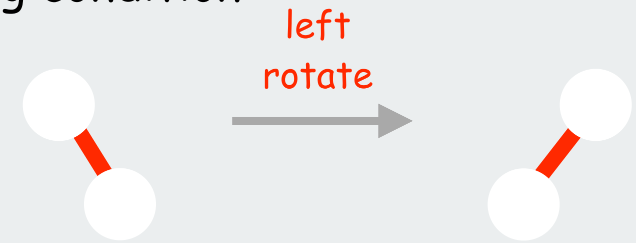


Key point: may leave right-leaning link to be fixed later

Split a 4-node



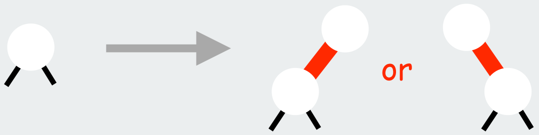
Enforce left-leaning condition



Insert implementation for left-leaning red-black trees (code for basic operations)

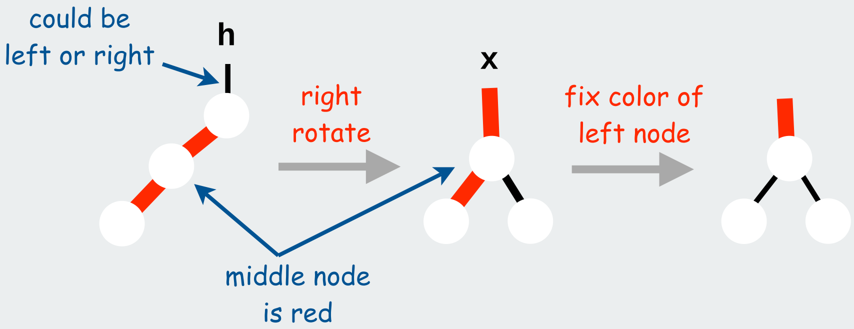
Insert a new node at bottom

```
if (h == null)
      return new Node(key, value, RED);
```



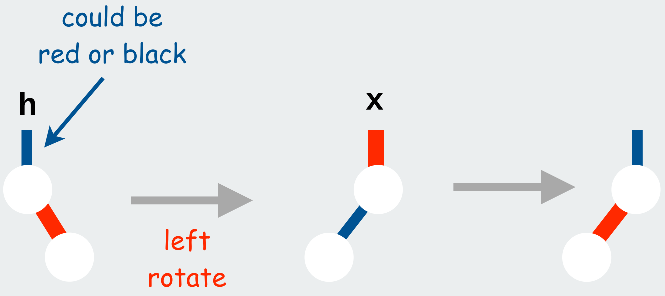
Split a 4-node

```
private Node splitFourNode(Node h)
{
   x = rotR(h);
   x.left.color  = BLACK;
   return x;
}
```



Enforce left-leaning condition

```
private Node leanLeft(Node h)
{
   x = rotL(h);
   x.color      = x.left.color;
   x.left.color = RED;
   return x;
}
```



Insert implementation for left-leaning red-black trees (code)

```
private Node insert(Node h, Key key, Value val)
{
   if (h == null)
      return new Node(key, val, RED);

   if (isRed(h.left))
      if (isRed(h.left.left))
         h = splitFourNode(h);

   int cmp = key.compareTo(h.key);
   if (cmp == 0) h.val = val;
   else if (cmp < 0)
      h.left = insert(h.left, key, val);
   else
      h.right = insert(h.right, key, val);

   if (isRed(h.right))
      h = leanLeft(h);

   return h;
}
```

← insert new node at bottom

← split 4-nodes on the way down

← search

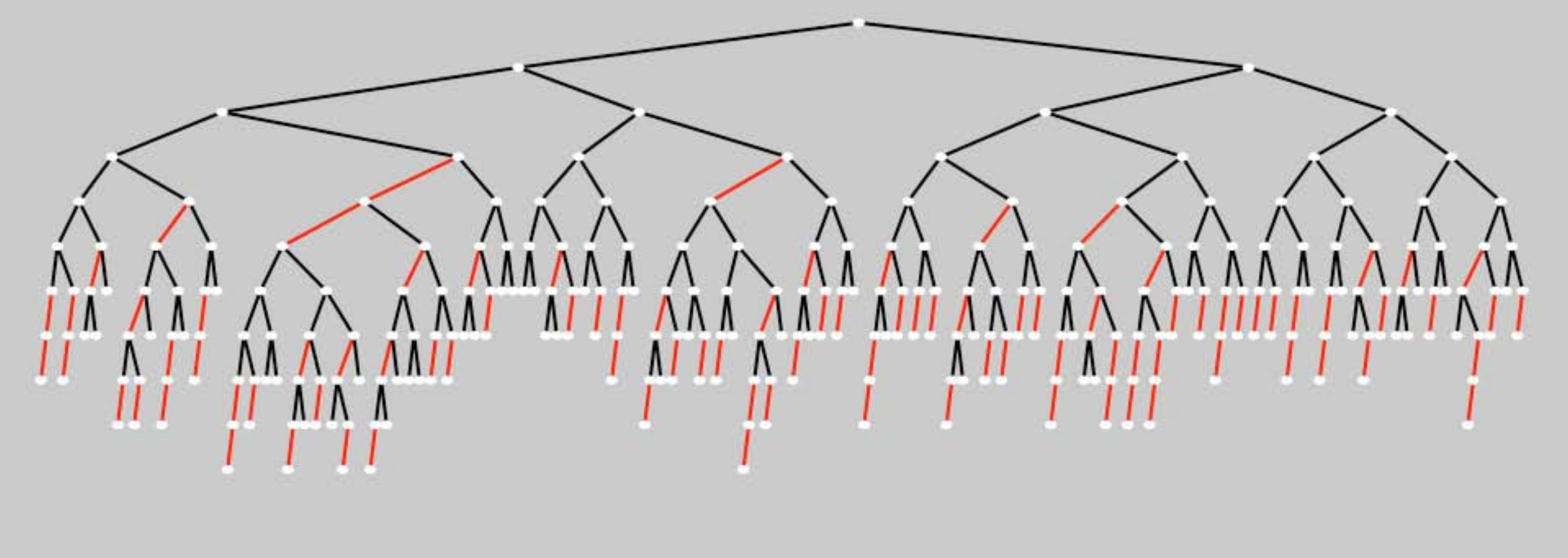
← enforce left-leaning condition
on the way back up

Balance in left-leaning red-black trees

Proposition A. Every path from root to leaf has same number of black links.

Proposition B. Never three red links in-a-row.

Proposition C. Height of tree is less than 3 lg N + 2 in the worst case.



Property D. Height of tree is ~lg N in typical applications.

Property E. Nearly all 4-nodes are on the bottom in the typical applications.

Why left-leaning trees?

Take your pick:

old code (that students had to learn in the past)

```
private Node insert(Node x, Key key, Value val, boolean sw)
{
   if (x == null)
      return new Node(key, value, RED);
   int cmp = key.compareTo(x.key);

   if (isRed(x.left) && isRed(x.right))
   {
      x.color = RED;
      x.left.color  = BLACK;
      x.right.color = BLACK;
   }
   if (cmp == 0) x.val = val;
   else if (cmp < 0))
   {
     x.left = insert(x.left, key, val, false);
     if (isRed(x) && isRed(x.left) && sw)
        x = rotR(x);
     if (isRed(x.left) && isRed(x.left.left))
      {
         x = rotR(x);
         x.color = BLACK; x.right.color = RED;
      }
   }
   else // if (cmp > 0)
   {
      x.right = insert(x.right, key, val, true);
      if (isRed(h) && isRed(x.right) && !sw)
         x = rotL(x);
      if (isRed(h.right) && isRed(h.right.right))
      {
         x = rotL(x);
         x.color = BLACK; x.left.color = RED;
      }
   }
   return x;
}
```

extremely tricky

new code (that you have to learn)

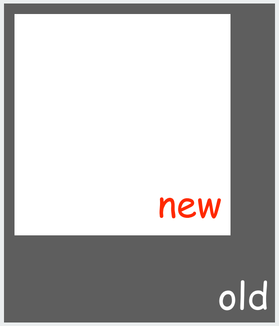
```
private Node insert(Node h, Key key, Value val)
{
    int cmp = key.compareTo(h.key);
    if (h == null)
      return new Node(key, val, RED);
    if (isRed(h.left))
      if (isRed(h.left.left))
      {
         h = rotR(h);
         h.left.color  = BLACK;
      }
   if (cmp == 0) x.val = val;
   else if (cmp < 0)
      h.left = insert(h.left, key, val);
   else
      h.right = insert(h.right, key, val);
   if (isRed(h.right))
   {
      h = rotL(h);
      h.color      = h.left.color;
      h.left.color = RED;
   }
   return h;
}
```

Balanced Trees

- 2-3-4 trees
- red-black trees
- B-trees

straightforward
(if you've paid attention)

Why left-leaning trees?



Simplified code

- left-leaning restriction reduces number of cases
- recursion gives two (easy) chances to fix each node
- short inner loop

Same ideas simplify implementation of other operations

- delete min
- delete max
- delete

Built on the shoulders of many, many old balanced tree algorithms

- AVL trees
- 2-3 trees
- 2-3-4 trees
- skip lists

Bottom line: Left-leaning red-black trees are the simplest to implement

↑ and at least as efficient

Summary of symbol-table implementations

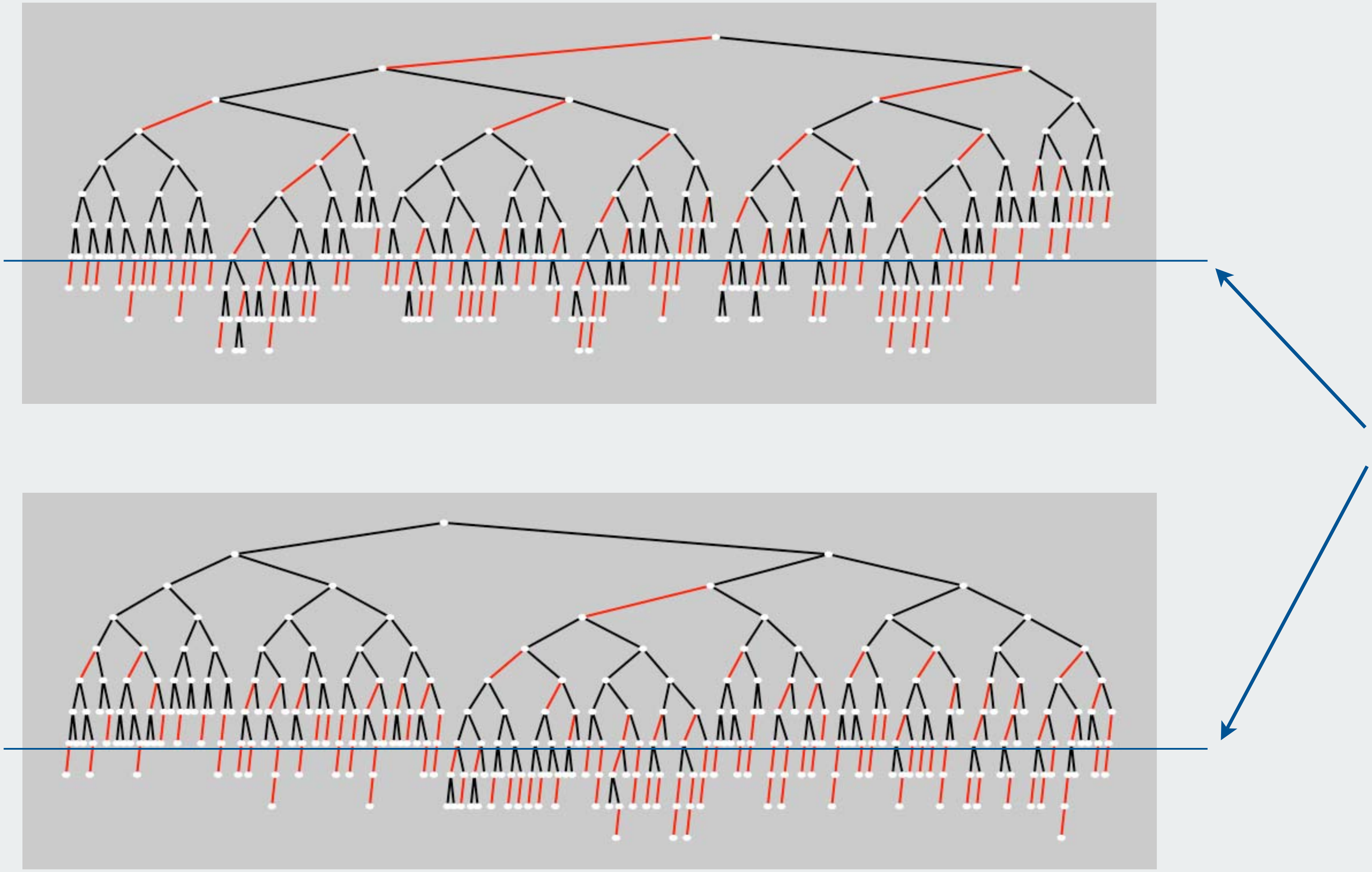
implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes
2-3-4 tree	c lg N	c lg N		c lg N	c lg N		yes
red-black tree	3 lg N	3 lg N	3 lg N	lg N	lg N	lg N	yes

exact value of coefficient unknown
but extremely close to 1

Typical random left-leaning red-black trees

$N = 500$

$\lg N \approx 9$



average node depth

- 2-3-4 trees
- red-black trees
- **B-trees**

B-trees (Bayer-McCreight, 1972)

B-Tree. Generalizes 2-3-4 trees by allowing up to M links per node.

Main application: file systems.

- Reading a page into memory from disk is expensive.
- Accessing info on a page in memory is free.
- Goal: minimize # page accesses.
- Node size M = page size.

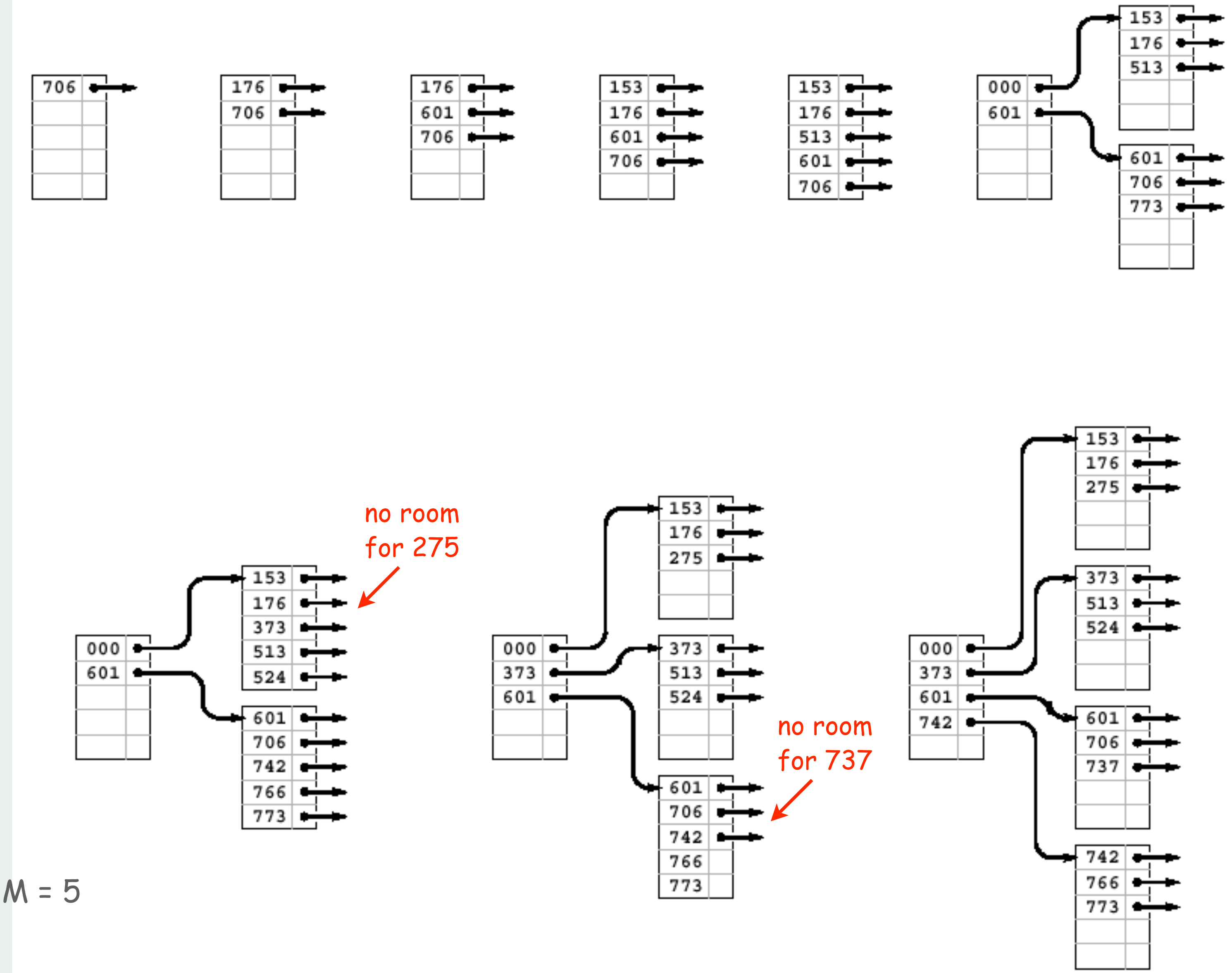
Space-time tradeoff.

- M large ⇒ only a few levels in tree.
- M small ⇒ less wasted space.
- Typical M = 1000, N < 1 trillion.

Bottom line. Number of page accesses is $\log_M N$ per op.

in practice: 3 or 4 (!)

B-Tree Example

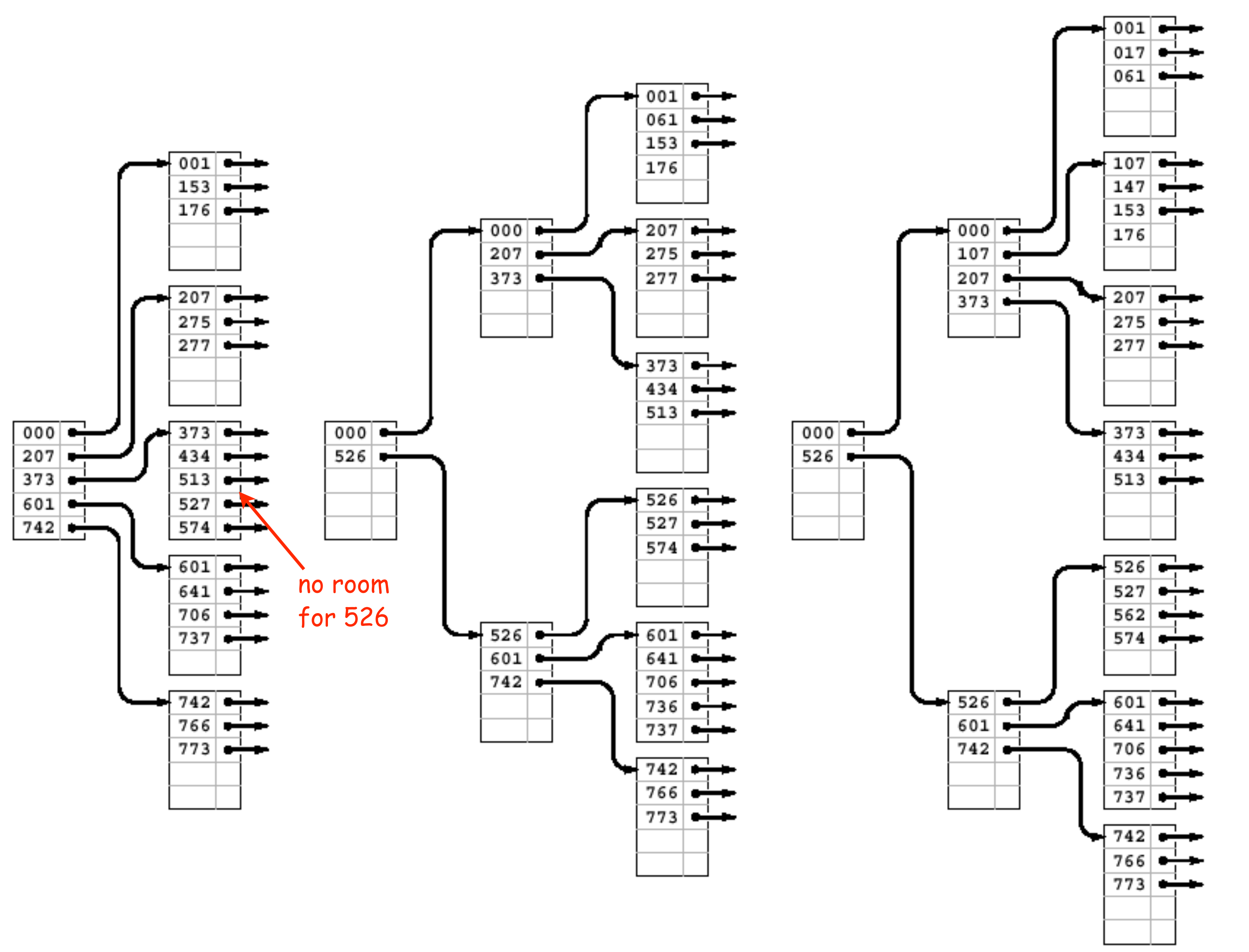


no room for 275

no room for 737

M = 5

B-Tree Example (cont)



no room for 526

Summary of symbol-table implementations

implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.44 lg N	1.44 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.44 lg N	1.44 lg N	1.44 lg N	yes
2-3-4 tree	c lg N	c lg N		c lg N	c lg N		yes
red-black tree	2 lg N	2 lg N	2 lg N	lg N	lg N	lg N	yes
B-tree	1	1	1	1	1	1	yes

B-Tree. Number of page accesses is $\log_M N$ per op.

Balanced trees in the wild

Red-black trees: widely used as system symbol tables

- Java: `java.util.TreeMap, java.util.TreeSet.`
- C++ STL: map, multimap, multiset.
- Linux kernel: `linux/rbtree.h.`

B-Trees: widely used for file systems and databases

- Windows: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL

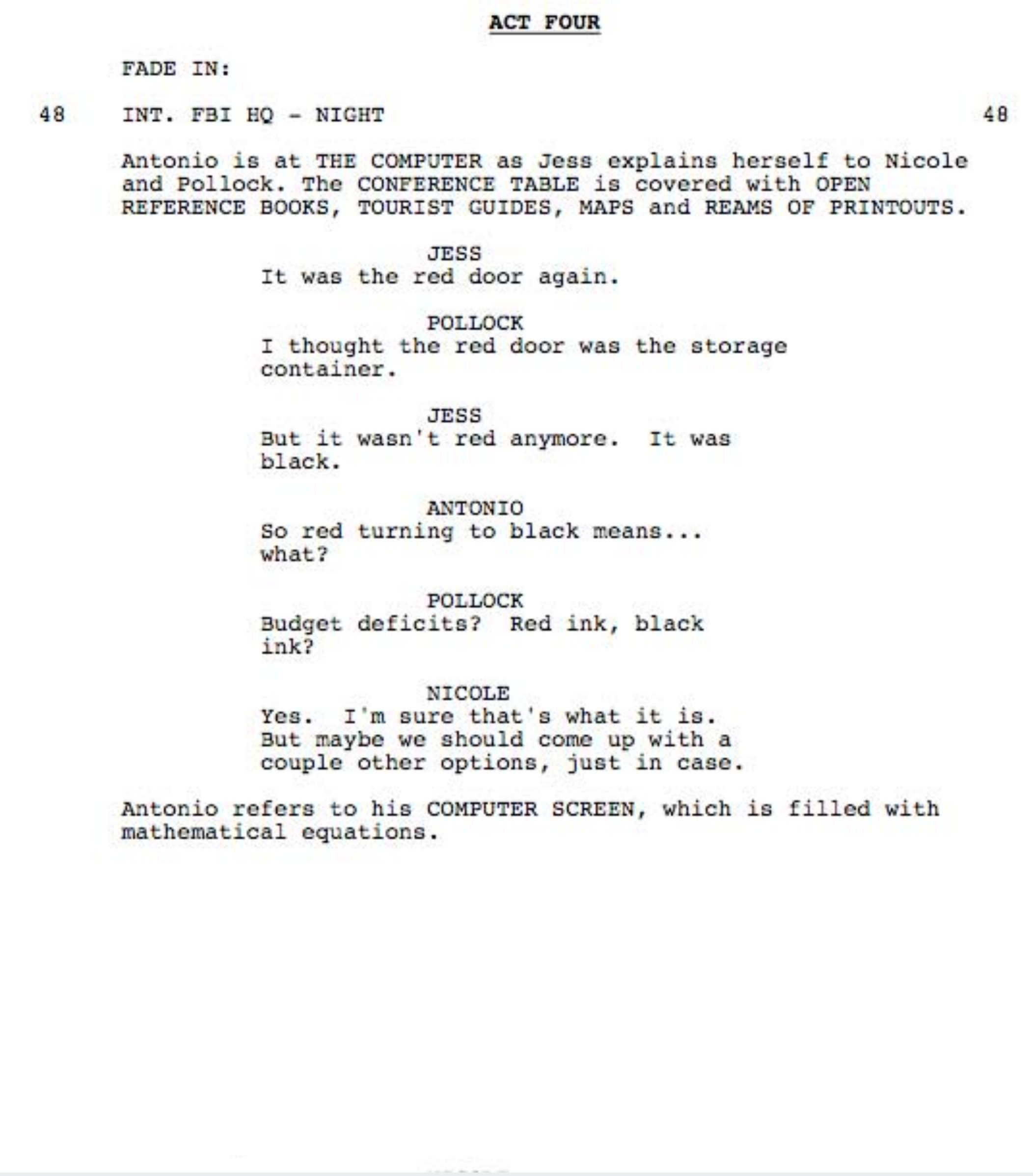
Bottom line: ST implementation with lg N guarantee for all ops.

- Algorithms are variations on a theme: rotations when inserting.
- Easiest to implement, optimal, fastest in practice: LLRB trees
- Abstraction extends to give search algorithms for huge files: B-trees

After the break: Can we do better??

Red-black trees in the wild

Common sense. Sixth sense.
Together they're the FBI's newest team.



ACT FOUR

FADE IN:

48 INT. FBI HQ - NIGHT 48

Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS OF PRINTOUTS.

JESS
It was the red door again.

POLLOCK
I thought the red door was the storage container.

JESS
But it wasn't red anymore. It was black.

ANTONIO
So red turning to black means... what?

POLLOCK
Budget deficits? Red ink, black ink?

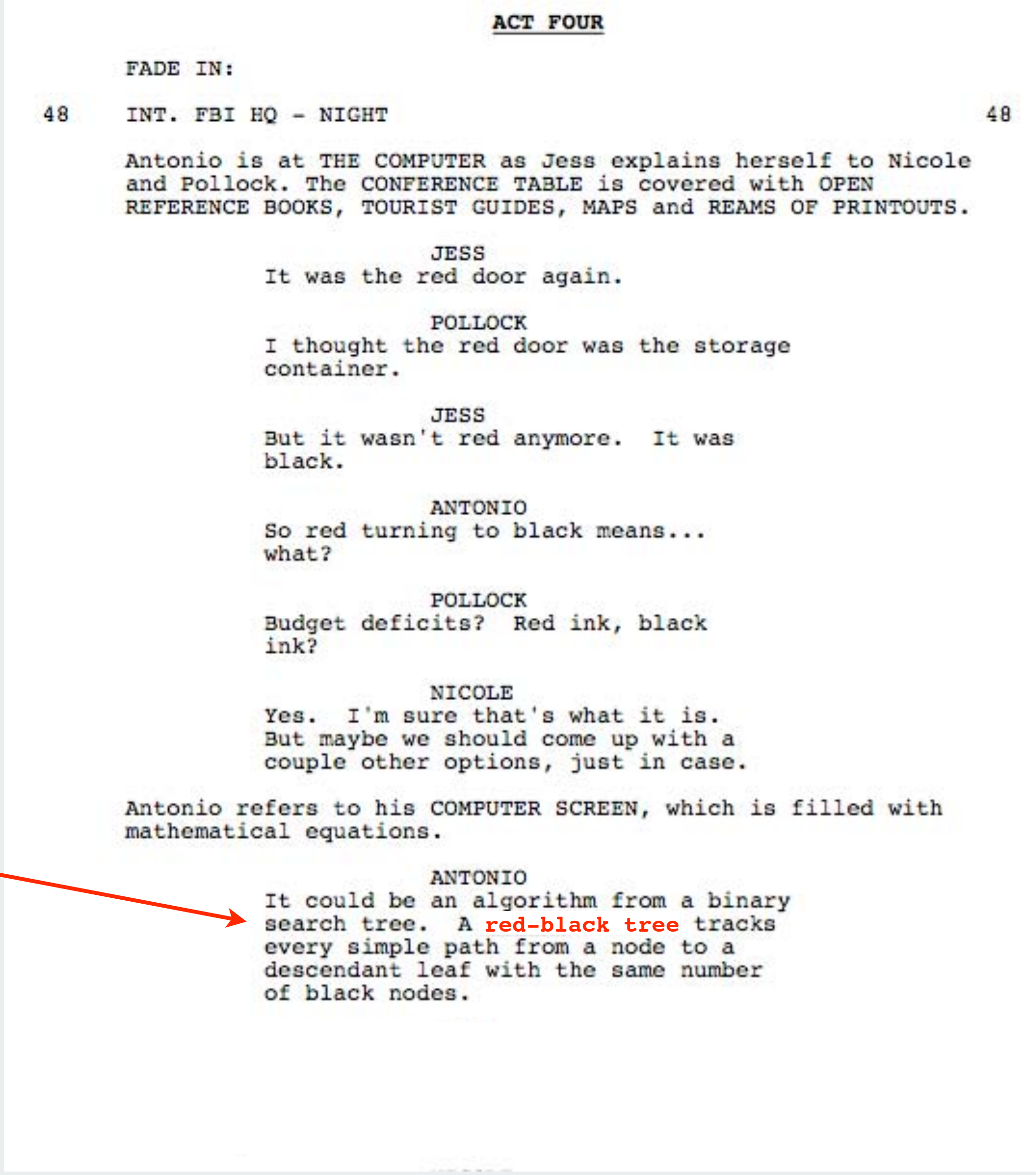
NICOLE
Yes. I'm sure that's what it is. But maybe we should come up with a couple other options, just in case.

Antonio refers to his COMPUTER SCREEN, which is filled with mathematical equations.

Red-black trees in the wild

Common sense. Sixth sense.
Together they're the FBI's newest team.

!!



ACT FOUR

FADE IN:

48 INT. FBI HQ - NIGHT 48

Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS OF PRINTOUTS.

JESS
It was the red door again.

POLLOCK
I thought the red door was the storage container.

JESS
But it wasn't red anymore. It was black.

ANTONIO
So red turning to black means... what?

POLLOCK
Budget deficits? Red ink, black ink?

NICOLE
Yes. I'm sure that's what it is. But maybe we should come up with a couple other options, just in case.

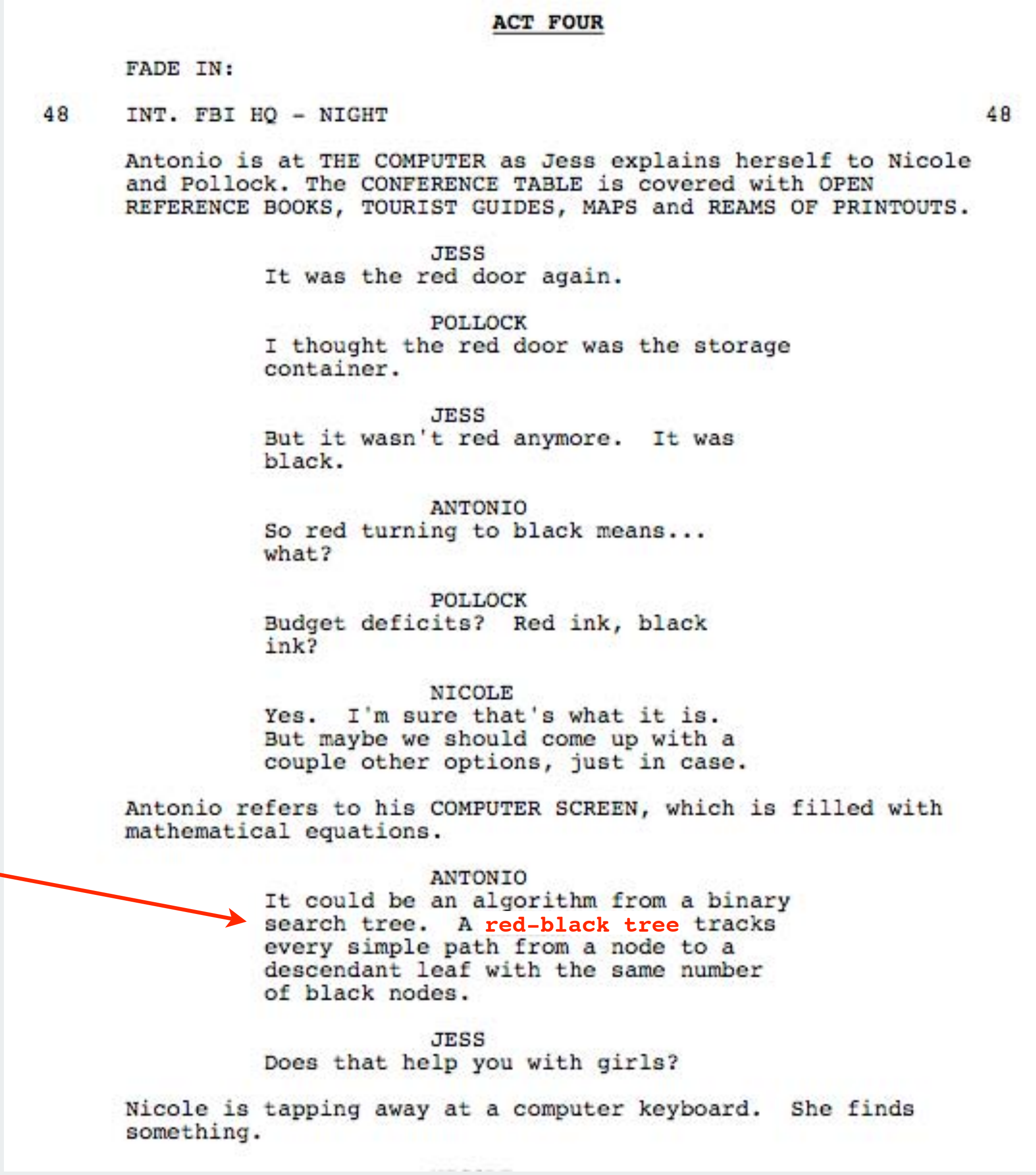
Antonio refers to his COMPUTER SCREEN, which is filled with mathematical equations.

ANTONIO
It could be an algorithm from a binary search tree. A **red-black tree** tracks every simple path from a node to a descendant leaf with the same number of black nodes.

Red-black trees in the wild

Common sense. Sixth sense.
Together they're the FBI's newest team.

!!



ACT FOUR

FADE IN:

48 INT. FBI HQ - NIGHT 48

Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS OF PRINTOUTS.

JESS
It was the red door again.

POLLOCK
I thought the red door was the storage container.

JESS
But it wasn't red anymore. It was black.

ANTONIO
So red turning to black means... what?

POLLOCK
Budget deficits? Red ink, black ink?

NICOLE
Yes. I'm sure that's what it is. But maybe we should come up with a couple other options, just in case.

Antonio refers to his COMPUTER SCREEN, which is filled with mathematical equations.

ANTONIO
It could be an algorithm from a binary search tree. A **red-black tree** tracks every simple path from a node to a descendant leaf with the same number of black nodes.

JESS
Does that help you with girls?

Nicole is tapping away at a computer keyboard. She finds something.

Hashing

- hash functions
- collision resolution
- applications

References:

Algorithms in Java, Chapter 14

http://www.cs.princeton.edu/introalgsds/42hash

Summary of symbol-table implementations

implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.39 lg N	1.39 lg N	1.39 lg N	yes
red-black tree	3 lg N	3 lg N	3 lg N	lg N	lg N	lg N	yes

Can we do better?

Optimize Judiciously

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity. - William A. Wulf

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. - Donald E. Knuth

We follow two rules in the matter of optimization:

Rule 1: Don't do it.

Rule 2 (for experts only). Don't do it yet - that is, not until you have a perfectly clear and unoptimized solution.

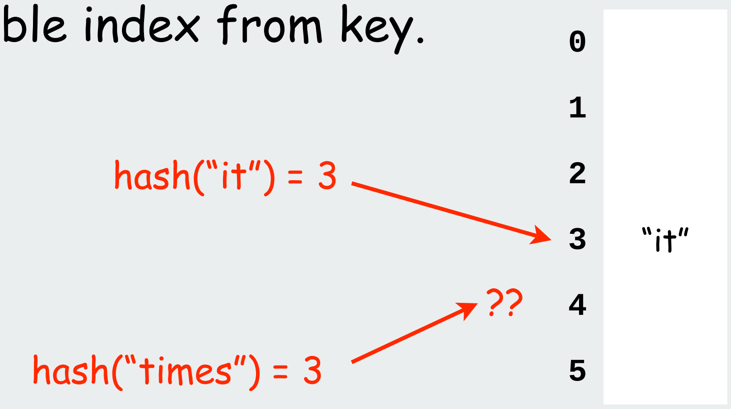
- M. A. Jackson

Reference: Effective Java by Joshua Bloch.

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing table index from key.



Issues.

1. Computing the hash function
2. Collision resolution: Algorithm and data structure to handle two keys that hash to the same index.
3. Equality test: Method for checking whether two keys are equal.

Classic space-time tradeoff.

- No space limitation: trivial hash function with key as address.
- No time limitation: trivial collision resolution with sequential search.
- Limitations on both time and space: hashing (the real world).

- hash functions
- collision resolution
- applications

Computing the hash function

Idealistic goal: scramble the keys uniformly.

- Efficiently computable.
- Each table position equally likely for each key.

thoroughly researched problem,
still problematic in practical applications

Practical challenge: need different approach for each type of key

Ex: Social Security numbers.

- Bad: first three digits.
- Better: last three digits.

573 = California, 574 = Alaska

assigned in chronological order within a
given geographic region

Ex: date of birth.

- Bad: birth year.
- Better: birthday.

Ex: phone numbers.

- Bad: first three digits.
- Better: last three digits.

Hash Codes and Hash Functions

Java convention: all classes implement **`hashCode()`**

`hashcode()` returns a 32-bit **`int`** (between **`-2147483648`** and **`2147483647`**)

Hash function. An **`int`** between **`0`** and **`M-1`** (for use as an array index)

First try:

```
String s = "call";
int code = s.hashCode();
int hash = code % M;
```

3045982

7121

8191

Bug. Don't use **`(code % M)`** as array index

1-in-a billion bug. Don't use **`(Math.abs(code) % M)`** as array index.

OK. Safe to use **`((code & 0x7fffffff) % M)`** as array index.

hex literal 31-bit mask

Java's `hashCode()` convention

Theoretical advantages

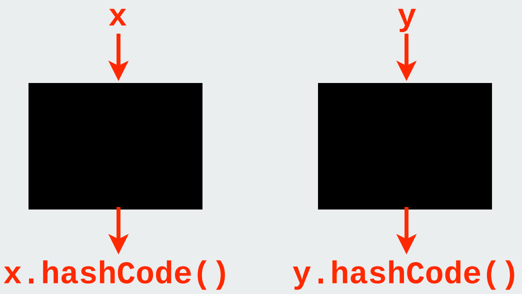
- Ensures hashing can be used for every type of object
- Allows expert implementations suited to each type

Requirements:

- If `x.equals(y)` then x and y must have the same hash code.
- Repeated calls to `x.hashCode()` must return the same value.

Practical realities

- True randomness is hard to achieve
- Cost is an important consideration



Available implementations

- default (inherited from Object): Memory address of x (! ! !)
- customized Java implementations: `String`, `URL`, `Integer`, `Date`.
- User-defined types: users are on their own

that's you!

A typical type

Assumption when using hashing in Java:

Key type has reasonable implementation of **`hashCode()`** and **`equals()`**

Ex. Phone numbers: (609) 867-5309.

exchange

extension

```
public final class PhoneNumber
{
   private final int area, exch, ext;
   public PhoneNumber(int area, int exch, int ext)
   {
      this.area = area;
      this.exch = exch;
      this.ext  = ext;
   }
   public boolean equals(Object y) { // as before }
   public int hashCode()
   {  return 10007 * (area + 1009 * exch) + ext;  }
}
```

sufficiently random?

Fundamental problem:

Need a theorem for each data type to ensure reliability.

A decent hash code design

Java 1.5 string library [see also Program 14.2 in Algs in Java].

```
public int hashCode()
{
   int hash = 0;
   for (int i = 0; i < length(); i++)
      hash = s[i] + (31 * hash);
   return hash;
}
```

ith character of s

char	Unicode
…	…
'a'	97
'b'	98
'c'	99
…	…

- Equivalent to $h = 31^{L-1} \cdot s_0 + \ldots + 31^2 \cdot s_{L-3} + 31 \cdot s_{L-2} + s_{L-1}$.
- Horner's method to hash string of length L: L multiplies/adds

Ex.

```
String s = "call";
int code = s.hashCode();
```

$$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$$

$$= 108 + 31 \cdot (108 + 31 \cdot (99 + 31 \cdot (97)))$$

Provably random? Well, no.

A poor hash code design

Java 1.1 string library.

- For long strings: only examines 8-9 evenly spaced characters.
- Saves time in performing arithmetic…

```
public int hashCode()
{
   int hash = 0;
   int skip = Math.max(1, length() / 8);
   for (int i = 0; i < length(); i += skip)
      hash = (37 * hash) + s[i];
   return hash;
}
```

but great potential for bad collision patterns.

```
http://www.cs.princeton.edu/introcs/13loop/Hello.java
http://www.cs.princeton.edu/introcs/13loop/Hello.class
http://www.cs.princeton.edu/introcs/13loop/Hello.html
http://www.cs.princeton.edu/introcs/13loop/index.html
http://www.cs.princeton.edu/introcs/12type/index.html
```

Basic rule: need to use the whole key.

Digression: using a hash function for data mining

Use content to characterize documents.

Applications

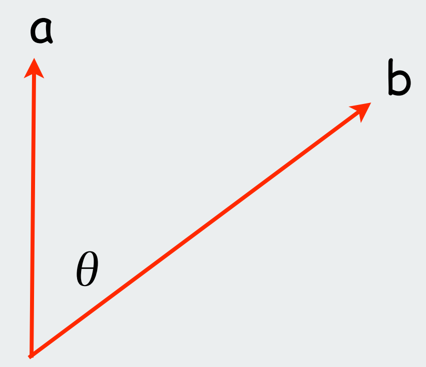
- Search documents on the web for documents similar to a given one.
- Determine whether a new document belongs in one set or another

Approach

- Fix order k and dimension d
- Compute `hashCode() % d` for all k-grams in the document
- Result: d-dimensional vector profile of each document
- To compare documents: Consider angle θ separating vectors

  cos θ close to 0: not similar

  cos θ close to 1: similar



$$\cos\theta = a \cdot b \,/\, |a|\;|b|$$

Digression: using a hash function for data mining

k = 10
d = 65536

```
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of
foolishness
...
```

```
% more genome.txt
CTTTCGGTTTGGAACC
GAAGCCGCGCGTCT
TGTCTGCTGCAGC
ATCGTTC
...
```

	tale.txt		genome.txt	
i	10-grams with hashcode() i	freq	10-grams with hashcode() i	freq
`0`		`0`		`0`
`1`		`0`		`0`
`2`		`0`		`0`

`435`	`best of ti`
	`foolishnes`
		`2`	`TTTCGGTTTG`
			`TGTCTGCTGC`
				`2`

`8999`	`it was the`	`8`		`0`
`...`				
`12122`		`0`	`CTTTCGGTTT`	`3`
`...`				
`34543`	`t was the b`	`5`	`ATGCGGTCGA`	`4`
`...`				
`65535`				
`65536`				

profiles

cos θ small: not similar

Digression: using a hash function to profile a document for data mining

```
public class Document
{
   private String name;
   private double[] profile;
   public Document(String name, int k, int d)
   {
      this.name = name;
      String doc = (new In(name)).readAll();
      int N = doc.length();
      profile = new double[d];
      for (int i = 0; i < N-k; i++)
      {
         int h = doc.substring(i, i+k).hashCode();
         profile[Math.abs(h % d)] += 1;
      }
   }
   public double simTo(Document other)
   {
      // compute dot product and divide by magnitudes
   }
}
```

Digression: using a hash function to compare documents

```
public class CompareAll
{
   public static void main(String args[])
   {
      int k = Integer.parseInt(args[0]);
      int d = Integer.parseInt(args[1]);
      int N = StdIn.readInt();
      Document[] a = new Document[N];
      for (int i = 0; i < N; i++)
         a[i] = new Document(StdIn.readString(), k, d);
      System.out.print("      ");
      for (int j = 0; j < N; j++)
         System.out.printf("    %.4s", a[j].name());
      System.out.println();
      for (int i = 0; i < N; i++)
      {
         System.out.printf("%.4s  ", a[i].name());
         for (int j = 0; j < N; j++)
            System.out.printf("%8.2f", a[i].simTo(a[j]));
         System.out.println();
      }
   }
}
```

Digression: using a hash function to compare documents

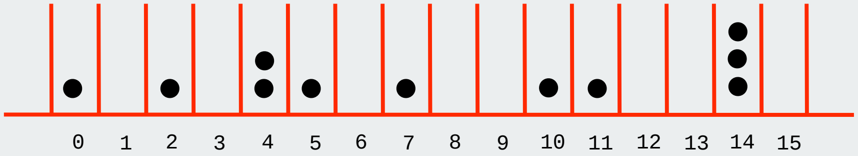
Cons	US Constitution
TomS	"Tom Sawyer"
Huck	"Huckleberry Finn"
Prej	"Pride and Prejudice"
Pict	a photograph
DJIA	financial data
Amaz	Amazon.com website .html source
ACTG	genome

```
% java CompareAll 5 1000 < docs.txt
          Cons    TomS    Huck    Prej    Pict    DJIA    Amaz    ACTG
Cons      1.00    0.89    0.87    0.88    0.35    0.70    0.63    0.58
TomS      0.89    1.00    0.98    0.96    0.34    0.75    0.66    0.62
Huck      0.87    0.98    1.00    0.94    0.32    0.74    0.65    0.61
Prej      0.88    0.96    0.94    1.00    0.34    0.76    0.67    0.63
Pict      0.35    0.34    0.32    0.34    1.00    0.29    0.48    0.24
DJIA      0.70    0.75    0.74    0.76    0.29    1.00    0.62    0.58
Amaz      0.63    0.66    0.65    0.67    0.48    0.62    1.00    0.45
ACTG      0.58    0.62    0.61    0.63    0.24    0.58    0.45    1.00
```

- hash functions
- **collision resolution**
- applications

Helpful results from probability theory

Bins and balls. Throw balls uniformly at random into M bins.



Birthday problem.

Expect two balls in the same bin after $\sqrt{\pi M / 2}$ tosses.

Coupon collector.

Expect every bin has ≥ 1 ball after $\Theta(M \ln M)$ tosses.

Load balancing.

After M tosses, expect most loaded bin has $\Theta(\log M / \log \log M)$ balls.

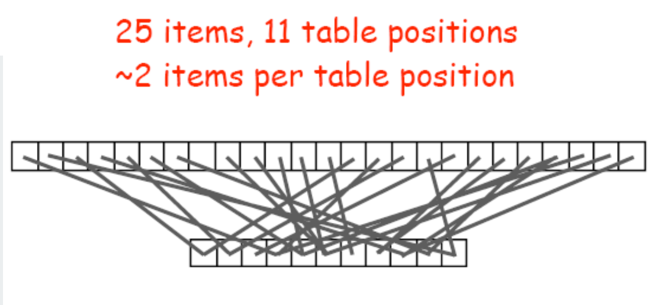
Collisions

Collision. Two distinct keys hashing to same index.

Conclusion. Birthday problem ⇒ can't avoid collisions unless you have a ridiculous amount of memory.

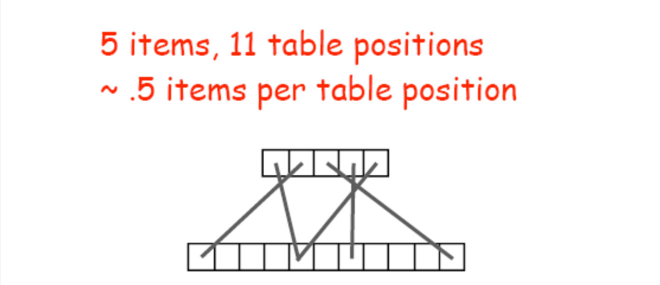
Challenge. Deal with collisions efficiently.

Approach 1:
accept multiple collisions



25 items, 11 table positions
~2 items per table position

Approach 2:
minimize collisions

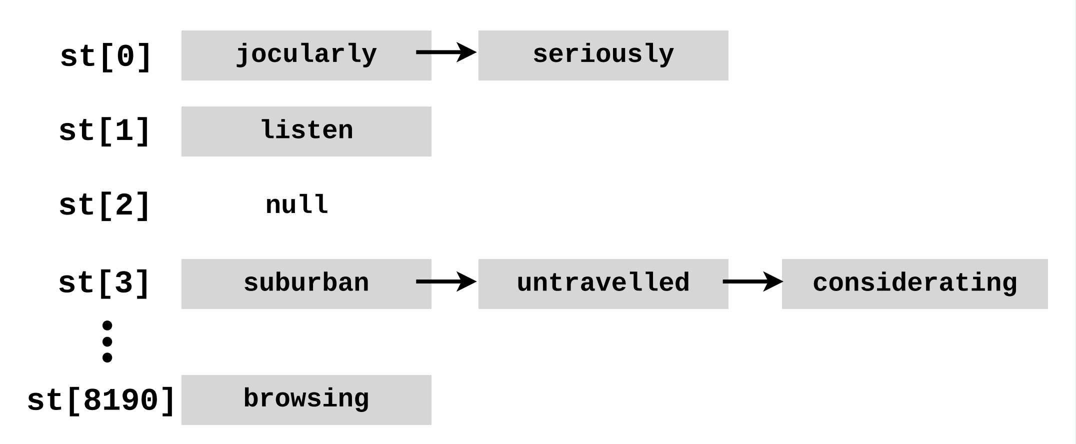


5 items, 11 table positions
~ .5 items per table position

Collision resolution: two approaches

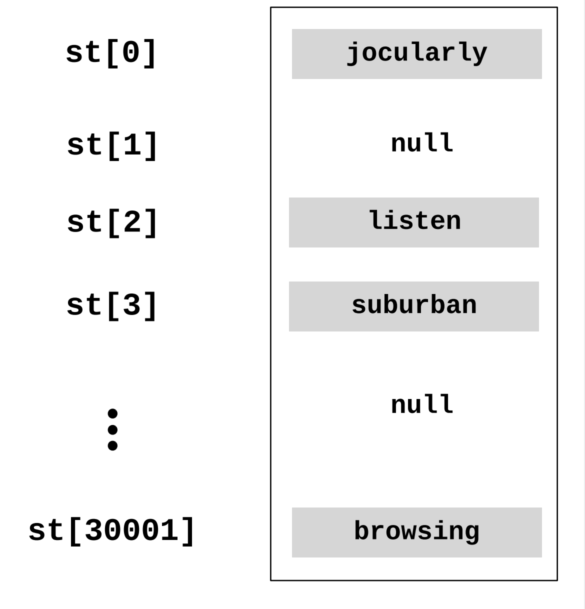
1. Separate chaining. [H. P. Luhn, IBM 1953]
Put keys that collide in a list associated with index.

2. Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]
When a new key collides, find next empty slot, and put it there.



separate chaining (M = 8191, N = 15000)

easy extension of linked list ST implementation



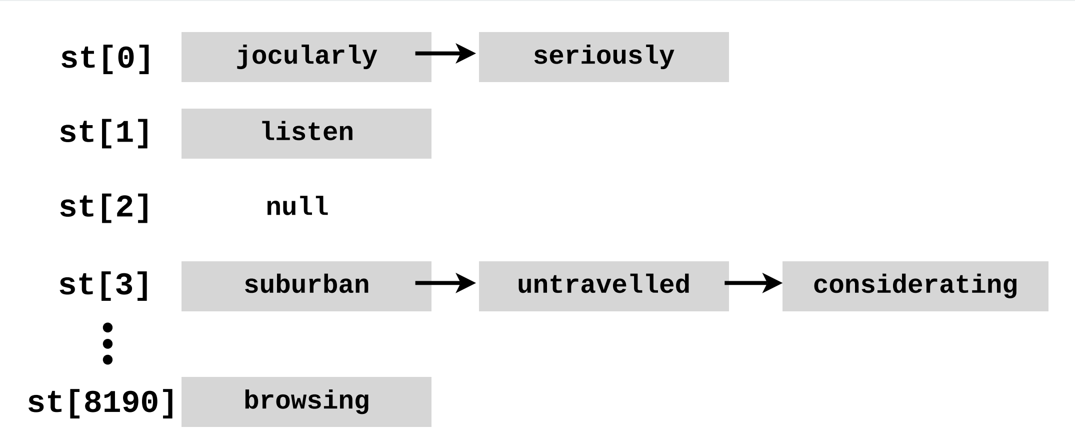
linear probing (M = 30001, N = 15000)

easy extension of array ST implementation

Collision resolution approach 1: separate chaining

Use an array of M < N linked lists. ← good choice: M ≈ N/10

- Hash: map key to integer i between 0 and M-1.
- Insert: put at front of ith chain (if not already there).
- Search: only need to search ith chain.



st[0] jocularly → seriously

st[1] listen

st[2] null

st[3] suburban → untravelled → considerating

⋮

st[8190] browsing

key	hash
call	7121
me	3480
ishmael	5017
seriously	0
untravelled	3
suburban	3
. . .	.

Separate chaining ST implementation (skeleton)

```
public class ListHashST<Key, Value>
{
    private int M = 8191;
    private Node[] st = new Node[M];

    private class Node
    {
        Object key;
        Object val;
        Node next;
        Node(Key key, Value val, Node next)
        {
            this.key   = key;
            this.val   = val;
            this.next  = next;
        }
    }

   private int hash(Key key)
   {  return (key.hashcode() & 0x7fffffff) % M;  }

   public void put(Key key, Value val)
   // see next slide

   public Val get(Key key)
   // see next slide
}
```

could use doubling →

no generics in arrays in Java →

compare with linked lists

Separate chaining ST implementation (put and get)

```
public void put(Key key, Value val)
{
   int i = hash(key);
   for (Node x = st[i]; x != null; x = x.next)
      if (key.equals(x.key))
         { x.val = val; return; }
   st[i] = new Node(key, value, first);
}
```

```
public Value get(Key key)
{
   int i = hash(key);
   for (Node x = st[i]; x != null; x = x.next)
      if (key.equals(x.key))
         return (Value) x.val;
   return null;
}
```

Identical to linked-list code, except hash to pick a list.

Analysis of separate chaining

Separate chaining performance.

- Cost is proportional to length of list.
- Average length = N / M.
- Worst case: all keys hash to same list.

Theorem. Let $\alpha = N / M > 1$ be average length of list. For any $t > 1$, probability that list length $> t\alpha$ is exponentially small in t.

depends on hash map being random map

Parameters.

- M too large ⇒ too many empty chains.
- M too small ⇒ chains too long.
- Typical choice: $\alpha = N / M \approx 10$ ⇒ constant-time ops.

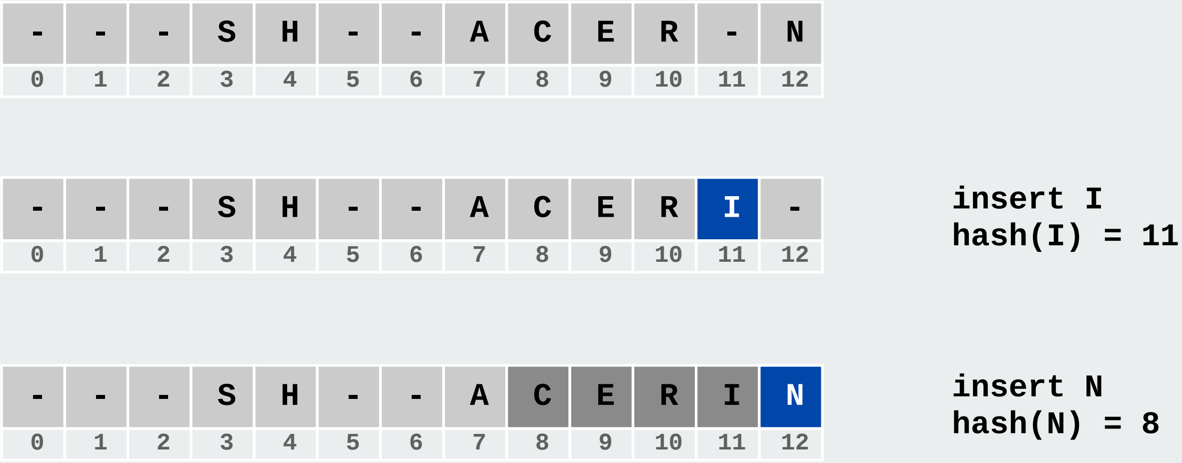
Collision resolution approach 2: open addressing

Use an array of size M >> N. ← good choice: M ≈ 2N

- Hash: map key to integer i between 0 and M-1.

Linear probing:

- Insert: put in slot i if free; if not try i+1, i+2, etc.
- Search: search slot i; if occupied but no match, try i+1, i+2, etc.



-	-	-	S	H	-	-	A	C	E	R	-	N
0	1	2	3	4	5	6	7	8	9	10	11	12

-	-	-	S	H	-	-	A	C	E	R	I	-
0	1	2	3	4	5	6	7	8	9	10	11	12

```
insert I
hash(I) = 11
```

-	-	-	S	H	-	-	A	C	E	R	I	N
0	1	2	3	4	5	6	7	8	9	10	11	12

```
insert N
hash(N) = 8
```

Linear probing ST implementation

compare with elementary unordered array implementation

```
public class ArrayHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[maxN];
   private Key[]   keys = (Key[]) new Object[maxN];

   privat int hash(Key key) // as before

   public void put(Key key, Value val)
   {
      int i;
      for (i = hash(key); keys[i] != null; i = (i+1) % M)
         if (key.equals(keys[i]))
             break;
      vals[i] = val;
      keys[i] = key;
   }

   public Value get(Key key)
   {
      for (int i = hash(key); keys[i] != null; i = (i+1) % M)
         if (key.equals(keys[i]))
             return vals]i];
      return null;
   }
}
```

standard ugly casts

standard array doubling code omitted (double when half full)

Clustering

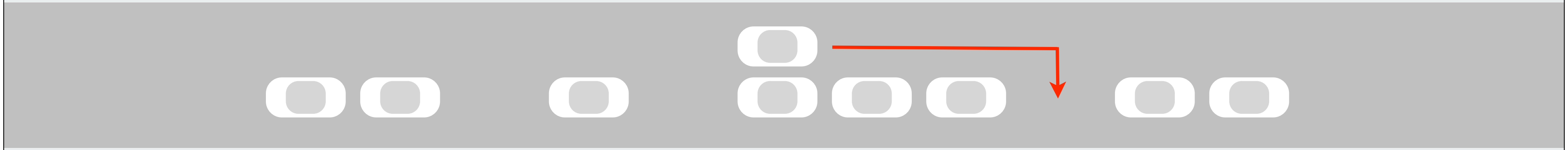
Cluster. A contiguous block of items.

Observation. New keys likely to hash into middle of big clusters.

-	-	-	S	H	A	C	E	-	-	-	X	M	I	-	-	-	P	-	-	R	L	-	-

cluster

Knuth's parking problem. Cars arrive at one-way street with M parking spaces. Each desires a random space i: if space i is taken, try i+1, i+2, …
What is mean displacement of a car?



Empty. With M/2 cars, mean displacement is about 3/2.

Full. Mean displacement for the last car is about $\sqrt{\pi M / 2}$

Analysis of linear probing

Linear probing performance.

- Insert and search cost depend on length of cluster.
- Average length of cluster = $\alpha = N / M$.
- Worst case: all keys hash to same cluster.

but keys more likely to hash to big clusters

Theorem. [Knuth 1962] Let $\alpha = N / M < 1$ be the load factor.

Average probes for insert/search miss

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) = (1 + \alpha + 2\alpha^2 + 3\alpha^3 + 4\alpha^4 + \ldots)/2$$

Average probes for search hit

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right) = 1 + (\alpha + \alpha^2 + \alpha^3 + \alpha^4 + \ldots)/2$$

Parameters.

- Load factor too small ⇒ too many empty array entries.
- Load factor too large ⇒ clusters coalesce.
- Typical choice: $M \approx 2N$ ⇒ constant-time ops.

Hashing: variations on the theme

Many improved versions have been studied:

Ex: Two-probe hashing

- hash to two positions, put key in shorter of the two lists
- reduces average length of the longest list to log log N

Ex: Double hashing

- use linear probing, but skip a variable amount, not just 1 each time
- effectively eliminates clustering
- can allow table to become nearly full

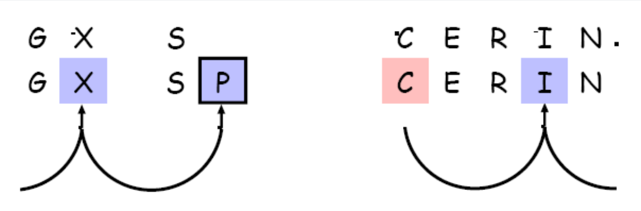
Double hashing

Idea Avoid clustering by using second hash to compute skip for search.

Hash. Map key to integer i between 0 and M-1.
Second hash. Map key to nonzero skip value k.

Ex: k = 1 + (v mod 97).

hashCode()



Effect. Skip values give different search paths for keys that collide.

Best practices. Make k and M relatively prime.

Double Hashing Performance

Theorem. [Guibas-Szemerédi] Let $\alpha = N / M < 1$ be average length of list.

Average probes for insert/search miss

$$\frac{1}{(1 - \alpha)} = 1 + \alpha + \alpha^2 + \alpha^3 + \alpha^4 + \ldots$$

Average probes for search hit

$$\frac{1}{\alpha} \ln \frac{1}{(1 - \alpha)} = 1 + \alpha/2 + \alpha^2/3 + \alpha^3/4 + \alpha^4/5 + \ldots$$

Parameters. Typical choice: $\alpha \approx 1.2 \Rightarrow$ constant-time ops.

Disadvantage. Delete cumbersome to implement.

Hashing Tradeoffs

Separate chaining vs. linear probing/double hashing.

- Space for links vs. empty table slots.
- Small table + linked allocation vs. big coherent array.

Linear probing vs. double hashing.

		load factor α			
		50%	66%	75%	90%
linear probing	get	1.5	2.0	3.0	5.5
	put	2.5	5.0	8.5	55.5
double hashing	get	1.4	1.6	1.8	2.6
	put	1.5	2.0	3.0	5.5

number of probes

Summary of symbol-table implementations

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search	insert	delete		
unordered array	N	N	N	N/2	N/2	N/2	no	`equals()`
ordered array	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
unordered list	N	N	N	N/2	N	N/2	no	`equals()`
ordered list	N	N	N	N/2	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
randomized BST	7 lg N	7 lg N	7 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	lg N	lg N	lg N	yes	`compareTo()`

hashing	1*	1*	1*	1*	1*	1*	no	`equals()`
								`hashCode()`

* assumes random hash code

Hashing versus balanced trees

Hashing

- simpler to code
- no effective alternative for unordered keys
- faster for simple keys (a few arithmetic ops versus lg N compares)
- (Java) better system support for strings [cached hashcode]
- does your hash function produce random values for your key type??

Balanced trees

- stronger performance guarantee
- can support many more operations for ordered keys
- easier to implement `compareTo()` correctly than `equals()` and `hashCode()`

Java system includes both

- red-black trees: `java.util.TreeMap, java.util.TreeSet`
- hashing: `java.util.HashMap, java.util.IdentityHashMap`

Typical “full” ST API

`public class *ST<Key extends Comparable<Key>, Value>`

	`*ST()`	create a symbol table
`void`	`put(Key key, Value val)`	put key-value pair into the table
`Value`	`get(Key key)`	return value paired with **key** (**null** if **key** is not in table)
`boolean`	`contains(Key key)`	is there a value paired with **key**?
`Key`	`min()`	smallest key
`Key`	`max()`	largest key
`Key`	`next(Key key)`	next largest key (**null** if **key** is max)
`Key`	`prev(Key key)`	next smallest key (**null** if **key** is min)
`void`	`remove(Key key)`	remove key-value pair from table
`Iterator<Key>`	`iterator()`	iterator through keys in table

Hashing is not suitable for implementing such an API (no order)

BSTs are easy to extend to support such an API (basic tree ops)

Ex: Can use LLRB trees implement priority queues for distinct keys

- hash functions
- collision resolution
- applications

Set ADT

Set. Collection of distinct keys.

`public class *SET<Key extends Comparable<Key>, Value>`

	`SET()`	create a set
`void`	`add(Key key)`	put key into the set
`boolean`	`contains(Key key)`	is there a value paired with **key**?
`void`	`remove(Key key)`	remove key from the set
`Iterator<Key>`	`iterator()`	iterator through all keys in the set

Normal mathematical assumption: collection is unordered

Typical (eventual) client expectation: ordered iteration

unordered iterator
O(1) search

Q. How to implement?

A0. Hashing (our ST code [value removed] or **java.util.HashSet**)

A1. Red-black BST (our ST code [value removed] or **java.util.TreeSet**)

ordered iterator
O(log N) search

SET client example 1: dedup filter

Remove duplicates from strings in standard input

- Read a key.
- If key is not in set, insert and print it.

```
public class DeDup
{
   public static void main(String[] args)
   {
      SET<String> set = new SET<String>();
      while (!StdIn.isEmpty())
      {
         String key = StdIn.readString();
         if (!set.contains(key))
         {
            set.add(key);
            StdOut.println(key);
         }
      }
   }
}
```

No iterator needed.
Output is in same order
as input with
dups removed.

```
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of
foolishness
...

% java Dedup < tale.txt
it
was
the
best
of
times
worst
age
wisdom
foolishness
...
```

Simplified version of **FrequencyCount** (no iterator needed)

SET client example 2A: lookup filter

Print words from standard input that are found in a list

- Read in a list of words from one file.
- Print out all words from standard input that are in the list.

```
public class LookupFilter
{
   public static void main(String[] args)
   {
      SET<String> set = new SET<String>();

      In in = new In(args[0]);
      while (!in.isEmpty())
         set.add(in.readString());

      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (set.contains(word))
            StdOut.println(word);
      }
   }
}
```

create SET

process list

print words that are not in list

SET client example 2B: exception filter

Print words from standard input that are not found in a list

- Read in a list of words from one file.
- Print out all words from standard input that are not in the list.

```
public class LookupFilter
{
   public static void main(String[] args)
   {
      SET<String> set = new SET<String>();

      In in = new In(args[0]);
      while (!in.isEmpty())
         set.add(in.readString());

      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (!set.contains(word))
            StdOut.println(word);
      }
   }
}
```

create SET

process list

print words that are not in list

SET filter applications

application	purpose	key	type	in list	not in list
dedup	eliminate duplicates		dedup	duplicates	unique keys
spell checker	find misspelled words	word	exception	dictionary	misspelled words
browser	mark visited pages	URL	lookup	visited pages	
chess	detect draw	board	lookup	positions	
spam filter	eliminate spam	IP addr	exception	spam	good mail
trusty filter	allow trusted mail	URL	lookup	good mail	
credit cards	check for stolen cards	number	exception	stolen cards	good cards

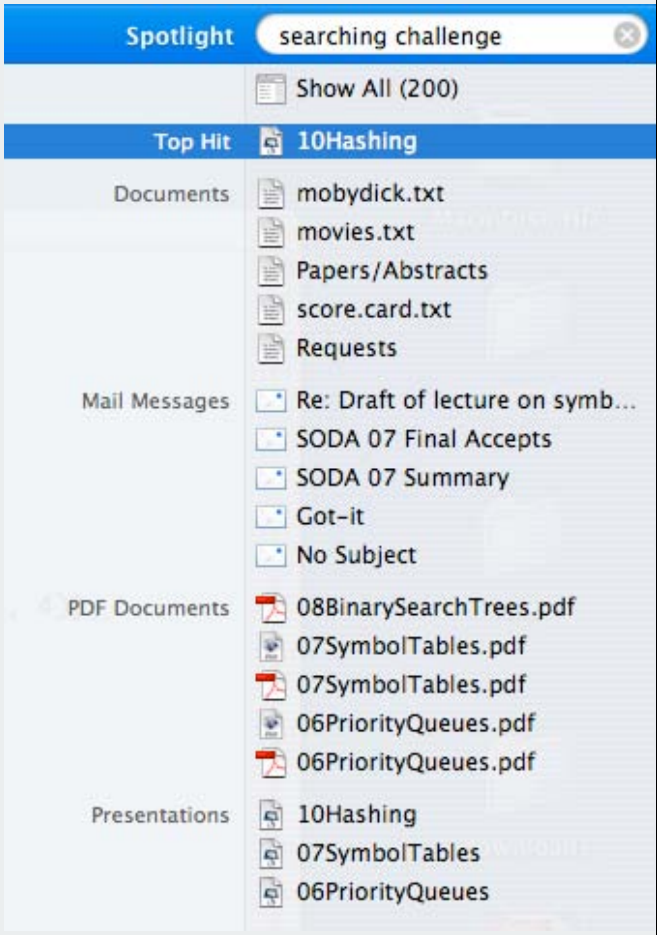
Searching challenge:

Problem: Index for a PC or the web

Assumptions: 1 billion++ words to index

Which searching method to use?

1) hashing implementation of SET
2) hashing implementation of ST
3) red-black-tree implementation of ST
4) red-black-tree implementation of SET
5) doesn't matter much



Spotlight searching challenge

Show All (200)

Top Hit 10Hashing

Documents
mobydick.txt
movies.txt
Papers/Abstracts
score.card.txt
Requests

Mail Messages
Re: Draft of lecture on symb...
SODA 07 Final Accepts
SODA 07 Summary
Got-it
No Subject

PDF Documents
08BinarySearchTrees.pdf
07SymbolTables.pdf
07SymbolTables.pdf
06PriorityQueues.pdf
06PriorityQueues.pdf

Presentations
10Hashing
07SymbolTables
06PriorityQueues

Index for search in a PC

```
ST<String, SET<File>> st = new ST<String, SET<File>>();
for (File f: filesystem)
{
   In in = new In(f);
   String[] words = in.readAll().split("\\s+");
   for (int i = 0; i < words.length; i++)
   {
       String s = words[i];
       if (!st.contains(s))
          st.put(s, new SET<File>());
       SET<File> files = st.get(s);
       files.add(f);
   }
}
```

← build index

```
SET<File> files = st.get(s);
for (File f: files) ...
```

← process lookup request

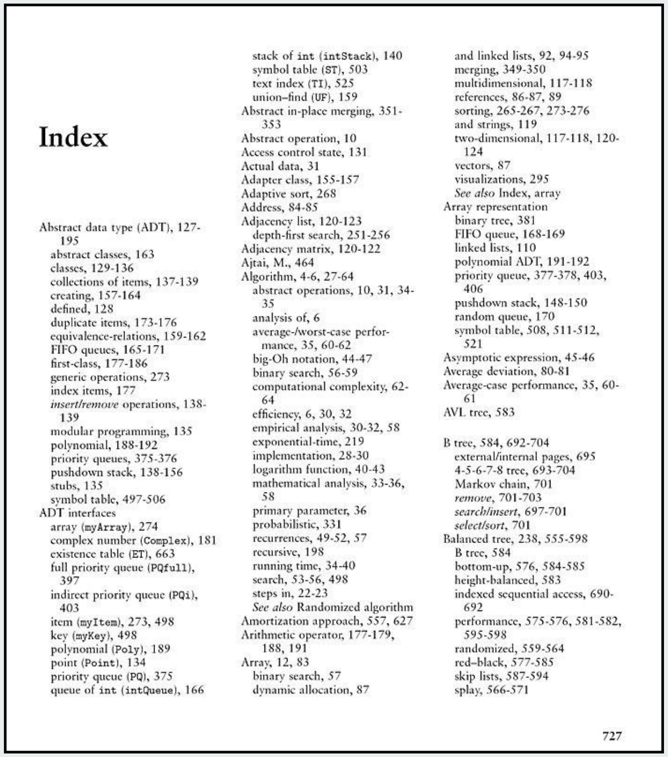
Searching challenge:

Problem: Index for a book

Assumptions: book has 100,000+ words

Which searching method to use?

1) hashing implementation of SET
2) hashing implementation of ST
3) red-black-tree implementation of ST
4) red-black-tree implementation of SET
5) doesn't matter much



Index

- Abstract data type (ADT), 127-195
  - abstract classes, 163
  - classes, 129-136
  - collections of items, 137-139
  - creating, 157-164
  - defined, 128
  - duplicate items, 173-176
  - equivalence-relations, 159-162
  - FIFO queues, 165-171
  - first-class, 177-186
  - generic operations, 273
  - index items, 177
  - *insert/remove* operations, 138-139
  - modular programming, 135
  - polynomial, 188-192
  - priority queues, 375-376
  - pushdown stack, 138-156
  - stubs, 135
  - symbol table, 497-506
- ADT interfaces
  - array (`myArray`), 274
  - complex number (`Complex`), 181
  - existence table (`ET`), 663
  - full priority queue (`PQfull`), 397
  - indirect priority queue (`PQi`), 403
  - item (`myItem`), 273, 498
  - key (`myKey`), 498
  - polynomial (`Poly`), 189
  - point (`Point`), 134
  - priority queue (`PQ`), 375
  - queue of `int` (`intQueue`), 166
  - stack of `int` (`intStack`), 140
  - symbol table (`ST`), 503
  - text index (`TI`), 525
  - union–find (`UF`), 159
- Abstract in-place merging, 351-353
- Abstract operation, 10
- Access control state, 131
- Actual data, 31
- Adapter class, 155-157
- Adaptive sort, 268
- Address, 84-85
- Adjacency list, 120-123
  - depth-first search, 251-256
- Adjacency matrix, 120-122
- Ajtai, M., 464
- Algorithm, 4-6, 27-64
  - abstract operations, 10, 31, 34-35
  - analysis of, 6
  - average-/worst-case performance, 35, 60-62
  - big-Oh notation, 44-47
  - binary search, 56-59
  - computational complexity, 62-64
  - efficiency, 6, 30, 32
  - empirical analysis, 30-32, 58
  - exponential-time, 219
  - implementation, 28-30
  - logarithm function, 40-43
  - mathematical analysis, 33-36, 58
  - primary parameter, 36
  - probabilistic, 331
  - recurrences, 49-52, 57
  - recursive, 198
  - running time, 34-40
  - search, 53-56, 498
  - steps in, 22-23
  - *See also* Randomized algorithm
- Amortization approach, 557, 627
- Arithmetic operator, 177-179, 188, 191
- Array, 12, 83
  - binary search, 57
  - dynamic allocation, 87
  - and linked lists, 92, 94-95
  - merging, 349-350
  - multidimensional, 117-118
  - references, 86-87, 89
  - sorting, 265-267, 273-276
  - and strings, 119
  - two-dimensional, 117-118, 120-124
  - vectors, 87
  - visualizations, 295
  - *See also* Index, array
- Array representation
  - binary tree, 381
  - FIFO queue, 168-169
  - linked lists, 110
  - polynomial ADT, 191-192
  - priority queue, 377-378, 403, 406
  - pushdown stack, 148-150
  - random queue, 170
  - symbol table, 508, 511-512, 521
- Asymptotic expression, 45-46
- Average deviation, 80-81
- Average-case performance, 35, 60-61
- AVL tree, 583

- B tree, 584, 692-704
  - external/internal pages, 695
  - 4-5-6-7-8 tree, 693-704
  - Markov chain, 701
  - *remove*, 701-703
  - *search/insert*, 697-701
  - *select/sort*, 701
- Balanced tree, 238, 555-598
  - B tree, 584
  - bottom-up, 576, 584-585
  - height-balanced, 583
  - indexed sequential access, 690-692
  - performance, 575-576, 581-582, 595-598
  - randomized, 559-564
  - red–black, 577-585
  - skip lists, 587-594
  - splay, 566-571

727

Index for a book

```
public class Index
{
   public static void main(String[] args)
   {
      String[] words = StdIn.readAll().split("\\s+");
      ST<String, SET<Integer>> st;
      st = new ST<String, SET<Integer>>();

      for (int i = 0; i < words.length; i++)
      {
         String s = words[i];
         if (!st.contains(s))
            st.put(s, new SET<Integer>());
         SET<Integer> pages = st.get(s);
         pages.add(page(i));
      }

      for (String s : st)
         StdOut.println(s + ": " + st.get(s));

   }
}
```

read book and
create ST

process all
words

print index!

Requires ordered iterators (not hashing)

Hashing in the wild: Java implementations

Java has built-in libraries for hash tables.

- **java.util.HashMap** = separate chaining implementation.
- **java.util.IdentityHashMap** = linear probing implementation.

```
import java.util.HashMap;
public class HashMapDemo
{
   public static void main(String[] args)
   {
      HashMap<String, String> st = new HashMap <String, String>();
      st.put("www.cs.princeton.edu", "128.112.136.11");
      st.put("www.princeton.edu",    "128.112.128.15");
      StdOut.println(st.get("www.cs.princeton.edu"));
   }
}
```

Null value policy.

- Java **HashMap** allows **null** values.
- Our implementation forbids **null** values.

Using HashMap

Implementation of our API with java.util.HashMap.

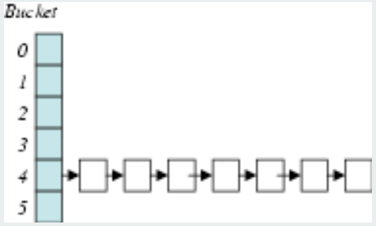
```
import java.util.HashMap;
import java.util.Iterator;

public class ST<Key, Value> implements Iterable<Key>
{
   private HashMap<Key, Value> st = new HashMap<Key, Value>();

   public void put(Key key, Value val)
   {
      if (val == null) st.remove(key);
      else             st.put(key, val);
   }
   public Value get(Key key)              { return st.get(key);            }
   public Value remove(Key key)           { return st.remove(key);         }
   public boolean contains(Key key)       { return st.contains(key);       }
   public int size() contains(Key key) { return st.size();              }
   public Iterator<Key> iterator()      { return st.keySet().iterator(); }
}
```

Hashing in the wild: algorithmic complexity attacks

Is the random hash map assumption important in practice?

- Obvious situations: aircraft control, nuclear reactor, pacemaker.
- Surprising situations: denial-of-service attacks.

malicious adversary learns your ad hoc hash function (e.g., by reading Java API) and causes a big pile-up in single address that grinds performance to a halt



Real-world exploits. [Crosby-Wallach 2003]

- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem
- Perl 5.8.0: insert carefully chosen strings into associative array.
- Linux 2.4.20 kernel: save files with carefully chosen names.

Reference: http://www.cs.rice.edu/~scrosby/hash

Algorithmic complexity attack on the Java Library

Goal. Find strings with the same hash code.

Solution. The base-31 hash code is part of Java's string API.

Key	hashCode()
Aa	2112
BB	2112

Key	hashCode()
AaAaAaAa	-540425984
AaAaAaBB	-540425984
AaAaBBAa	-540425984
AaAaBBBB	-540425984
AaBBAaAa	-540425984
AaBBAaBB	-540425984
AaBBBBAa	-540425984
AaBBBBBB	-540425984
BBAaAaAa	-540425984
BBAaAaBB	-540425984
BBAaBBAa	-540425984
BBAaBBBB	-540425984
BBBBAaAa	-540425984
BBBBAaBB	-540425984
BBBBBBAa	-540425984
BBBBBBBB	-540425984

2^N strings of length 2N
that hash to same value!

Does your hash function produce random values for your key type??

One-Way Hash Functions

One-way hash function. Hard to find a key that will hash to a desired value, or to find two keys that hash to same value.

Ex. $\underbrace{\text{MD4, MD5, SHA-0, SHA-1}}_{\text{insecure}}$, SHA-2, WHIRLPOOL, RIPEMD-160.

```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

// prints bytes as hex string
```

Applications. Digital fingerprint, message digest, storing passwords.

Too expensive for use in ST implementations (use balanced trees)

Undirected Graphs

- Graph API
- maze exploration
- depth-first search
- breadth-first search
- connected components
- challenges

References:

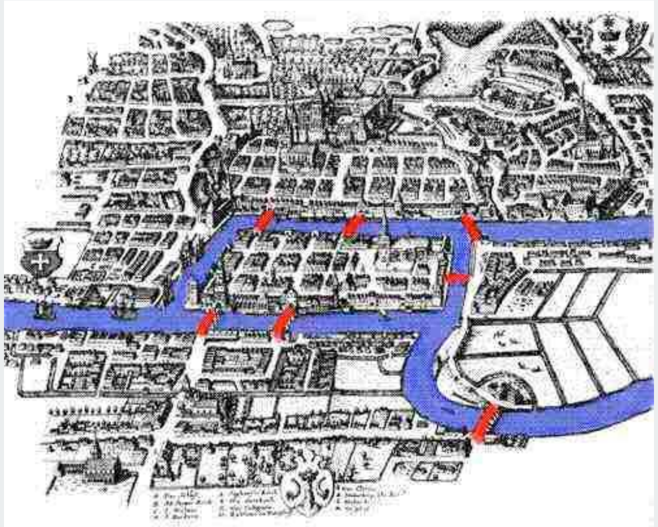
Algorithms in Java, Chapters 17 and 18
Intro to Programming in Java, Section 4.5
http://www.cs.princeton.edu/introalgsds/51undirected

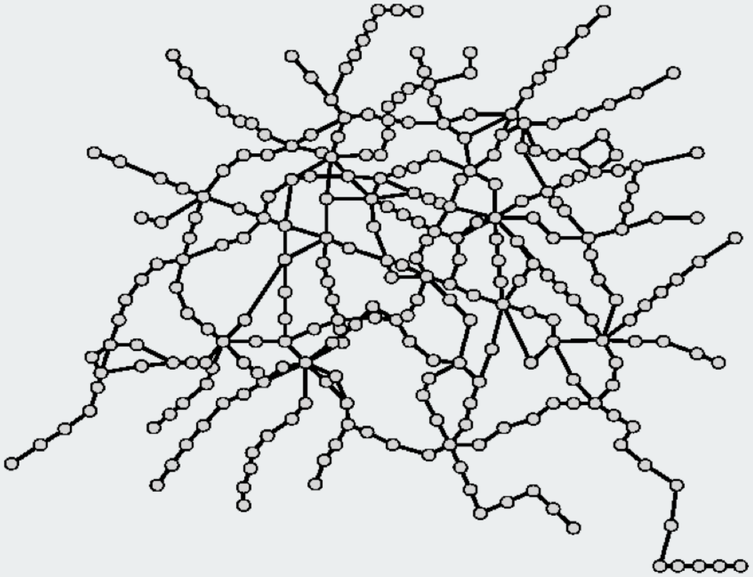
Undirected graphs

Graph. Set of vertices connected pairwise by edges.

Why study graph algorithms?

- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.
- Hundreds of graph algorithms known.
- Thousands of practical applications.



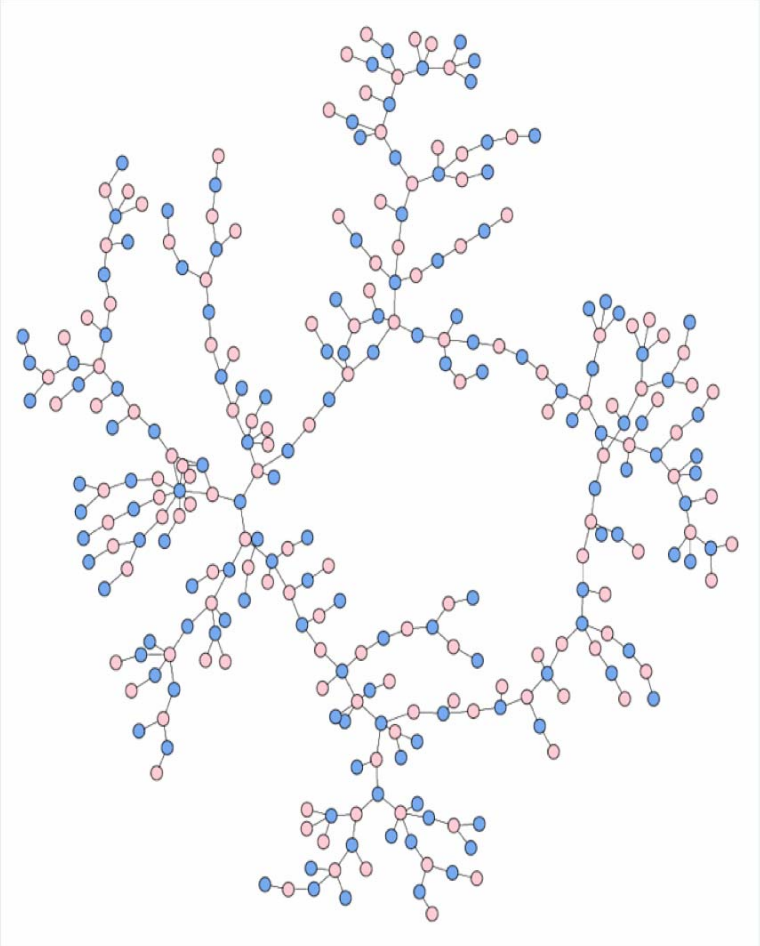


Graph applications

graph	vertices	edges
communication	telephones, computers	fiber optic cables
circuits	gates, registers, processors	wires
mechanical	joints	rods, beams, springs
hydraulic	reservoirs, pumping stations	pipelines
financial	stocks, currency	transactions
transportation	street intersections, airports	highways, airway routes
scheduling	tasks	precedence constraints
software systems	functions	function calls
internet	web pages	hyperlinks
games	board positions	legal moves
social relationship	people, actors	friendships, movie casts
neural networks	neurons	synapses
protein networks	proteins	protein-protein interactions
chemical compounds	molecules	bonds

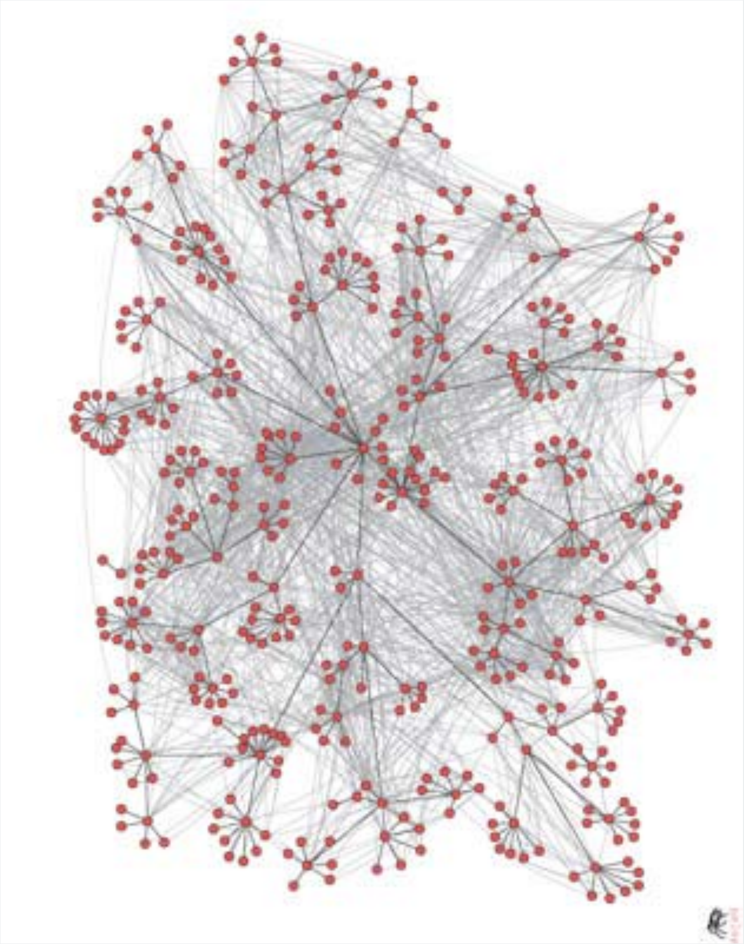
Social networks

high school dating



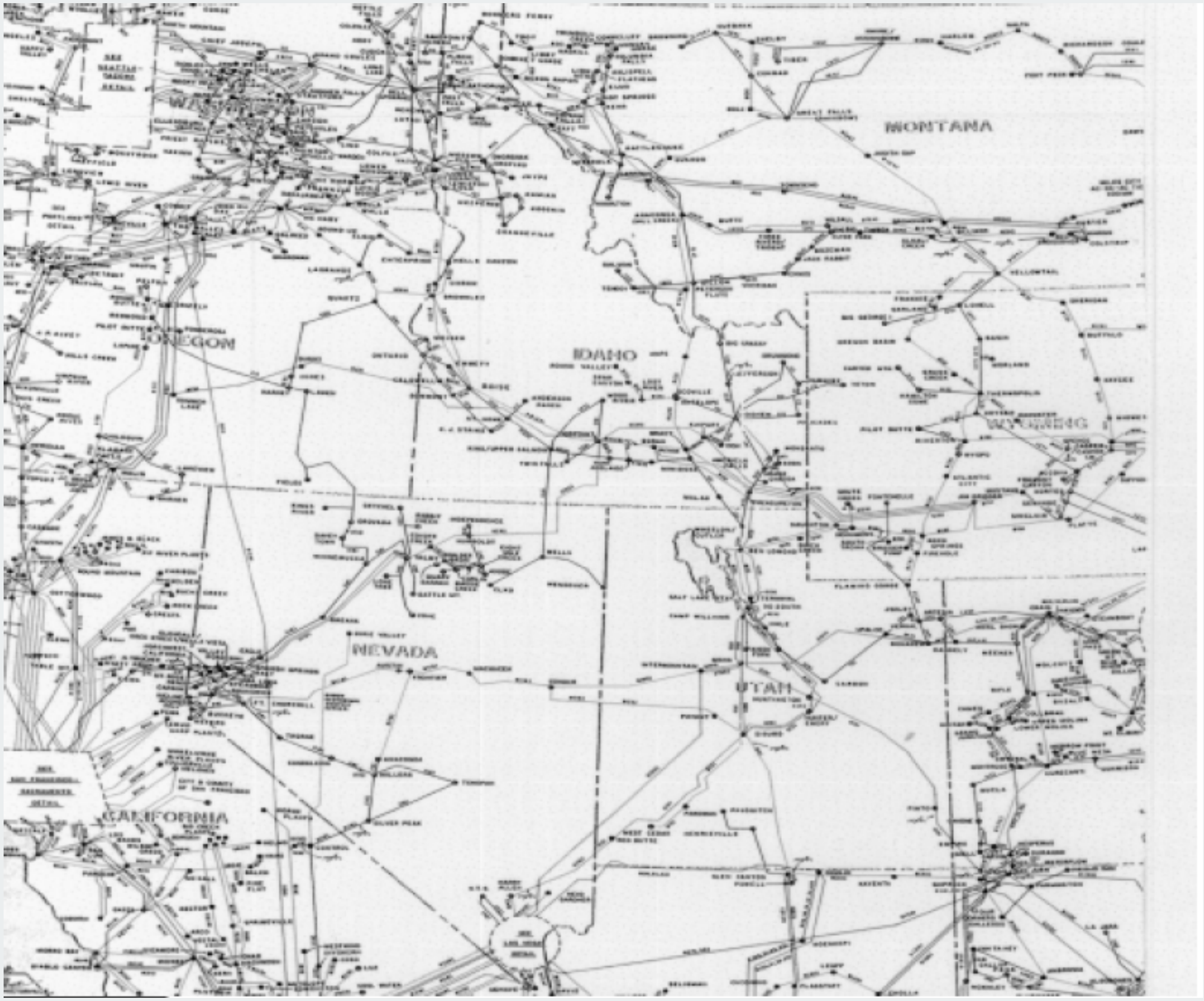
Reference: Bearman, Moody and Stovel, 2004
image by Mark Newman

corporate e-mail



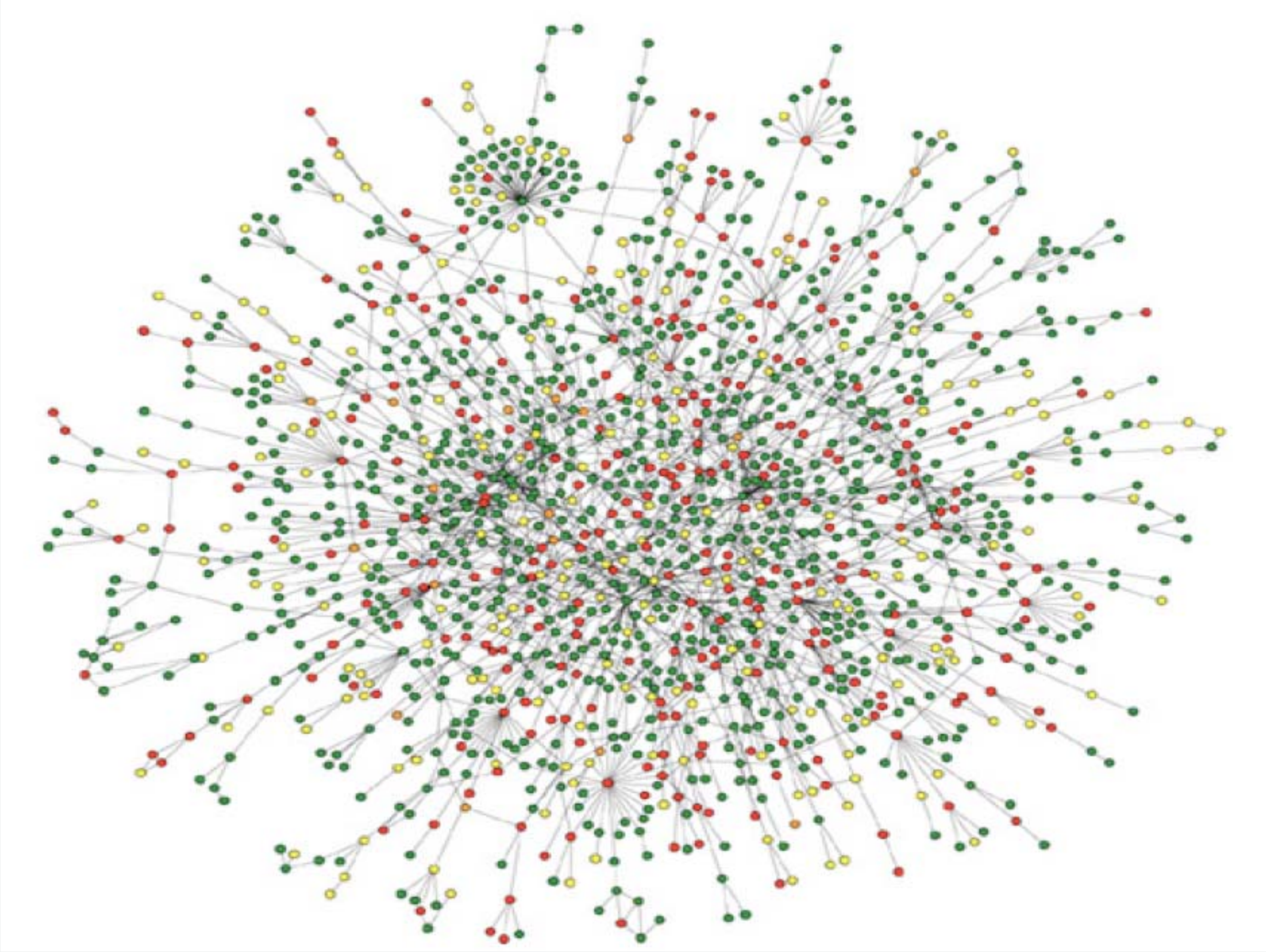
Reference: Adamic and Adar, 2004

Power transmission grid of Western US



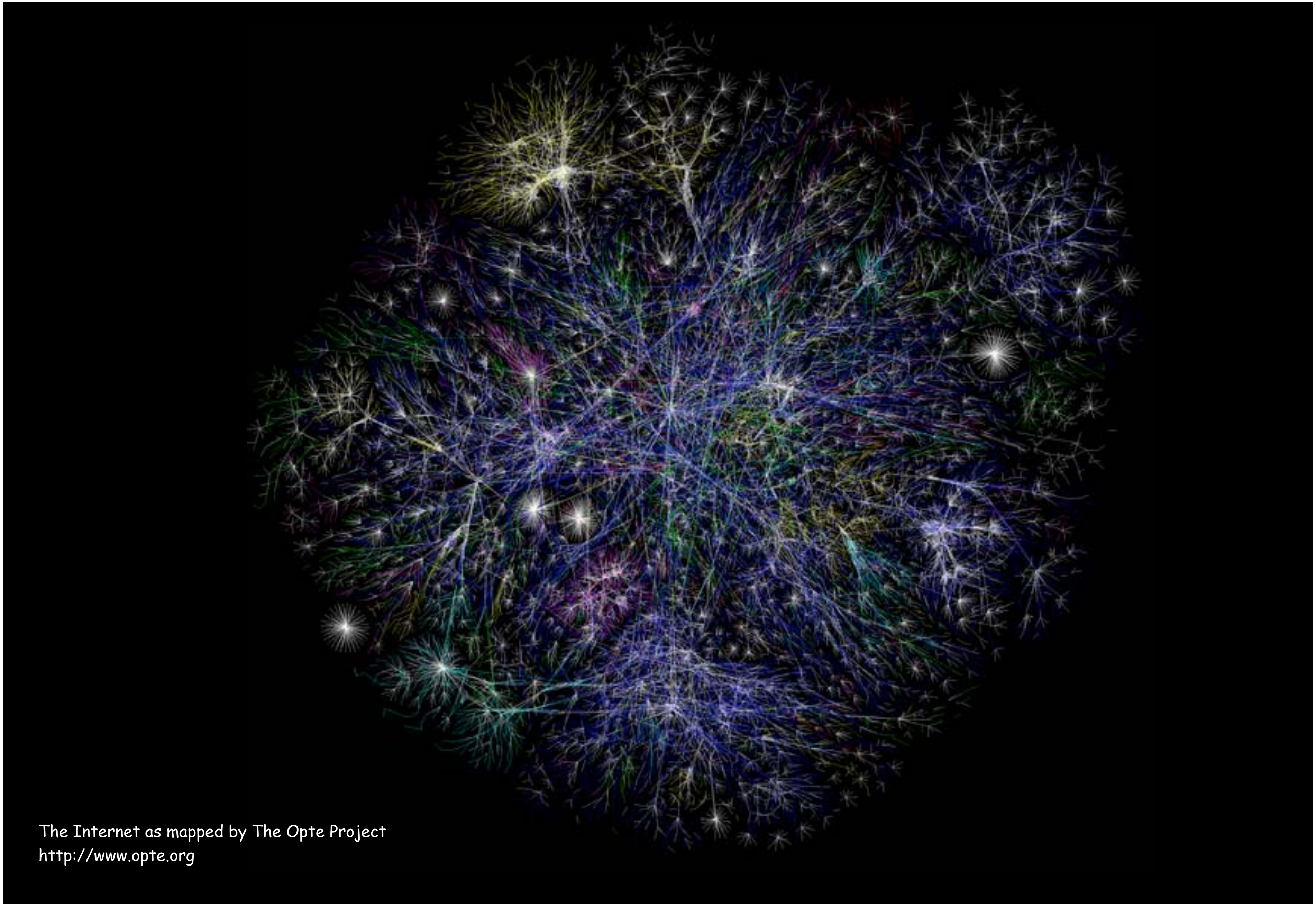
Reference: Duncan Watts

Protein interaction network



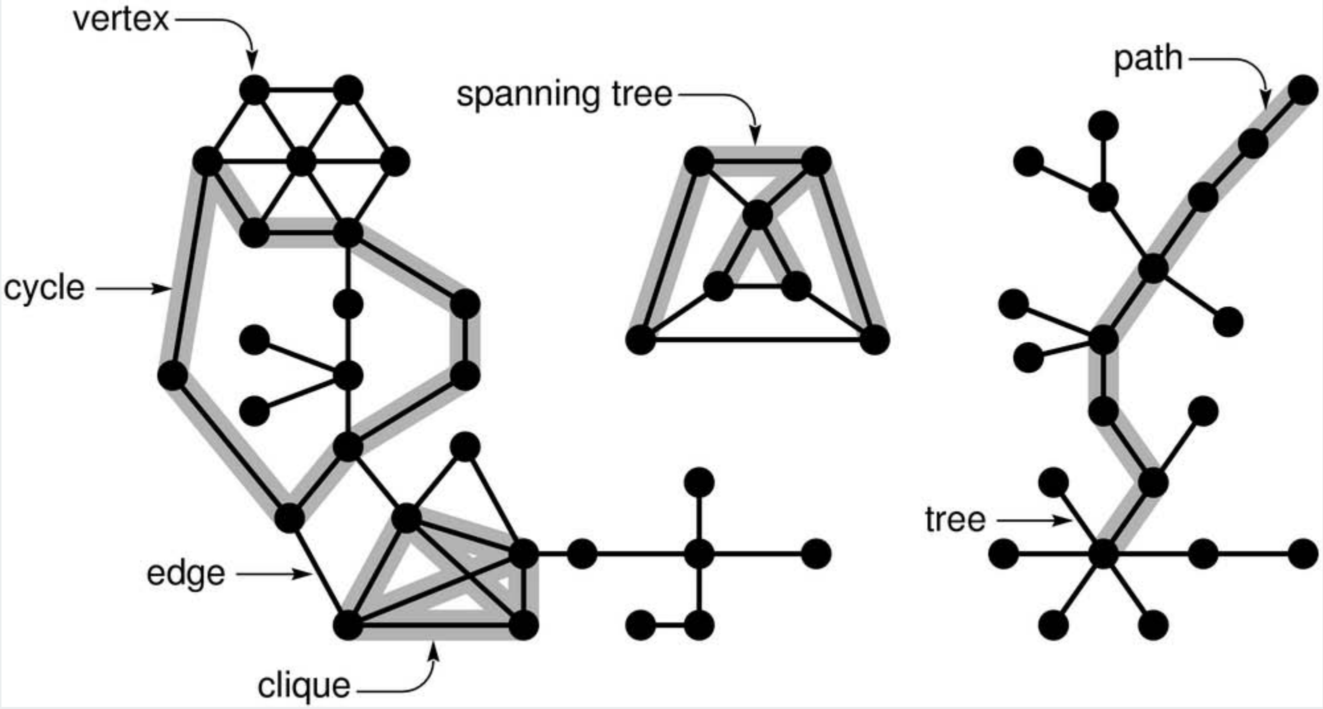
Reference: Jeong et al, Nature Review | Genetics

The Internet



The Internet as mapped by The Opte Project
http://www.opte.org

Graph terminology



Some graph-processing problems

Path. Is there a path between s to t?
Shortest path. What is the shortest path between s and t?
Longest path. What is the longest simple path between s and t?

Cycle. Is there a cycle in the graph?
Euler tour. Is there a cycle that uses each edge exactly once?
Hamilton tour. Is there a cycle that uses each vertex exactly once?

Connectivity. Is there a way to connect all of the vertices?
MST. What is the best way to connect all of the vertices?
Biconnectivity. Is there a vertex whose removal disconnects the graph?

Planarity. Can you draw the graph in the plane with no crossing edges?

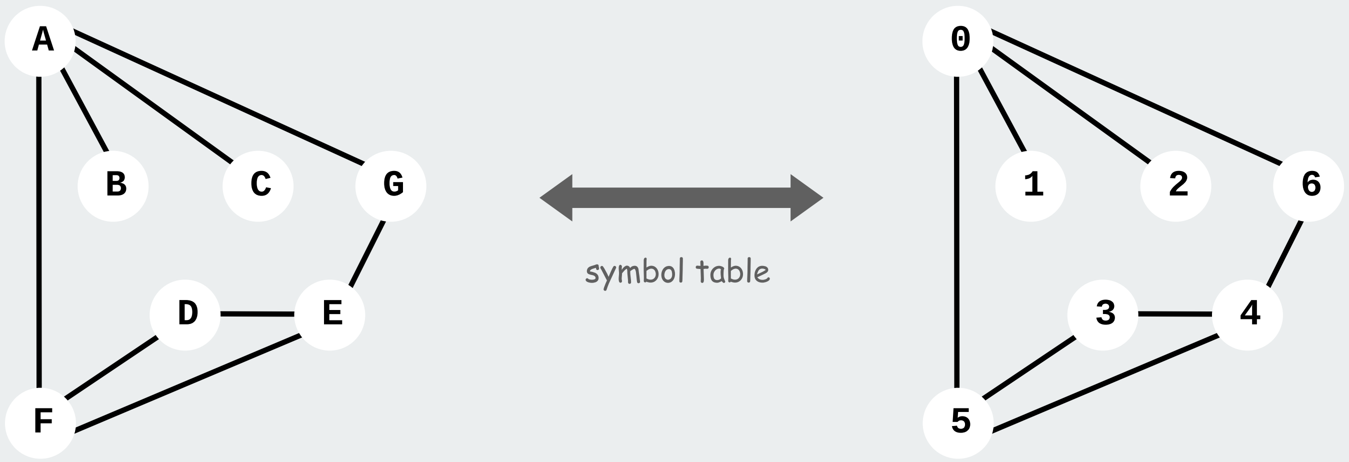
First challenge: Which of these problems is easy? difficult? intractable?

- ‣ **Graph API**
- ‣ maze exploration
- ‣ depth-first search
- ‣ breadth-first search
- ‣ connected components
- ‣ challenges

Graph representation

Vertex representation.

- This lecture: use integers between `0` and `V-1`.
- Real world: convert between names and integers with symbol table.



Other issues. Parallel edges, self-loops.

Graph API

public class Graph		(graph data type)
	`Graph(int V)`	create an empty graph with V vertices
	`Graph(int V, int E)`	create a random graph with V vertices, E edges
`void`	`addEdge(int v, int w)`	add an edge v-w
`Iterable<Integer>`	`adj(int v)`	return an iterator over the neighbors of v
`int`	`V()`	return number of vertices
`String`	`toString()`	return a string representation

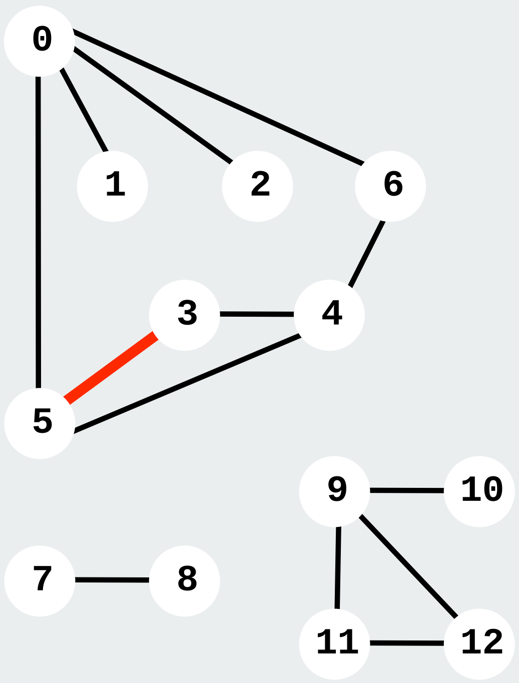
Client that iterates through all edges

```
Graph G = new Graph(V, E);
StdOut.println(G);
for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      // process edge v-w
```

processes BOTH
v-w and w-v

Set of edges representation

Store a list of the edges (linked list or array)

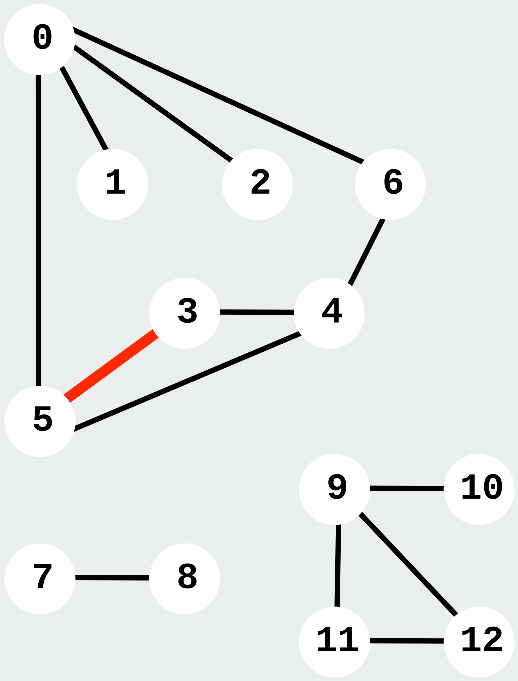


```
0-1
0-6
0-2
11-12
9-12
9-11
9-10
4-3
5-3
7-8
5-4
0-5
6-4
```

Adjacency matrix representation

Maintain a two-dimensional **V** × **V** boolean array.

For each edge **v-w** in graph: **`adj[v][w] = adj[w][v] = true`**.



two entries for each edge

$$
\begin{array}{r|ccccccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}
$$

Adjacency-matrix graph representation: Java implementation

```
public class Graph
{
   private int V;
   private boolean[][] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = new boolean[V][V];
   }

   public void addEdge(int v, int w)
   {
      adj[v][w] = true;
      adj[w][v] = true;
   }

   public Iterable<Integer> adj(int v)
   {
      return new AdjIterator(v);
   }
}
```

adjacency
matrix

create empty
V-vertex graph

add edge v-w
(no parallel edges)

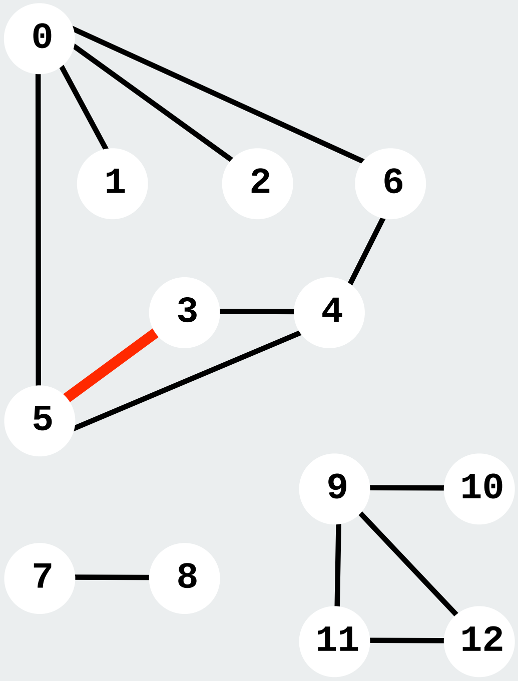
iterator for
v's neighbors

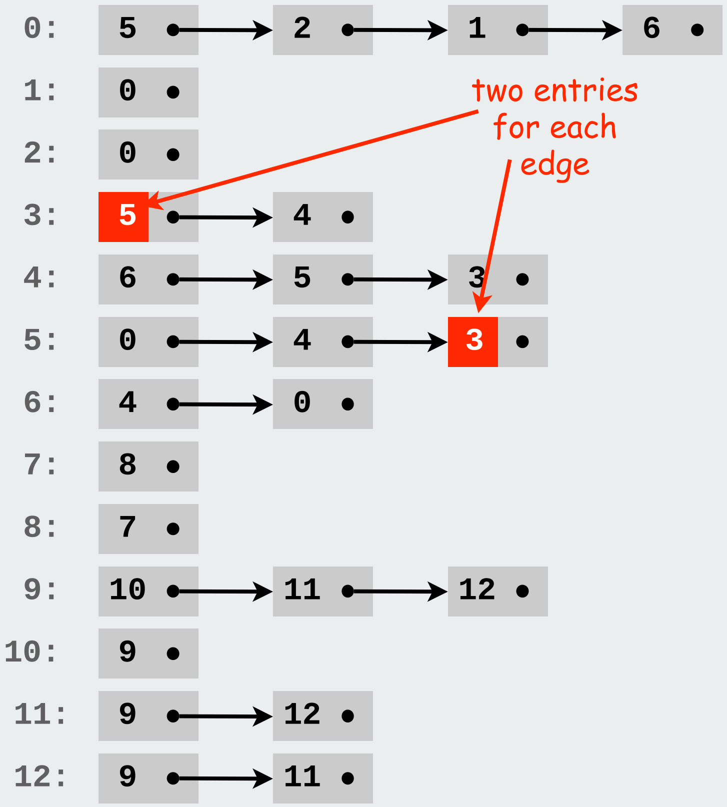
Adjacency matrix: iterator for vertex neighbors

```
private class AdjIterator implements Iterator<Integer>,
                                     Iterable<Integer>
{
   int v, w = 0;
   AdjIterator(int v)
   {  this.v = v;  }

   public boolean hasNext()
   {
      while (w < V)
      {  if (adj[v][w]) return true; w++ }
      return false;
   }

   public int next()
   {
      if (hasNext()) return w++ ;
      else throw new NoSuchElementException();
   }

   public Iterator<Integer> iterator()
   { return this; }

}
```

Adjacency-list graph representation

Maintain vertex-indexed array of lists (implementation omitted)





0: 5 → 2 → 1 → 6

1: 0

2: 0

3: 5 → 4

4: 6 → 5 → 3

5: 0 → 4 → 3

6: 4 → 0

7: 8

8: 7

9: 10 → 11 → 12

10: 9

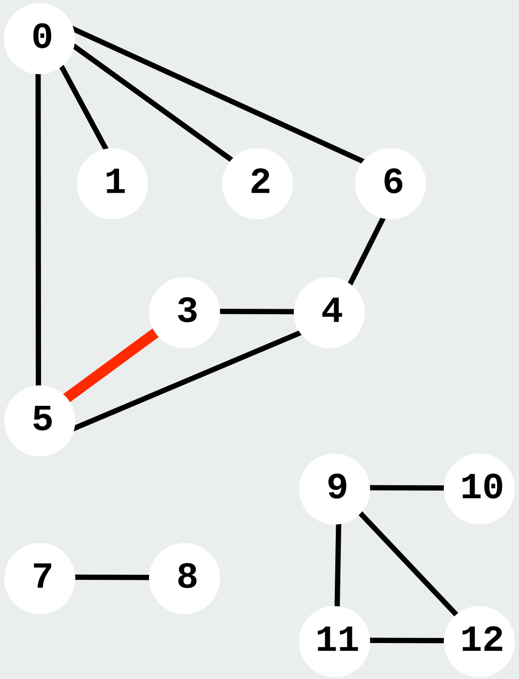
11: 9 → 12

12: 9 → 11

two entries for each edge

Adjacency-SET graph representation

Maintain vertex-indexed array of SETs
(take advantage of balanced-tree or hashing implementations)



```
 0:  { 1  2  5  6 }
 1:  { 0 }
 2:  { 0 }
 3:  { 4  5 }
 4:  { 3  5  6 }
 5:  { 0  3  4 }
 6:  { 0  4 }
 7:  { 8 }
 8:  { 7 }
 9:  { 10  11  12 }
10:  { 9 }
11:  { 9  12 }
12:  { 9  1 }
```

two entries
for each
edge

Adjacency-SET graph representation: Java implementation

```
public class Graph
{
   private int V;
   private SET<Integer>[] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = (SET<Integer>[]) new SET[V];
      for (int v = 0; v < V; v++)
         adj[v] = new SET<Integer>();
   }


   public void addEdge(int v, int w)
   {
      adj[v].add(w);
      adj[w].add(v);
   }

   public Iterable<Integer> adj(int v)
   {
      return adj[v];
   }
}
```

adjacency
sets

create empty
V-vertex graph

add edge v-w
(no parallel edges)

iterable SET for
v's neighbors

Graph representations

Graphs are abstract mathematical objects, BUT

- ADT implementation requires specific representation.
- Efficiency depends on matching algorithms to representations.

representation	space	edge between v and w?	iterate over edges incident to v?
list of edges	E	E	E
adjacency matrix	V^2	1	V
adjacency list	E + V	degree(v)	degree(v)
adjacency SET	E + V	log (degree(v))	degree(v)*

* easy to also support ordered iteration and randomized iteration

In practice: Use adjacency SET representation

- Take advantage of proven technology
- Real-world graphs tend to be "sparse"

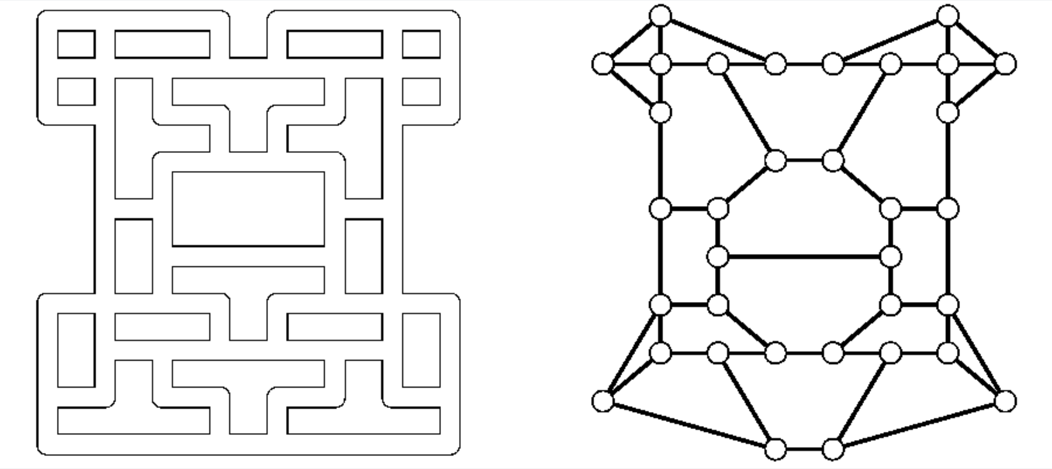
  [huge number of vertices, small average vertex degree]
- Algs all based on iterating over edges incident to v.

- Graph API
- **maze exploration**
- depth-first search
- breadth-first search
- connected components
- challenges

Maze exploration

Maze graphs.

- Vertex = intersections.
- Edge = passage.



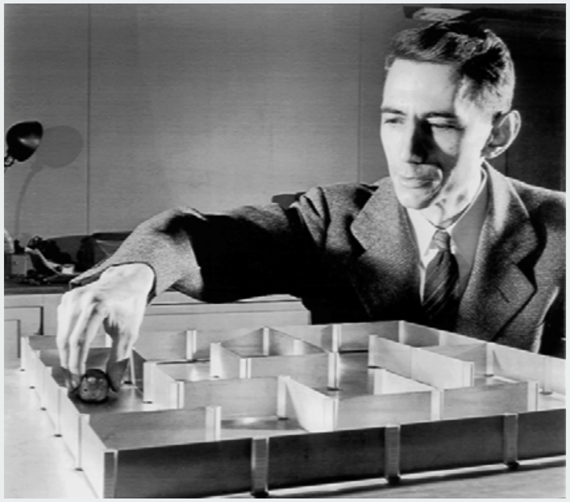
Goal. Explore every passage in the maze.

Trémaux Maze Exploration

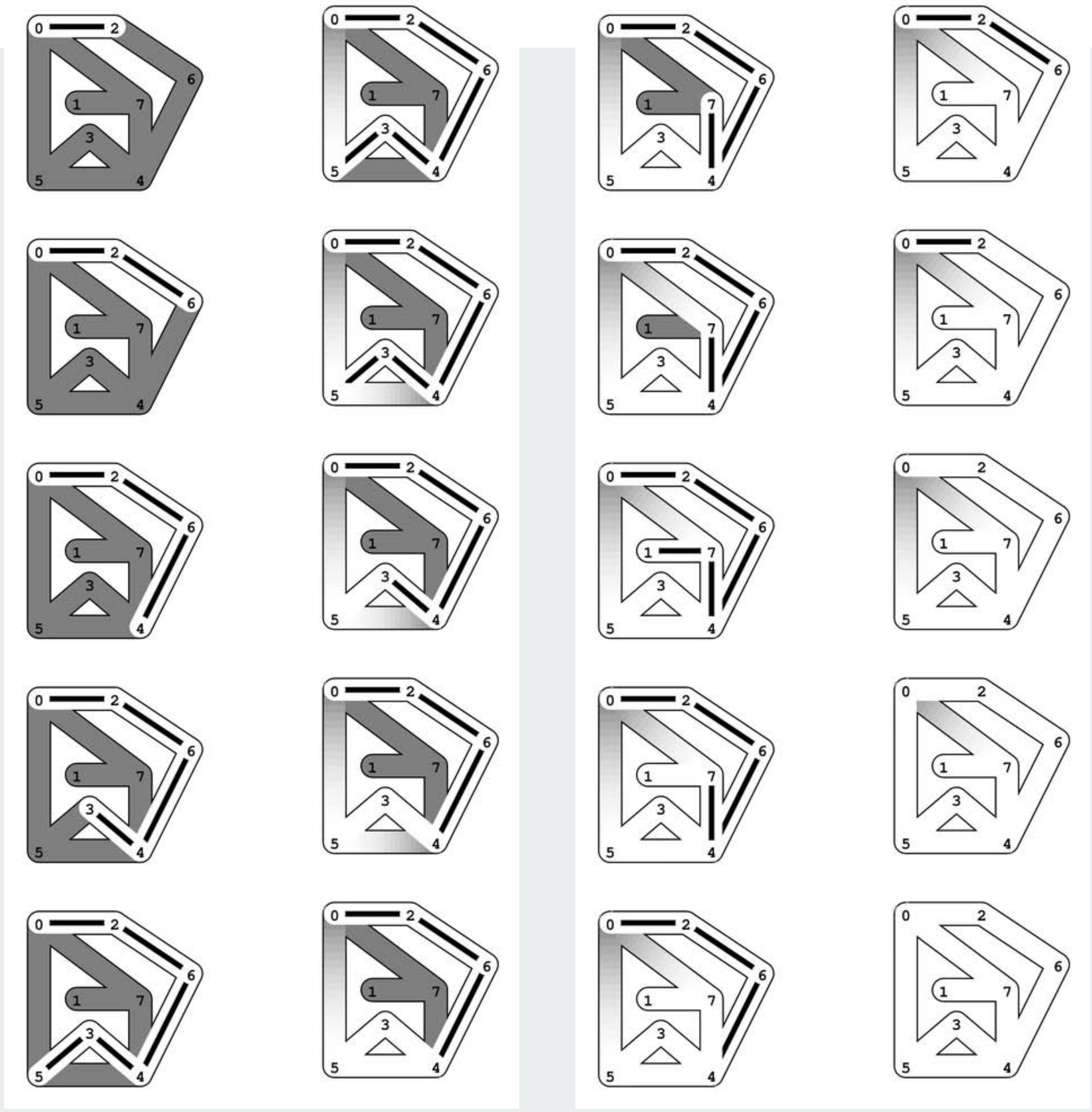
Trémaux maze exploration.

- Unroll a ball of string behind you.
- Mark each visited intersection by turning on a light.
- Mark each visited passage by opening a door.

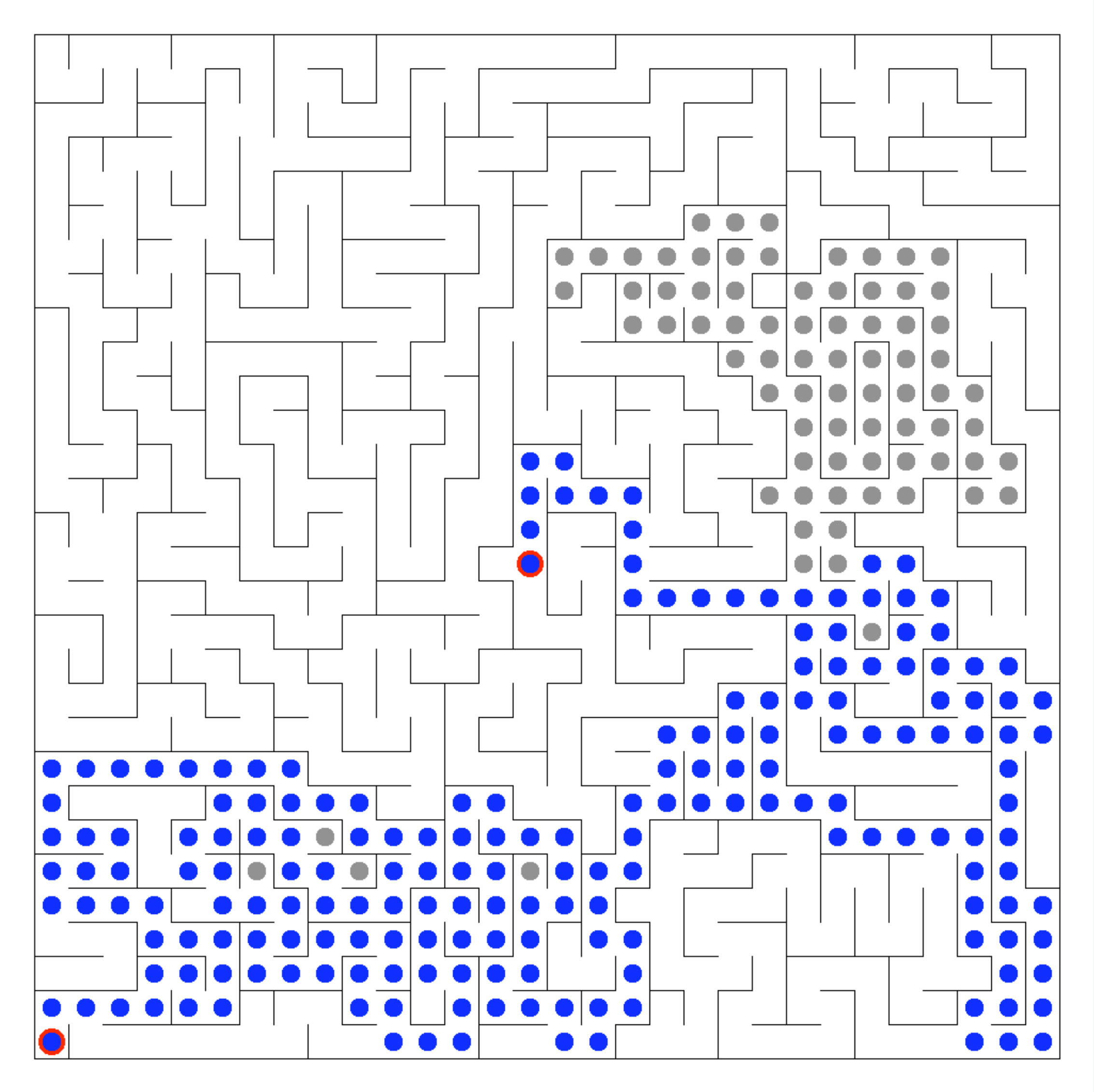
First use? Theseus entered labyrinth to kill the monstrous Minotaur; Ariadne held ball of string.



Claude Shannon (with Theseus mouse)


0
2
6
1
7
3
5
4

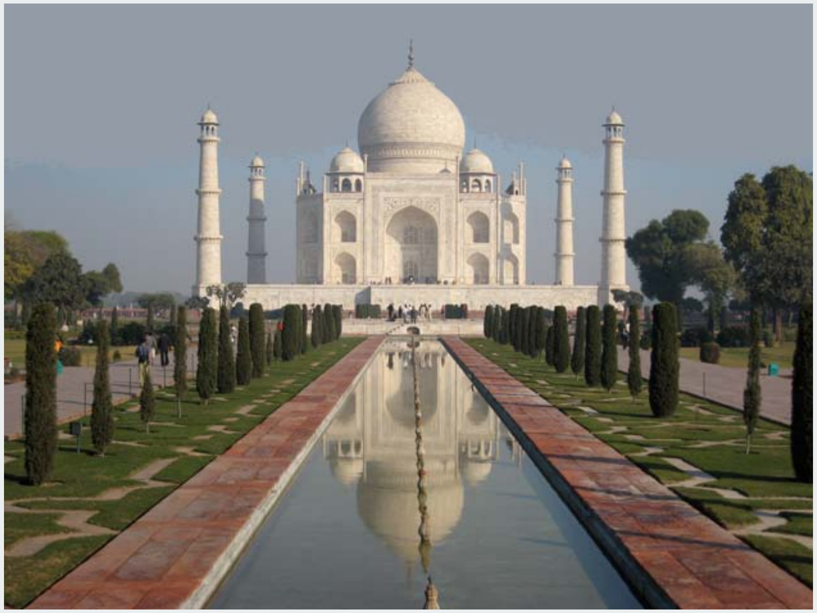
Maze Exploration

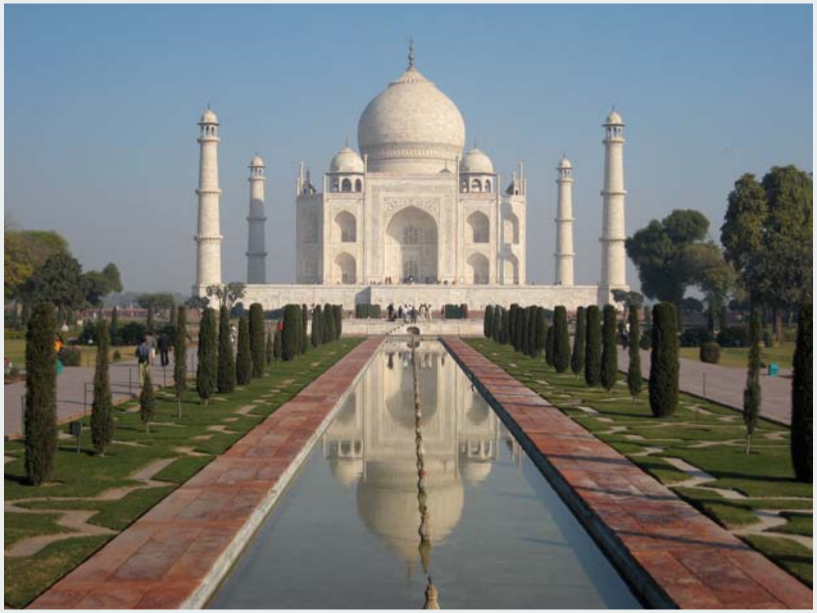


- Graph API
- maze exploration
- **depth-first search**
- breadth-first search
- connected components
- challenges

Flood fill

Photoshop "magic wand"





Graph-processing challenge 1:

Problem: Flood fill

Assumptions: picture has millions to billions of pixels

How difficult?

1) any COS126 student could do it
2) need to be a typical diligent COS226 student
3) hire an expert
4) intractable
5) no one knows

Depth-first search

Goal. Systematically search through a graph.

Idea. Mimic maze exploration.

Typical applications.

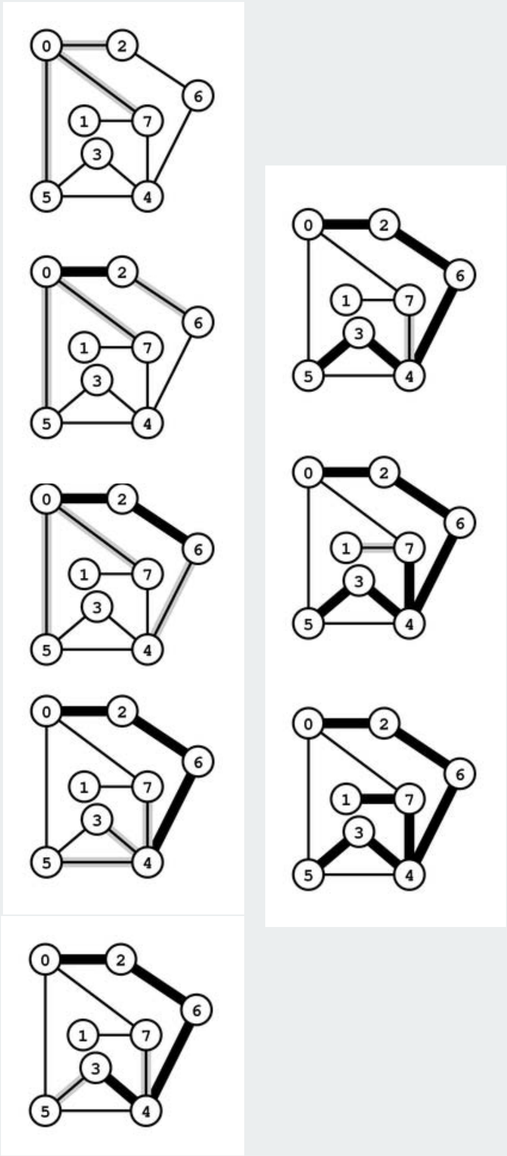
- find all vertices connected to a given s
- find a path from s to t

```
DFS (to visit a vertex s)
Mark s as visited.
Visit all unmarked vertices v adjacent to s.
```

recursive

Running time.

- O(E) since each edge examined at most twice
- usually less than V to find paths in real graphs



Design pattern for graph processing

Typical client program.

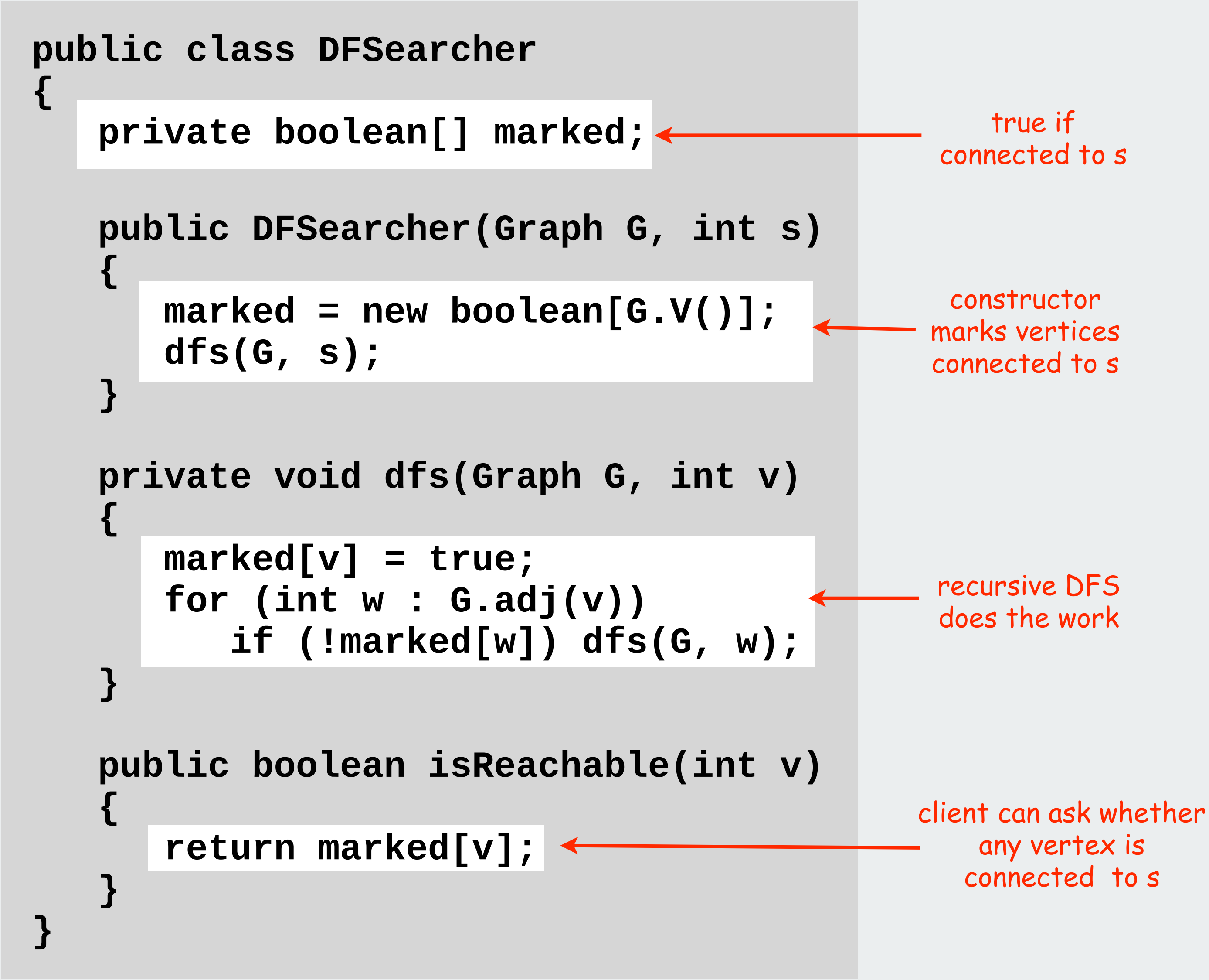
- Create a **`Graph`**.
- Pass the **`Graph`** to a graph-processing routine, e.g., **`DFSearcher`**.
- Query the graph-processing routine for information.

Client that prints all vertices connected to (reachable from) s

```
public static void main(String[] args)
{
   In in = new In(args[0]);
   Graph G = new Graph(in);
   int s = 0;
   DFSearcher dfs = new DFSearcher(G, s);
   for (int v = 0; v < G.V(); v++)
      if (dfs.isConnected(v))
         System.out.println(v);
}
```

Decouple graph from graph processing.

Depth-first search (connectivity)



```
public class DFSearcher
{
   private boolean[] marked;

   public DFSearcher(Graph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
   }

   public boolean isReachable(int v)
   {
      return marked[v];
   }
}
```

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

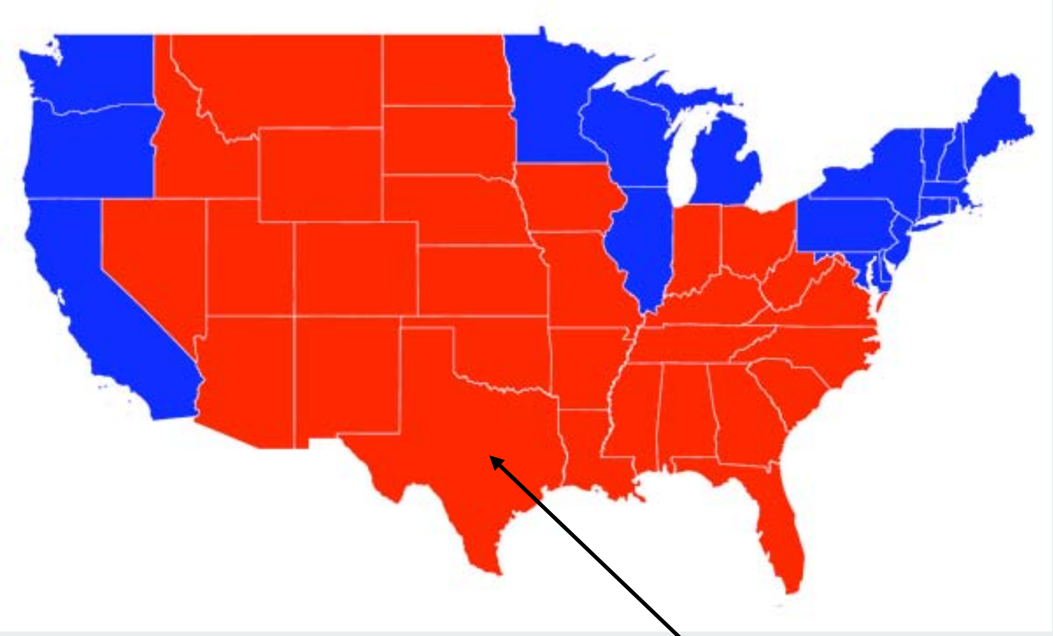
client can ask whether any vertex is connected to s

Connectivity application: Flood fill

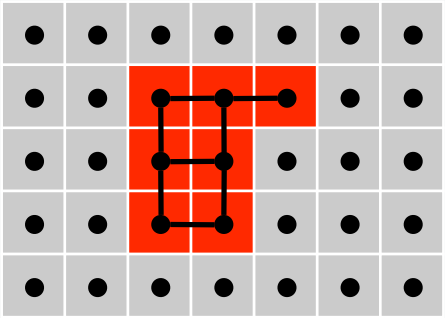
Change color of entire blob of neighboring red pixels to blue.

Build a grid graph

- vertex: pixel.
- edge: between two adjacent lime pixels.
- blob: all pixels connected to given pixel.



recolor red blob to blue

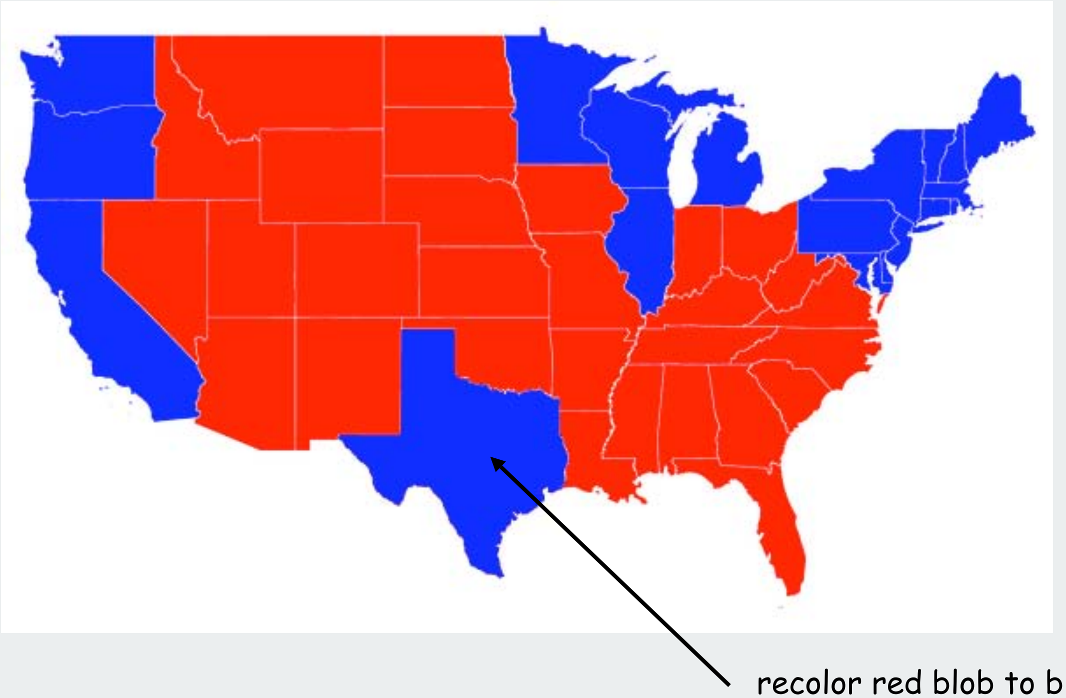


Connectivity Application: Flood Fill

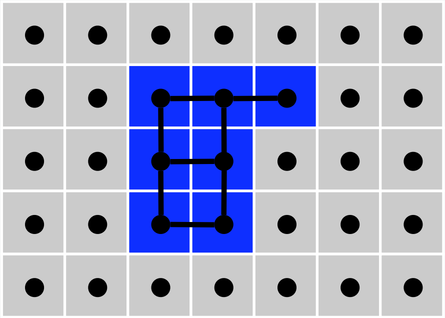
Change color of entire blob of neighboring red pixels to blue.

Build a grid graph

- vertex: pixel.
- edge: between two adjacent red pixels.
- blob: all pixels connected to given pixel.

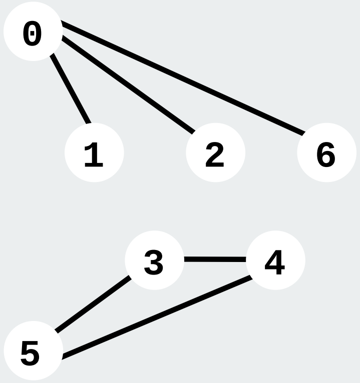


recolor red blob to blue



Graph-processing challenge 2:

Problem: Is there a path from s to t ?



0-1
0-6
0-2
4-3
5-3
5-4

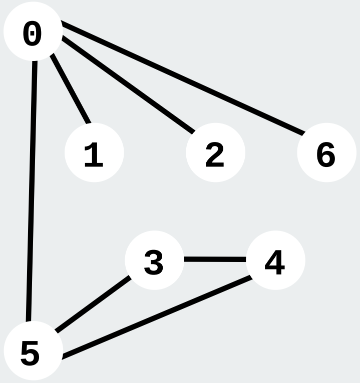
How difficult?

1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows

Graph-processing challenge 3:

Problem: Find a path from s to t.

Assumptions: any path will do



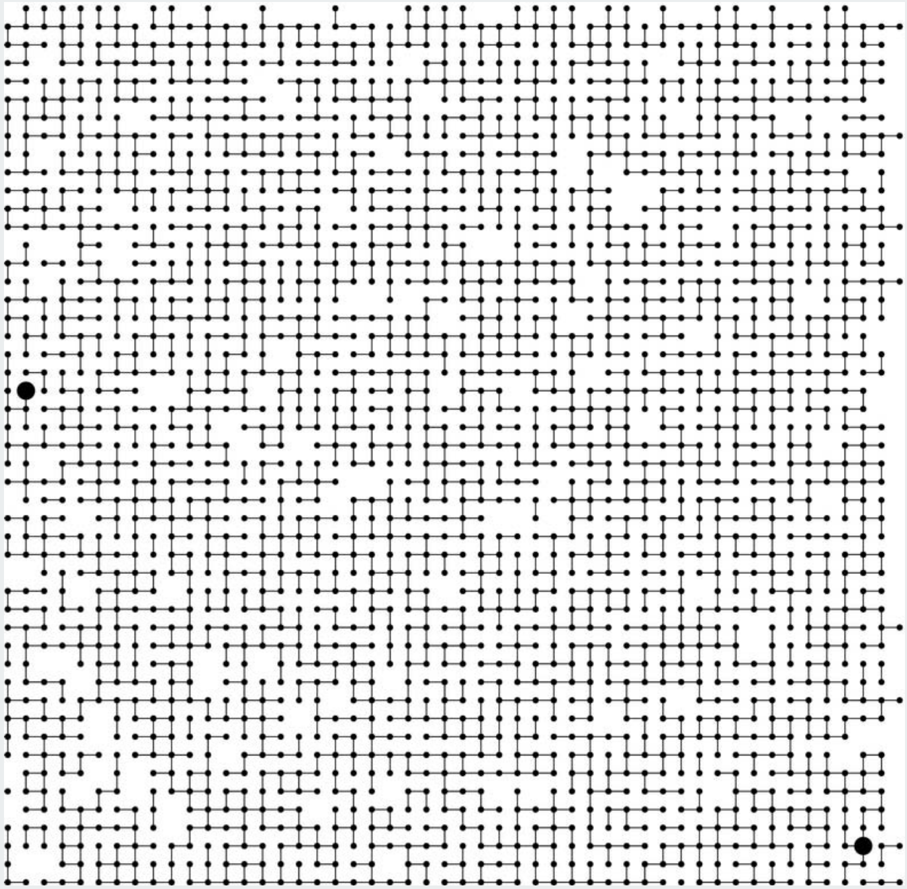
0-1
0-6
0-2
4-3
5-3
5-4
0-5

How difficult?

1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows

Paths in graphs

Is there a path from s to t? If so, find one.



Paths in graphs

Is there a path from s to t?

method	preprocess time	query time	space
Union Find	V + E log* V	log* V †	V
DFS	E + V	1	E + V

† amortized

If so, find one.

- Union-Find: no help (use DFS on connected subgraph)
- DFS: easy (stay tuned)

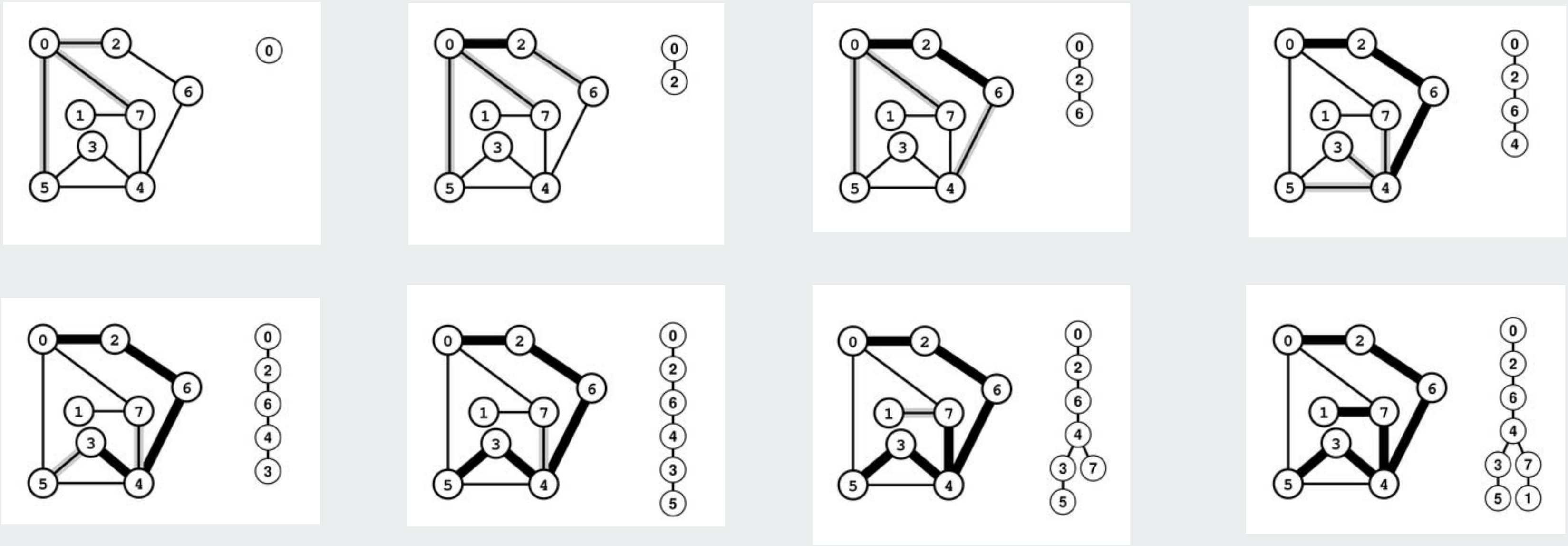
UF advantage. Can intermix queries and edge insertions.

DFS advantage. Can recover path itself in time proportional to its length.

Keeping track of paths with DFS

DFS tree. Upon visiting a vertex v for the first time, remember that you came from `pred[v]` (parent-link representation).

Retrace path. To find path between s and v, follow `pred` back from v.



Depth-first-search (pathfinding)

```
public class DFSearcher
{
   ...
   private int[] pred;
   public DFSearcher(Graph G, int s)
   {
      ...
      pred = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
         pred[v] = -1;
      ...
   }
   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w])
         {
            pred[w] = v;
            dfs(G, w);
         }
   }

   public Iterable<Integer> path(int v)
   {  // next slide  }
}
```

add instance variable for
parent-link representation
of DFS tree

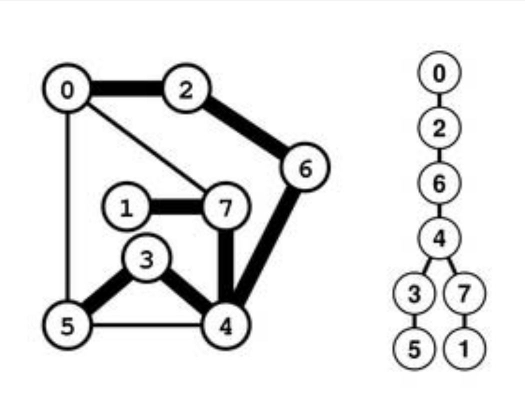
initialize it in the
constructor

set parent link

add method for client
to iterate through path

Depth-first-search (pathfinding iterator)

```
   public Iterable<Integer> path(int v)
   {
      Stack<Integer> path = new Stack<Integer>();
      while (v != -1 && marked[v])
      {
         list.push(v);
         v = pred[v];
      }
      return path;
   }
}
```



DFS summary

Enables direct solution of simple graph problems.

- Find path from s to t. ✓
- Connected components (stay tuned).
- Euler tour (see book).
- Cycle detection (simple exercise).
- Bipartiteness checking (see book).

Basis for solving more difficult graph problems.

- Biconnected components (see book).
- Planarity testing (beyond scope).

- Graph API
- maze exploration
- depth-first search
- **breadth-first search**
- connected components
- challenges

Breadth First Search

Depth-first search. Put unvisited vertices on a stack.
Breadth-first search. Put unvisited vertices on a queue.

Shortest path. Find path from s to t that uses fewest number of edges.

```
BFS (from source vertex s)
---------------------------------------------------
Put s onto a FIFO queue.
Repeat until the queue is empty:
    ▪ remove the least recently added vertex v
    ▪ add each of v's unvisited neighbors to the queue,
      and mark them as visited.
---------------------------------------------------
```

Property. BFS examines vertices in increasing distance from s.

Breadth-first search scaffolding

```
public class BFSearcher
{
   private int[] dist;

   public BFSearcher(Graph G, int s)
   {
      dist = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
         dist[v] = G.V() + 1;
      dist[s] = 0;

      bfs(G, s);
   }

   public int distance(int v)
   {  return dist[v];   }

   private void bfs(Graph G, int s)
   {  // See next slide.    }

}
```

distances from s

initialize distances

compute
distances

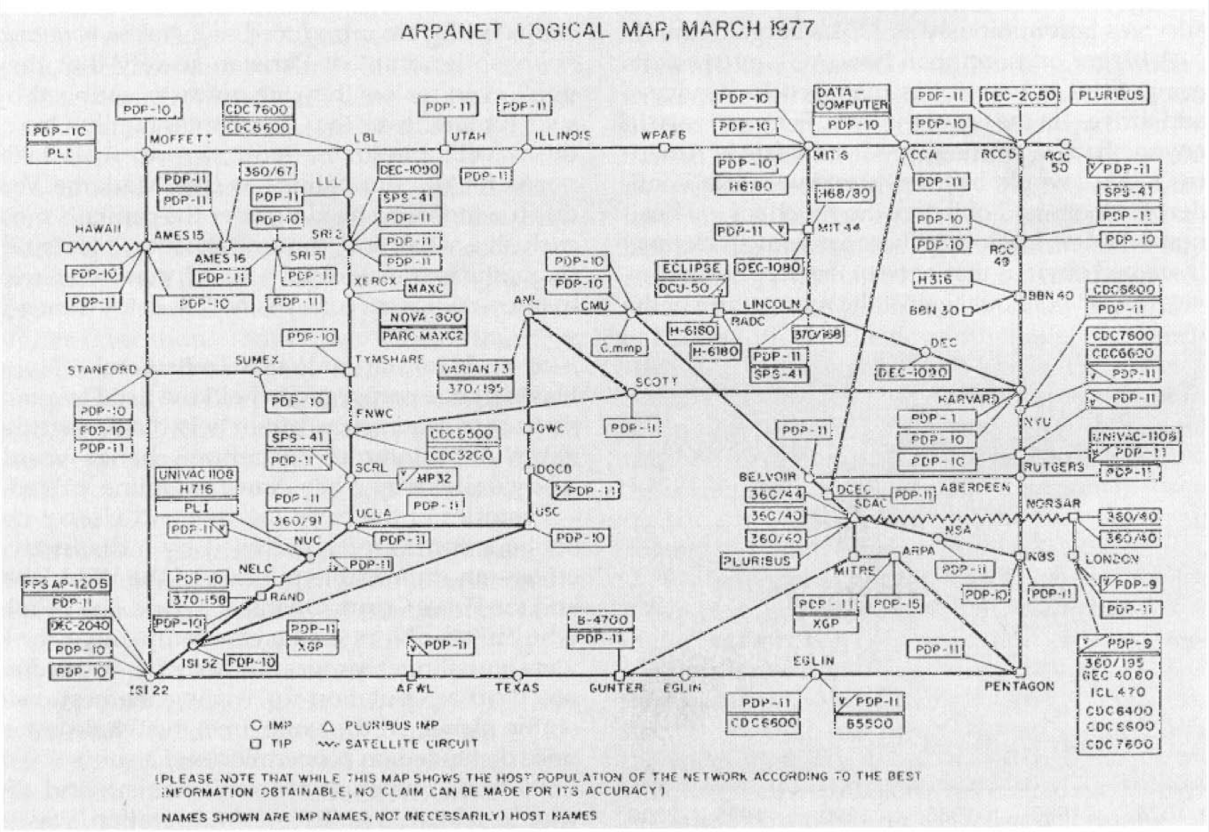
answer client
query

Breadth-first search (compute shortest-path distances)

```
private void bfs(Graph G, int s)
{
   Queue<Integer> q = new Queue<Integer>();
   q.enqueue(s);
   while (!q.isEmpty())
   {
      int v = q.dequeue();
      for (int w : G.adj(v))
      {
         if (dist[w] > G.V())
         {
            q.enqueue(w);
            dist[w] = dist[v] + 1;
         }
      }
   }
}
```

BFS Application

- Kevin Bacon numbers.
- Facebook.
- Fewest number of hops in a communication network.



ARPANET LOGICAL MAP, MARCH 1977

○ IMP   △ PLURIBUS IMP
□ TIP   ﹏ SATELLITE CIRCUIT

(PLEASE NOTE THAT WHILE THIS MAP SHOWS THE HOST POPULATION OF THE NETWORK ACCORDING TO THE BEST INFORMATION OBTAINABLE, NO CLAIM CAN BE MADE FOR ITS ACCURACY)

NAMES SHOWN ARE IMP NAMES, NOT (NECESSARILY) HOST NAMES

ARPANET

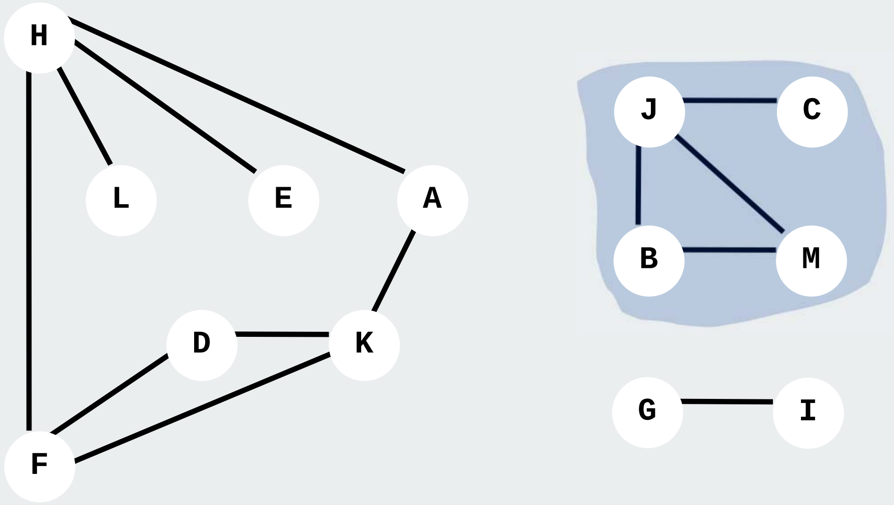
- Graph API
- maze exploration
- depth-first search
- breadth-first search
- **connected components**
- challenges

Connectivity Queries

Def. Vertices v and w are connected if there is a path between them.

Def. A connected component is a maximal set of connected vertices.

Goal. Preprocess graph to answer queries: is v connected to w?
in constant time



Vertex	Component
A	0
B	1
C	1
D	0
E	0
F	0
G	2
H	0
I	2
J	1
K	0
L	0
M	1

Union-Find? not quite

Connected Components

Goal. Partition vertices into connected components.

Connected components

Initialize all vertices **v** as unmarked.

For each unmarked vertex **v**, run DFS and identify all vertices discovered as part of the same connected component.

preprocess Time	query Time	extra Space
E + V	1	V

Depth-first search for connected components

```
public class CCFinder
{
   private final static int UNMARKED = -1;
   private int components;
   private int[] cc;
   public CCFinder(Graph G)
   {
      cc = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
         cc[v] = UNMARKED;
      for (int v = 0; v < G.V(); v++)
         if (cc[v] == UNMARKED)
            { dfs(G, v); components++; }
   }
   private void dfs(Graph G, int v)
   {
      cc[v] = components;
      for (int w : G.adj(v))
         if (cc[w] == UNMARKED) dfs(G, w);
   }

   public int connected(int v, int w)
   {  return cc[v] == cc[w];  }

}
```

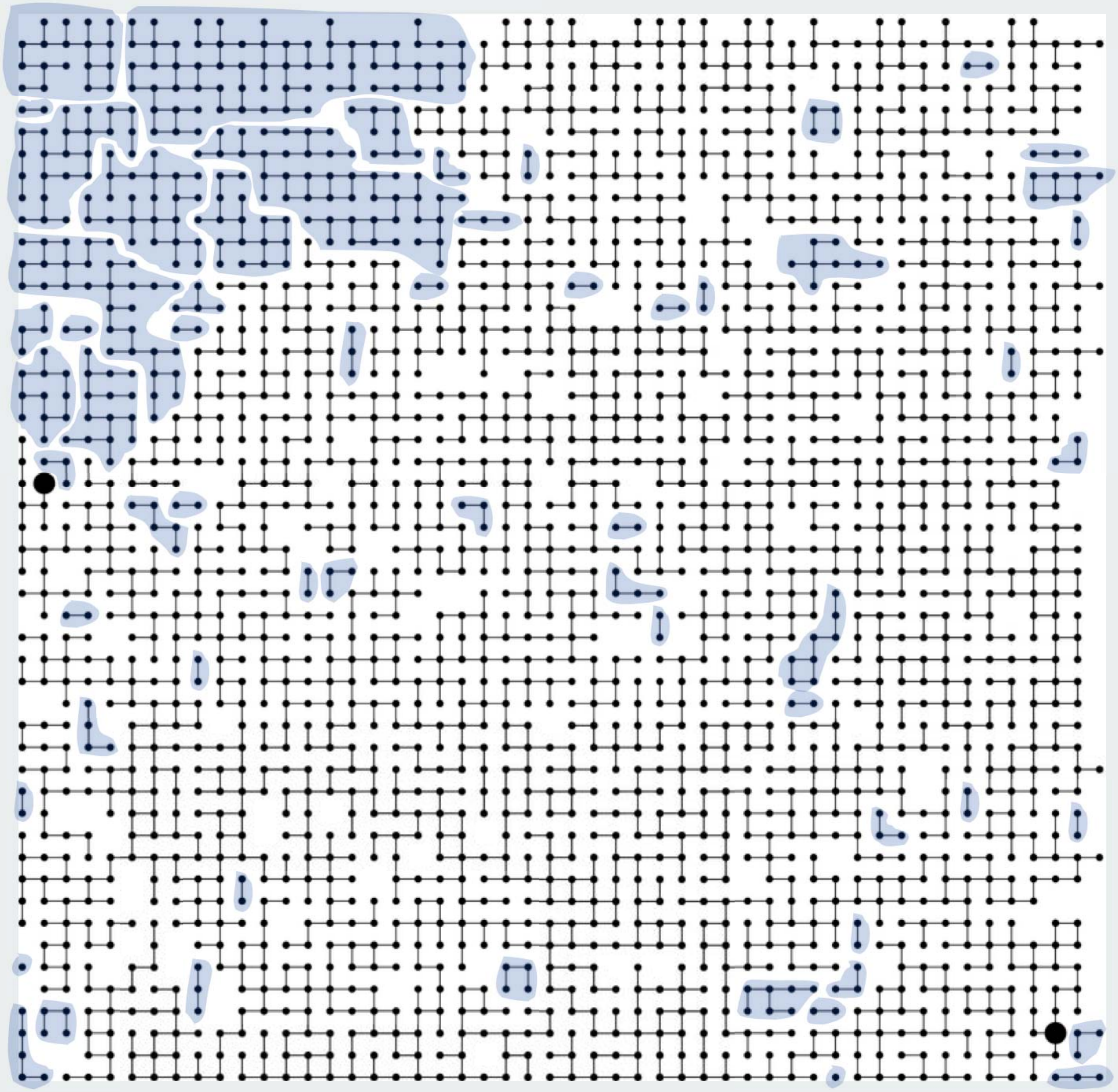
component labels

DFS for each component

standard DFS

constant-time connectivity query

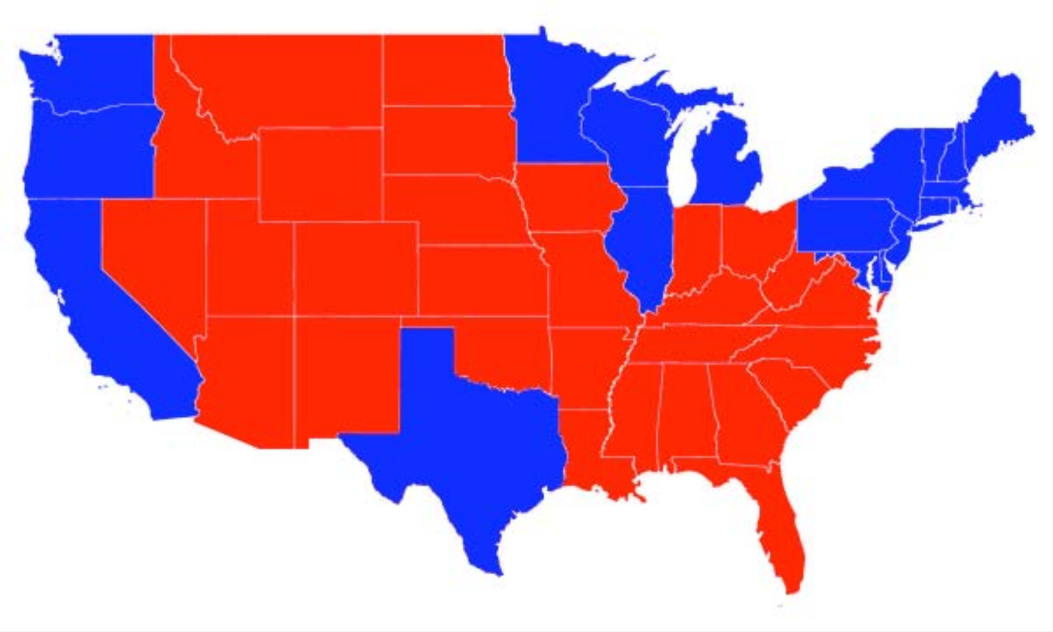
Connected Components



63 components

Connected components application: Image processing

Goal. Read in a 2D color image and find regions of connected pixels that have the same color.



Input: scanned image

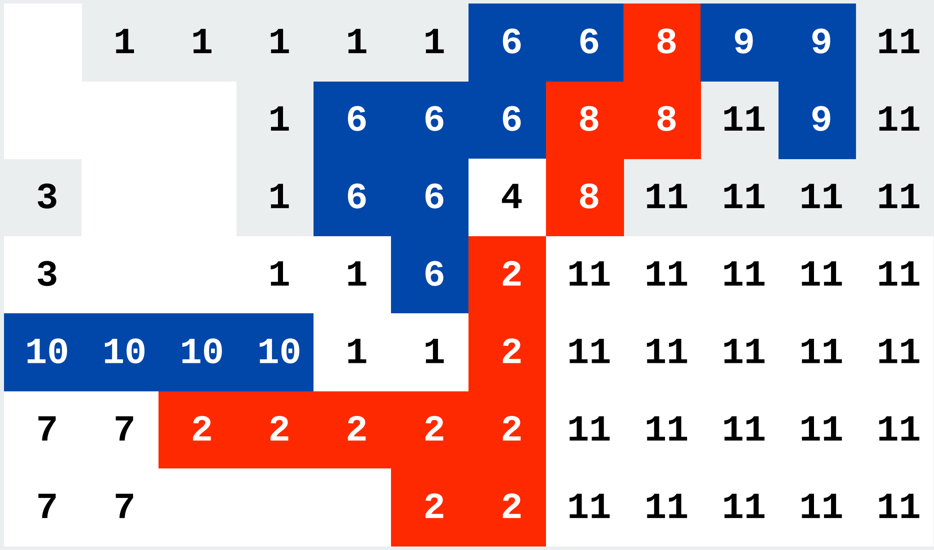
Output: number of red and blue states

Connected components application: Image Processing

Goal. Read in a 2D color image and find regions of connected pixels that have the same color.

Efficient algorithm.

- Connect each pixel to neighboring pixel if same color.
- Find connected components in resulting graph.



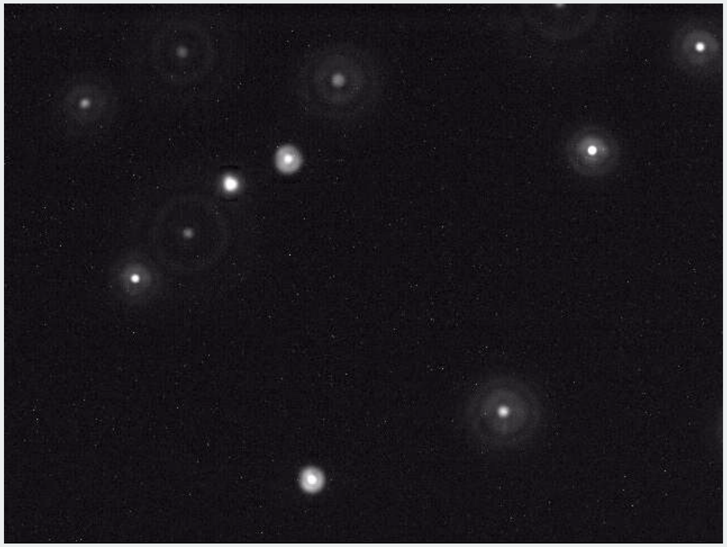
	1	1	1	1	1	6	6	8	9	9	11
			1	6	6	6	8	8	11	9	11
3			1	6	6	4	8	11	11	11	11
3			1	1	6	2	11	11	11	11	11
10	10	10	10	1	1	2	11	11	11	11	11
7	7	2	2	2	2	2	11	11	11	11	11
7	7				2	2	11	11	11	11	11

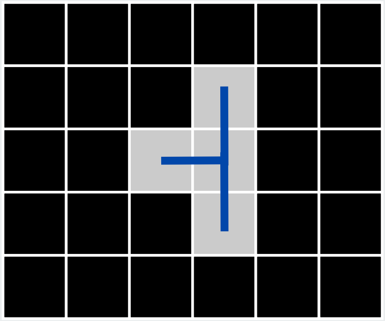
Connected components application: Particle detection

Particle detection. Given grayscale image of particles, identify "blobs."

- Vertex: pixel.
- Edge: between two adjacent pixels with grayscale value ≥ 70.
- Blob: connected component of 20-30 pixels.

black = 0
white = 255





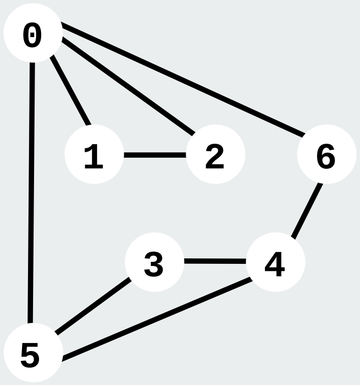
Particle tracking. Track moving particles over time.

- Graph API
- maze exploration
- depth-first search
- breadth-first search
- connected components
- **challenges**

Graph-processing challenge 4:

Problem: Find a path from s to t

Assumptions: any path will do



```
0-1
0-6
0-2
4-3
5-3
5-4
0-5
6-4
1-2
5-0
```

Which is faster, DFS or BFS?

1) DFS

2) BFS

3) about the same

4) depends on the graph

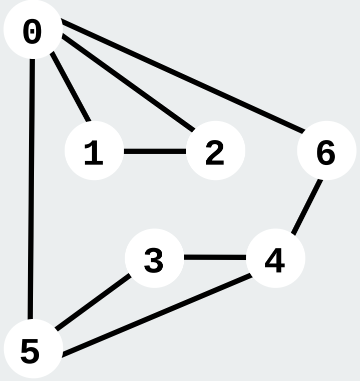
5) depends on the graph representation

Graph-processing challenge 5:

Problem: Find a path from s to t

Assumptions: any path will do

randomized iterators



```
0-1
0-6
0-2
4-3
5-3
5-4
0-5
6-4
1-2
5-0
```

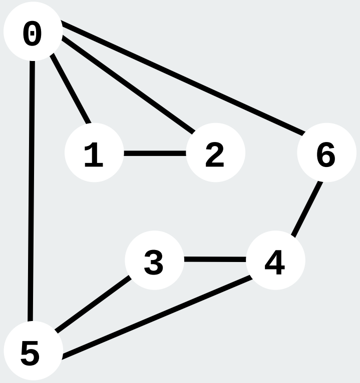
Which is faster, DFS or BFS?

1) DFS
2) BFS
3) about the same
4) depends on the graph
5) depends on the graph representation

Graph-processing challenge 6:

Problem: Find a path from s to t that uses every edge

Assumptions: need to use each edge exactly once



```
0-1
0-6
0-2
4-3
5-3
5-4
0-5
6-4
1-2
5-0
```

How difficult?

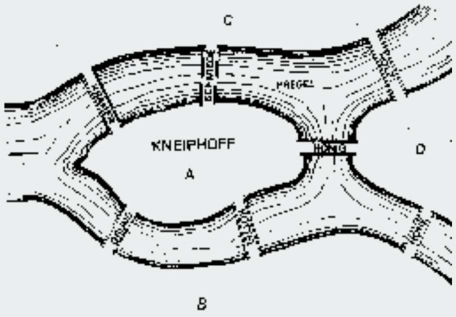
1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows

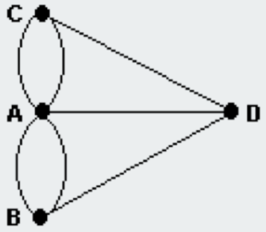
Bridges of Königsberg

earliest application of graph theory or topology

The Seven Bridges of Königsberg. [Leonhard Euler 1736]

> "... in Königsberg in Prussia, there is an island A, called the Kneiphof; the river which surrounds it is divided into two branches ... and these branches are crossed by seven bridges. Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he could cross each bridge once and only once..."





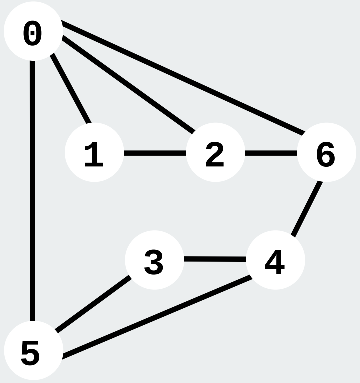
Euler tour. Is there a cyclic path that uses each edge exactly once?

Answer. Yes iff connected and all vertices have even degree.
Tricky DFS-based algorithm to find path (see Algs in Java).

Graph-processing challenge 7:

Problem: Find a path from s to t that visits every vertex

Assumptions: need to visit each vertex exactly once



```
0-1
0-6
0-2
4-3
5-3
5-4
0-5
6-4
1-2
2-6
```

How difficult?

1) any CS126 student could do it

2) need to be a typical diligent CS226 student

3) hire an expert

4) intractable

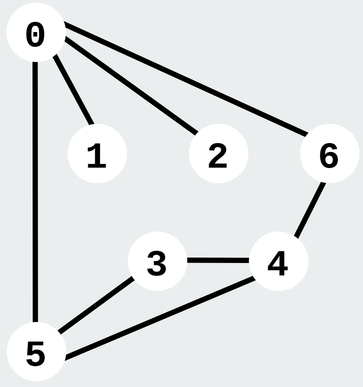
5) no one knows

Graph-processing challenge 8:

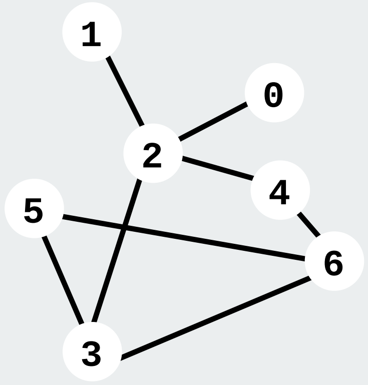
Problem: Are two graphs identical except for vertex names?

How difficult?

1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows



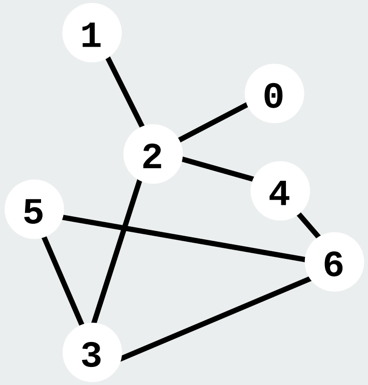
```
0-1
0-6
0-2
4-3
5-3
5-4
0-5
6-4
```



```
2-1
2-4
2-0
6-5
5-3
3-6
2-3
6-4
```

Graph-processing challenge 9:

Problem: Can you lay out a graph in the plane without crossing edges?



```
2-1
2-4
2-0
6-5
5-3
3-6
2-3
6-4
```

How difficult?

1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows

Directed Graphs

- digraph search
- transitive closure
- topological sort
- strong components

References:

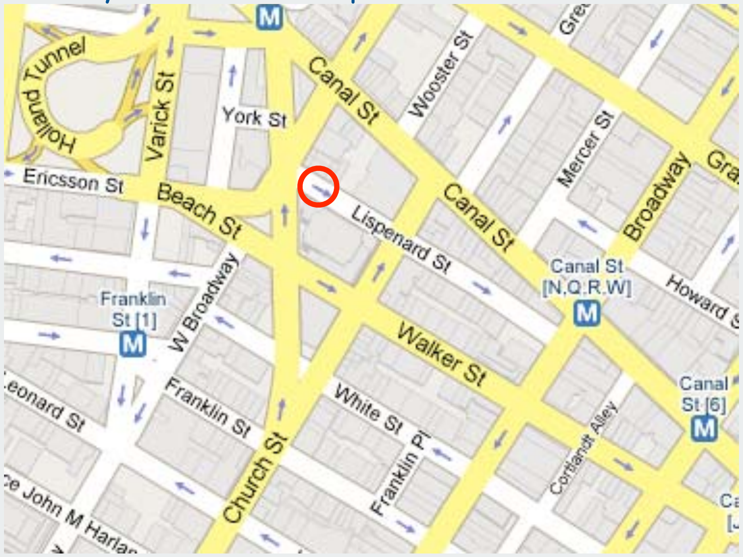
Algorithms in Java, Chapter 19

http://www.cs.princeton.edu/introalgsds/52directed

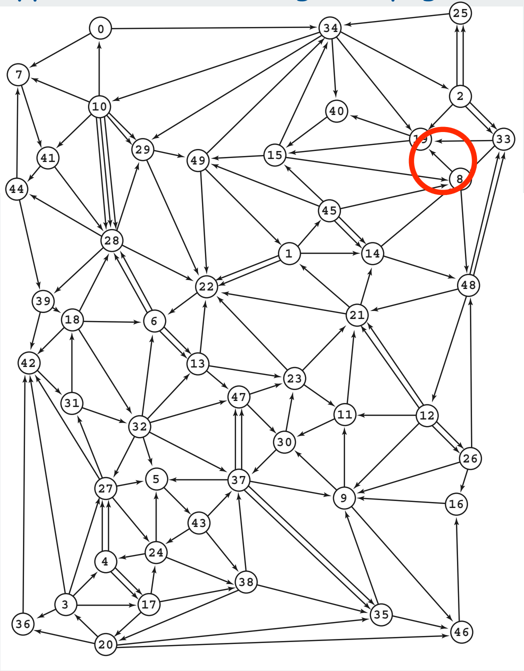
Directed graphs (digraphs)

Set of objects with oriented pairwise connections.

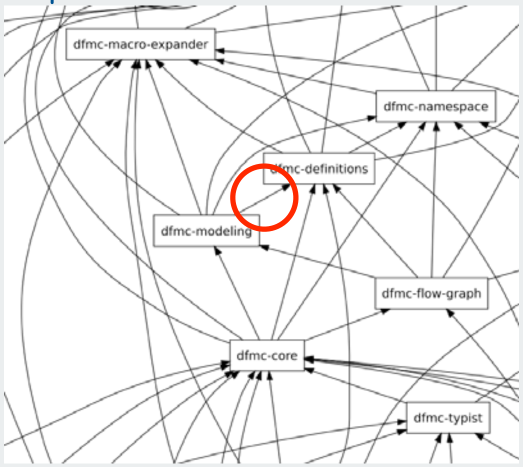
one-way streets in a map



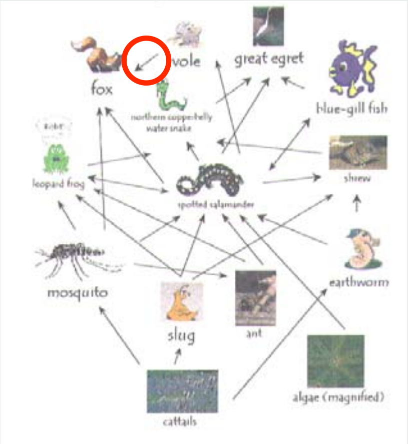
hyperlinks connecting web pages



dependencies in software modules



prey-predator relationships



Digraph applications

digraph	vertex	edge
financial	stock, currency	transaction
transportation	street intersection, airport	highway, airway route
scheduling	task	precedence constraint
WordNet	synset	hypernym
Web	web page	hyperlink
game	board position	legal move
telephone	person	placed call
food web	species	predator-prey relation
infectious disease	person	infection
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Some digraph problems

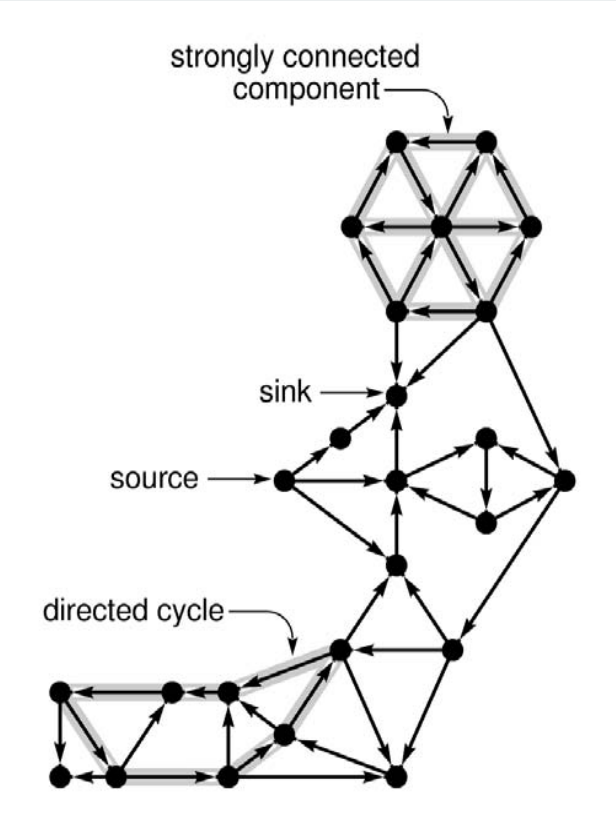
Transitive closure.
Is there a directed path from v to w?

Strong connectivity.
Are all vertices mutually reachable?

Topological sort.
Can you draw the digraph so that all edges point from left to right?

PERT/CPM.
Given a set of tasks with precedence constraints, how we can we best complete them all?

Shortest path. Find best route from s to t in a weighted digraph



PageRank. What is the importance of a web page?

Digraph representations

Vertices

- this lecture: use integers between `0` and `V-1`.
- real world: convert between names and integers with symbol table.

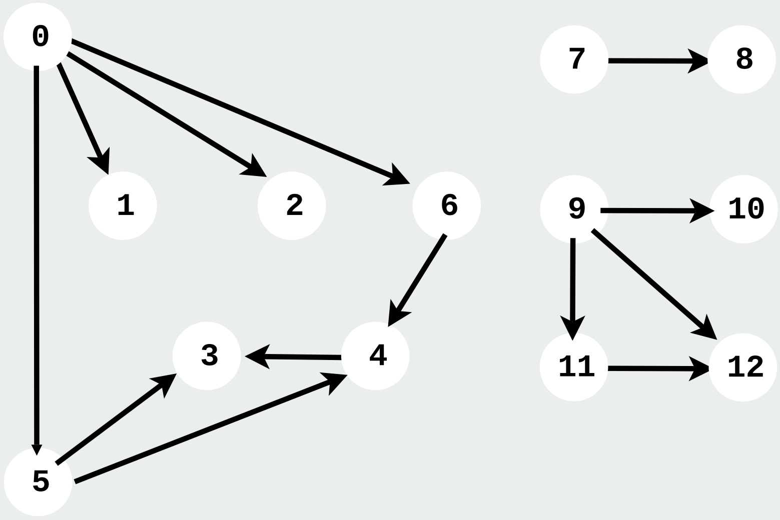
Edges: four easy options

- list of vertex pairs
- vertex-indexed adjacency arrays (adjacency matrix)
- vertex-indexed adjacency lists
- vertex-indexed adjacency SETs

Same as undirected graph

BUT

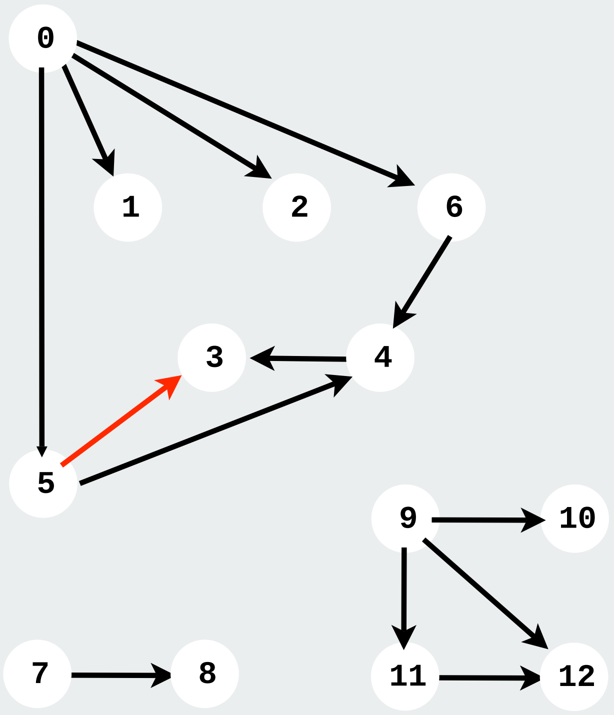
orientation of edges is significant.



Adjacency matrix digraph representation

Maintain a two-dimensional V × V boolean array.

For each edge v→w in graph: `adj[v][w] = true`.



one entry for each edge

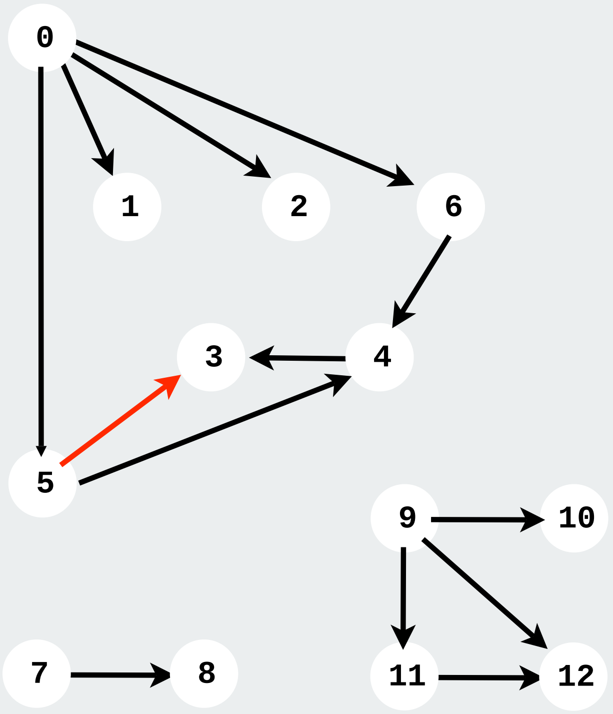
to

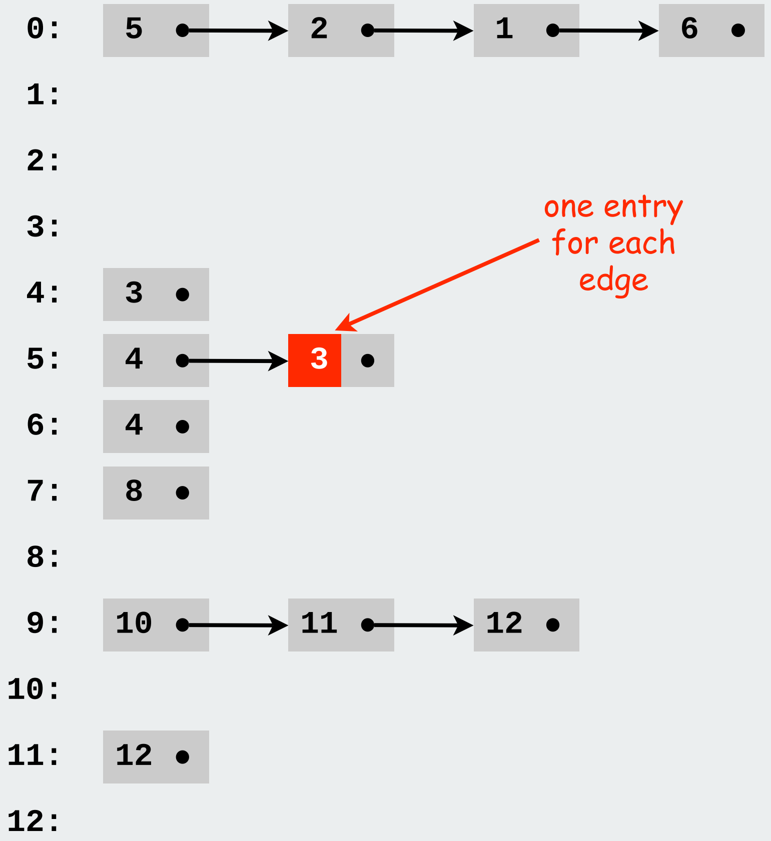
from

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	0	0	0
5	0	0	0	1	1	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1
12	0	0	0	0	0	0	0	0	0	0	0	0	0

Adjacency-list digraph representation

Maintain vertex-indexed array of lists.





0: 5 → 2 → 1 → 6

1:

2:

3:

4: 3

5: 4 → 3

6: 4

7: 8

8:

9: 10 → 11 → 12

10:

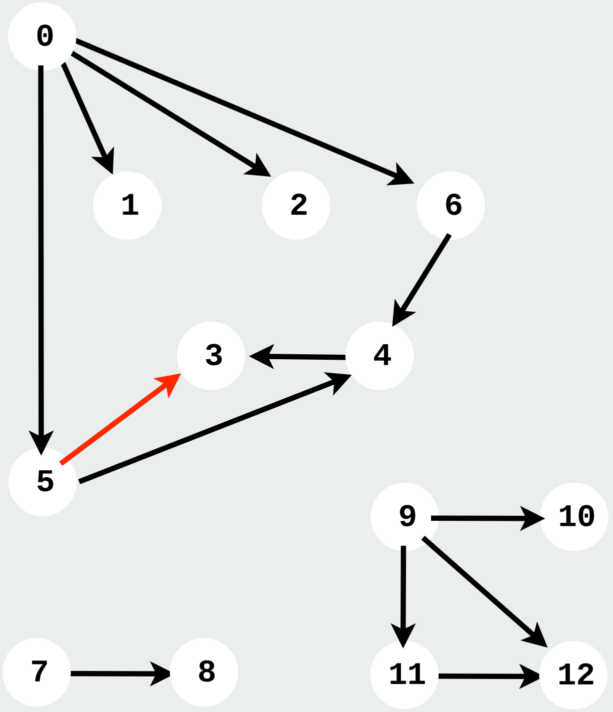
11: 12

12:

one entry for each edge

Adjacency-SET digraph representation

Maintain vertex-indexed array of SETs.



0:	{ 1 2 5 6 }
1:	{ }
2:	{ }
3:	{ }
4:	{ 3 }
5:	{ 3 4 }
6:	{ 4 }
7:	{ 8 }
8:	{ }
9:	{ 10 11 12 }
10:	{ }
11:	{ 12 }
12:	{ }

one entry for each edge

Adjacency-SET digraph representation: Java implementation

Same as **Graph**, but only insert one copy of each edge.

```
public class Digraph
{
   private int V;
   private SET<Integer>[] adj;

   public Digraph(int V)
   {
      this.V = V;
      adj = (SET<Integer>[]) new SET[V];
      for (int v = 0; v < V; v++)
         adj[v] = new SET<Integer>();
   }

   public void addEdge(int v, int w)
   {
      adj[v].add(w);
   }

   public Iterable<Integer> adj(int v)
   {
      return adj[v];
   }
}
```

adjacency SETs

create empty V-vertex graph

add edge from v to w (**Graph** also has **adj[w].add[v]**)

iterable SET for v's neighbors

Digraph representations

Digraphs are abstract mathematical objects, BUT

- ADT implementation requires specific representation.
- Efficiency depends on matching algorithms to representations.

representation	space	edge between v and w?	iterate over edges incident to v?
list of edges	E	E	E
adjacency matrix	V^2	1	V
adjacency list	E + V	degree(v)	degree(v)
adjacency SET	E + V	log (degree(v))	degree(v)

In practice: Use adjacency SET representation

- Take advantage of proven technology
- Real-world digraphs tend to be “sparse”

  [huge number of vertices, small average vertex degree]
- Algs all based on iterating over edges incident to v.

Typical digraph application: Google's PageRank algorithm

Goal. Determine which web pages on Internet are important.

Solution. Ignore keywords and content, focus on hyperlink structure.

Random surfer model.

- Start at random page.
- With probability 0.85, randomly select a hyperlink to visit next; with probability 0.15, randomly select any page.
- PageRank = proportion of time random surfer spends on each page.

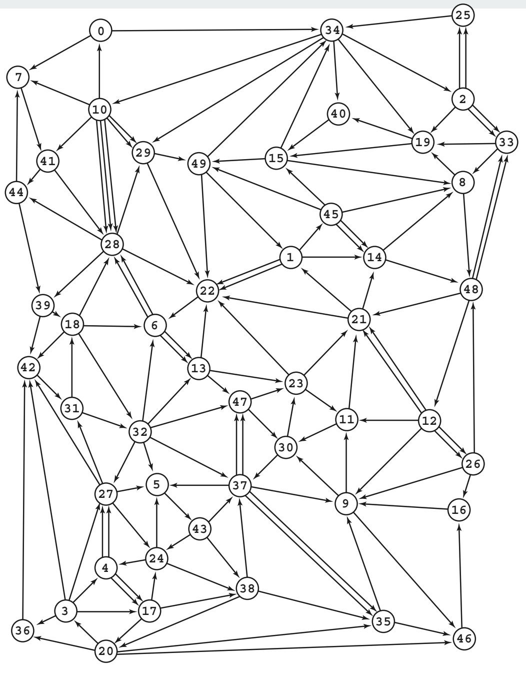
Solution 1: Simulate random surfer for a long time.

Solution 2: Compute ranks directly until they converge

Solution 3: Compute eigenvalues of adjacency matrix!

None feasible without sparse digraph representation

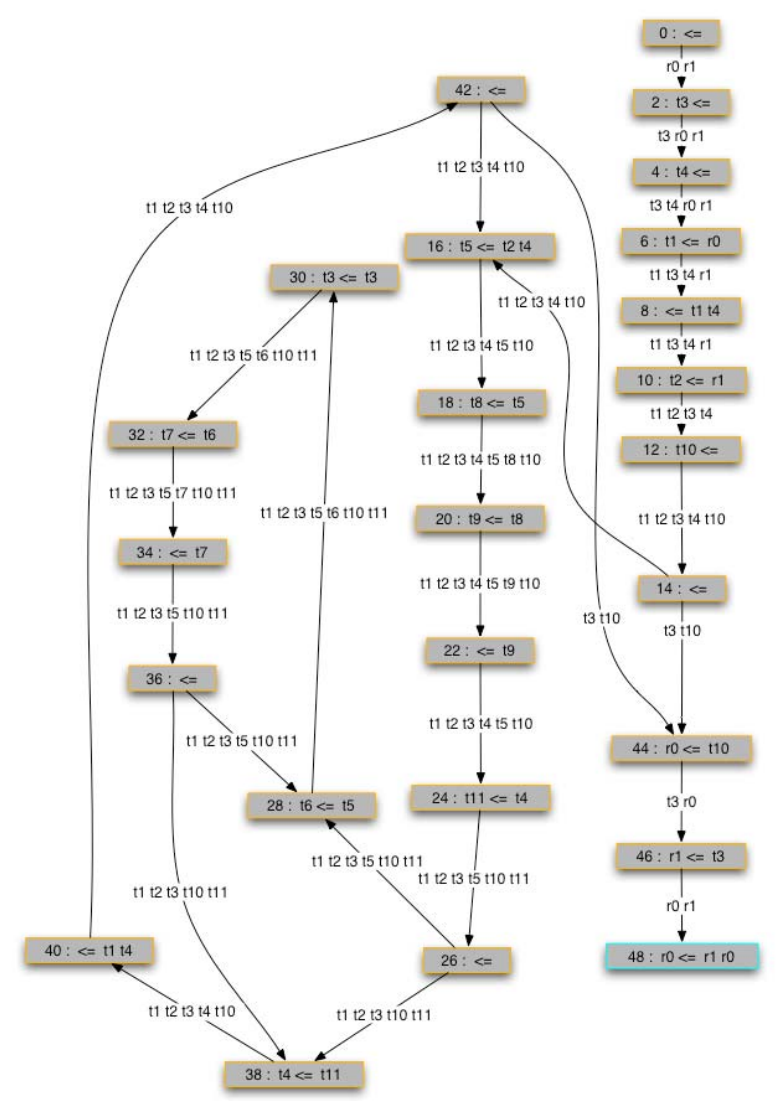
Every square matrix is a weighted digraph



- digraph search
- transitive closure
- topological sort
- strong components

Digraph application: program control-flow analysis

Every program is a digraph (instructions connected to possible successors)



Dead code elimination.

Find (and remove) unreachable code

can arise from compiler optimization (or bad code)

Infinite loop detection.

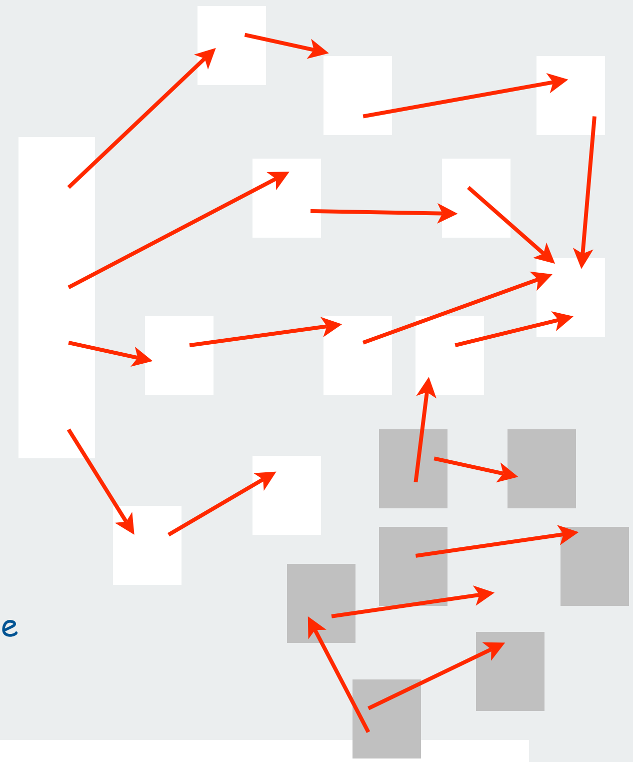
Determine whether exit is unreachable

can't detect all possible infinite loops (halting problem)

Digraph application: mark-sweep garbage collector

Every data structure is a digraph (objects connected by references)

Roots. Objects known to be directly accessible by program (e.g., stack).



Reachable objects.

Objects indirectly accessible by program (starting at a root and following a chain of pointers).

easy to identify pointers in type-safe language

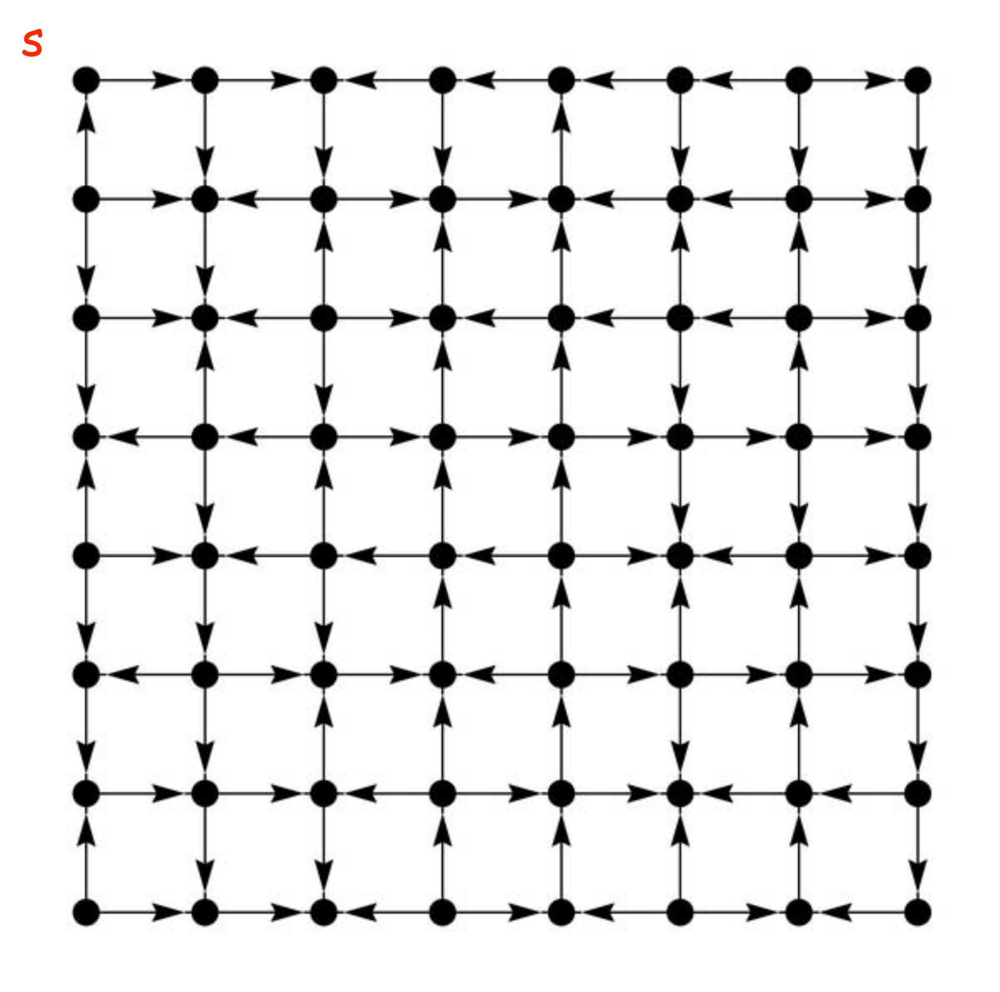
Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage, so add to free list.

Memory cost: Uses 1 extra mark bit per object, plus DFS stack.

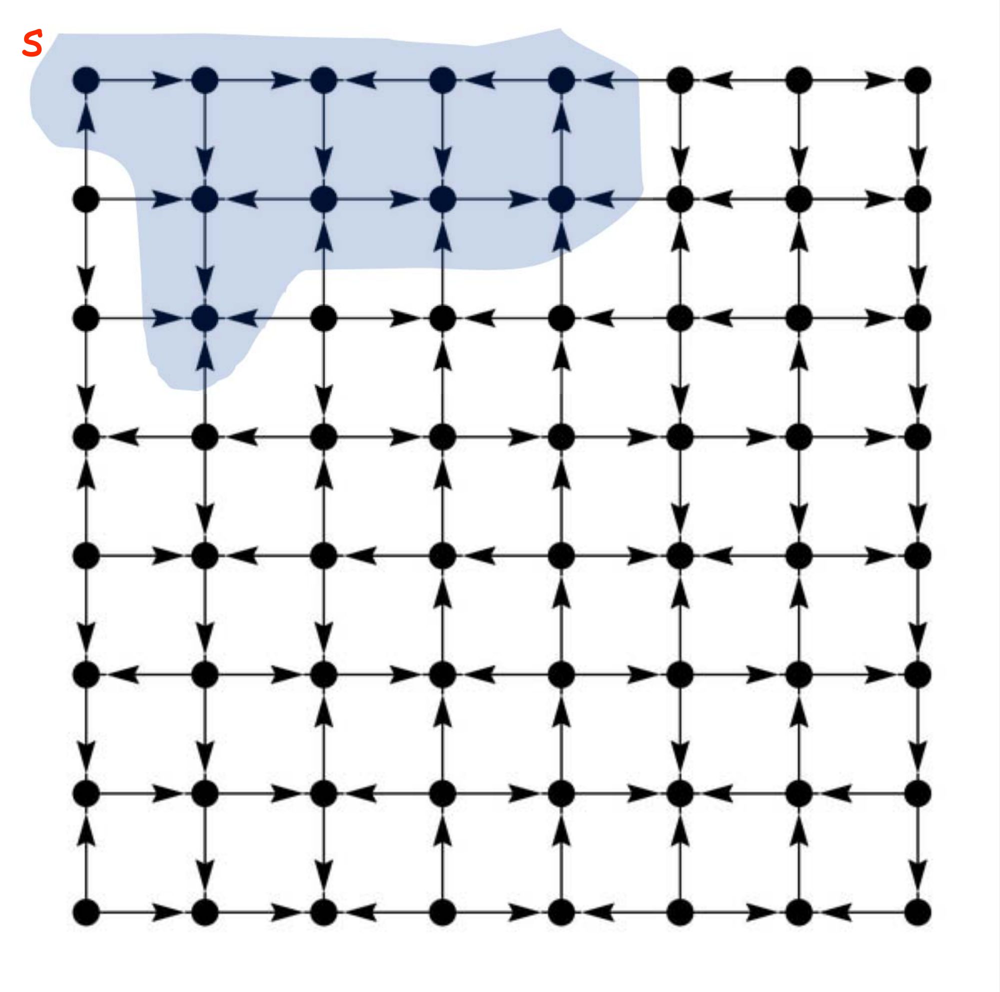
Reachability

Goal. Find all vertices reachable from s along a directed path.



Reachability

Goal. Find all vertices reachable from s along a directed path.



Digraph-processing challenge 1:

Problem: Mark all vertices reachable from a given vertex.

How difficult?

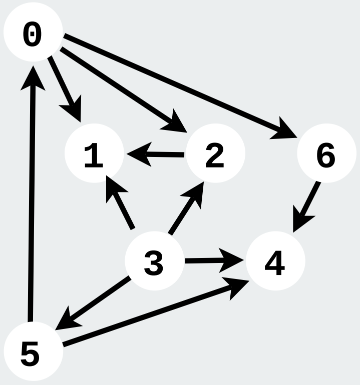
1) any COS126 student could do it

2) need to be a typical diligent COS226 student

3) hire an expert

4) intractable

5) no one knows



```
0-1
0-6
0-2
3-4
3-2
5-4
5-0
3-5
2-1
6-4
3-1
```

Depth-first search in digraphs

Same method as for undirected graphs

Every undirected graph is a digraph

- happens to have edges in both directions
- DFS is a digraph algorithm

```
DFS (to visit a vertex v)

Mark v as visited.
Visit all unmarked vertices w adjacent to v.
```

recursive

Depth-first search (single-source reachability)

Identical to undirected version (substitute `Digraph` for `Graph`).

```
public class DFSearcher
{
   private boolean[] marked;

   public DFSearcher(Digraph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

   private void dfs(Digraph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
   }

   public boolean isReachable(int v)
   {
      return marked[v];
   }
}
```

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex is connected to s

Depth-first search (DFS)

DFS enables direct solution of simple digraph problems.

- ✓ Reachability.
- Cycle detection
- Topological sort
- Transitive closure.
- Is there a path from s to t ?

stay tuned

Basis for solving difficult digraph problems.

- Directed Euler path.
- Strong connected components.

Breadth-first search in digraphs

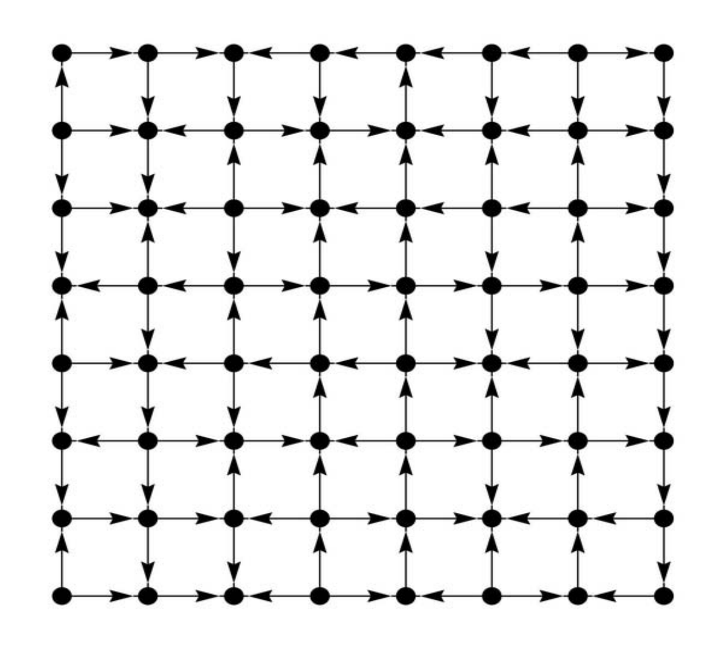
Same method as for undirected graphs

Every undirected graph is a digraph

- happens to have edges in both directions
- BFS is a digraph algorithm

BFS (from source vertex s)

```
Put s onto a FIFO queue.
Repeat until the queue is empty:
    ▪ remove the least recently added vertex v
    ▪ add each of v's unvisited neighbors to the
      queue and mark them as visited.
```

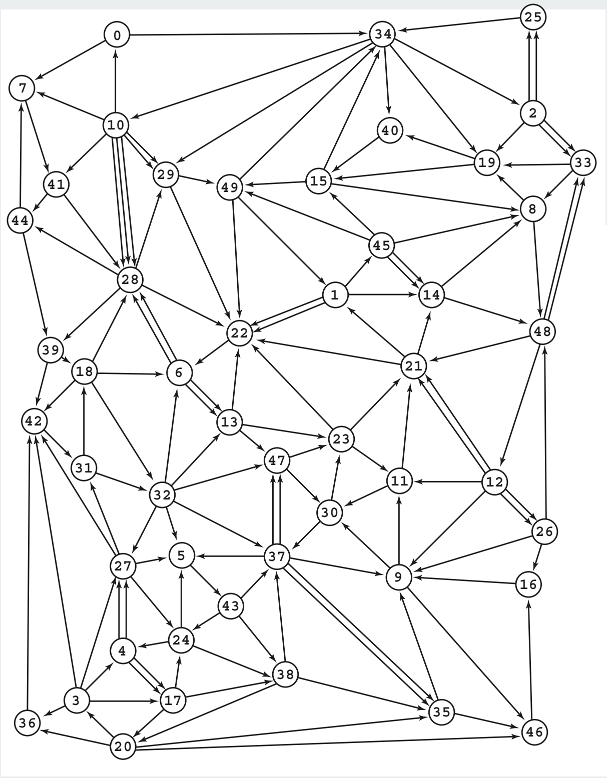


Visits vertices in increasing distance from s

Digraph BFS application: Web Crawler

The internet is a digraph

Goal. Crawl Internet, starting from some root website.
Solution. BFS with implicit graph.



BFS.

- Start at some root website
  (say **`http://www.princeton.edu`**.).
- Maintain a **`Queue`** of websites to explore.
- Maintain a **`SET`** of discovered websites.
- Dequeue the next website
  and enqueue websites to which it links
  (provided you haven't done so before).

Q. Why not use DFS?

A. Internet is not fixed (some pages generate new ones when visited)

subtle point: think about it!

Web crawler: BFS-based Java implementation

```
Queue<String> q = new Queue<String>();
SET<String> visited = new SET<String>();

String s = "http://www.princeton.edu";
q.enqueue(s);
visited.add(s);

while (!q.isEmpty())
{
   String v = q.dequeue();
   System.out.println(v);
   In in = new In(v);
   String input = in.readAll();

   String regexp = "http://(\\w+\\.)*(\\w+)";
   Pattern pattern = Pattern.compile(regexp);
   Matcher matcher = pattern.matcher(input);
   while (matcher.find())
   {
      String w = matcher.group();
      if (!visited.contains(w))
      {
            visited.add(w);
            q.enqueue(w);
      }
   }
}
```

queue of sites to crawl

set of visited sites

start crawling from s

read in raw html for next site in queue

http://xxx.yyy.zzz

use regular expression
to find all URLs in site

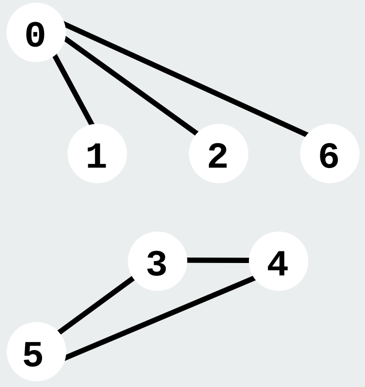
if unvisited, mark as visited
and put on queue

- digraph search
- transitive closure
- topological sort
- strong components

Graph-processing challenge (revisited)

Problem: Is there a path from s to t ?

Goals: linear ~(V + E) preprocessing time
constant query time



0-1
0-6
0-2
4-3
5-3
5-4

How difficult?

1) any COS126 student could do it

2) need to be a typical diligent COS226 student

3) hire an expert

4) intractable

5) no one knows

6) impossible

Digraph-processing challenge 2

Problem: Is there a directed path from s to t ?

Goals: linear ~(V + E) preprocessing time

constant query time

How difficult?

1) any COS126 student could do it

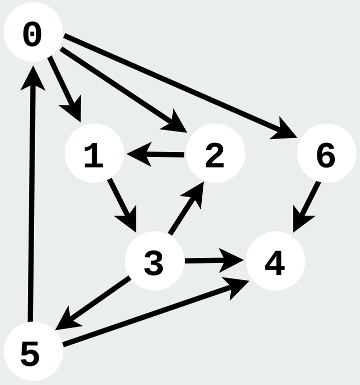
2) need to be a typical diligent COS226 student

3) hire an expert

4) intractable

5) no one knows

6) impossible



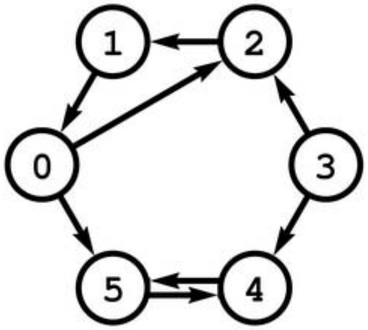
```
0-1
0-6
0-2
3-4
3-2
5-4
5-0
3-5
2-1
6-4
1-3
```

Transitive Closure

The transitive closure of G has an directed edge from v to w
if there is a directed path from v to w in G

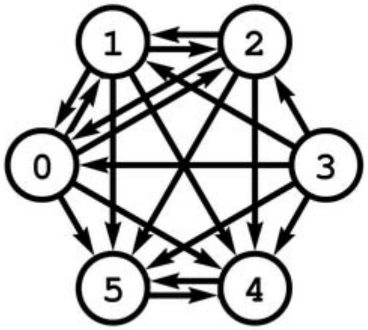
graph is usually sparse

G



	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	0	0	0	0
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1

Transitive closure of G



	0	1	2	3	4	5
0	1	1	1	0	1	1
1	1	1	1	0	1	1
2	1	1	1	0	1	1
3	1	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1

TC is usually dense
so adjacency matrix
representation is OK

Digraph-processing challenge 2 (revised)

Problem: Is there a directed path from s to t ?

Goals: $\sim V^2$ preprocessing time

constant query time

How difficult?

1) any COS126 student could do it

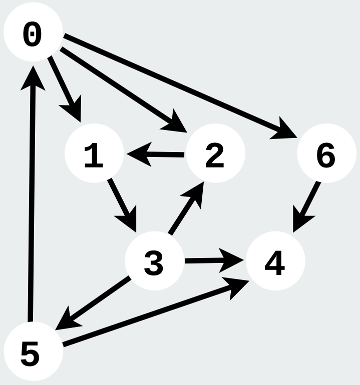
2) need to be a typical diligent COS226 student

3) hire an expert

4) intractable

5) no one knows

6) impossible



```
0-1
0-6
0-2
3-4
3-2
5-4
5-0
3-5
2-1
6-4
1-3
```

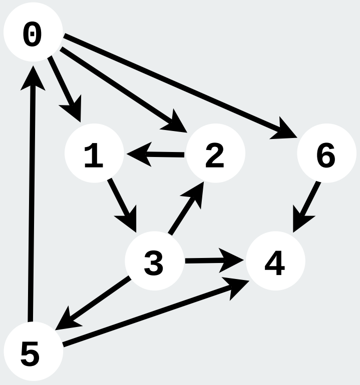
Digraph-processing challenge 2 (revised again)

Problem: Is there a directed path from s to t ?

Goals: ~VE preprocessing time (~V^3 for dense digraphs)
~V^2 space
constant query time

How difficult?

1) any COS126 student could do it
2) need to be a typical diligent COS226 student
3) hire an expert
4) intractable
5) no one knows
6) impossible



```
0-1
0-6
0-2
3-4
3-2
5-4
5-0
3-5
2-1
6-4
1-3
```

Transitive closure: Java implementation

Use an array of **DFSearcher** objects,
one for each row of transitive closure

```
public class TransitiveClosure
{
   private DFSearcher[] tc;

   public TransitiveClosure(Digraph G)
   {
      tc = new DFSearcher[G.V()];
      for (int v = 0; v < G.V(); v++)
         tc[v] = new DFSearcher(G, v);
   }

   public boolean reachable(int v, int w)
   {
      return tc[v].isReachable(w);
   }
}
```

is there a directed path from v to w ?

```
public class DFSearcher
{
   private boolean[] marked;
   public DFSearcher(Digraph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }
   private void dfs(Digraph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
   }
   public boolean isReachable(int v)
   {
      return marked[v];
   }
}
```

- digraph search
- transitive closure
- **topological sort**
- strong components

Digraph application: Scheduling

Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

Graph model.

- Create a vertex v for each task.
- Create an edge v→w if task v must precede task w.
- Schedule tasks in topological order.

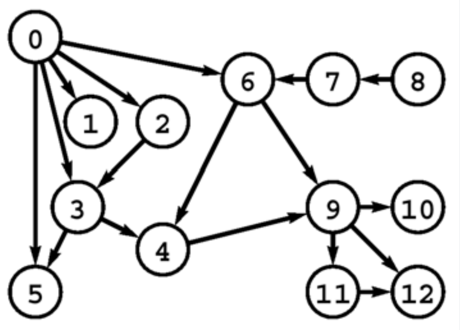
tasks

0. read programming assignment
1. download files
2. write code
3. attend precept

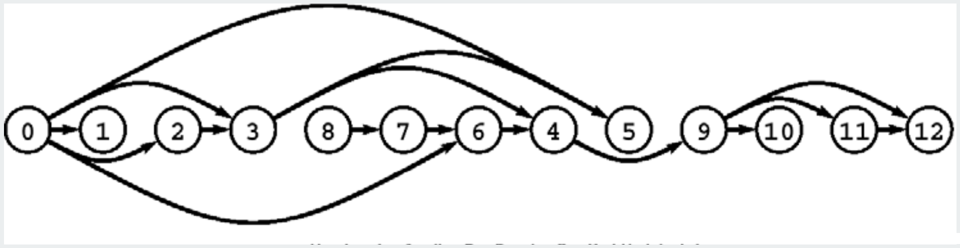
...

12. sleep

precedence constraints

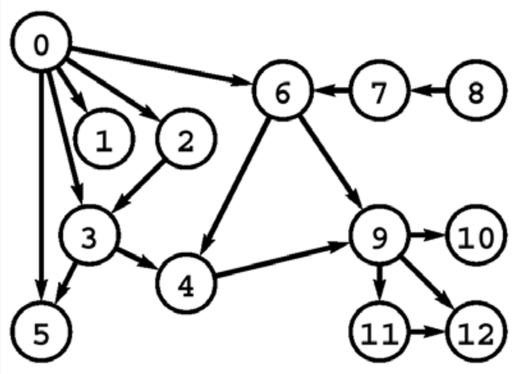


feasible schedule

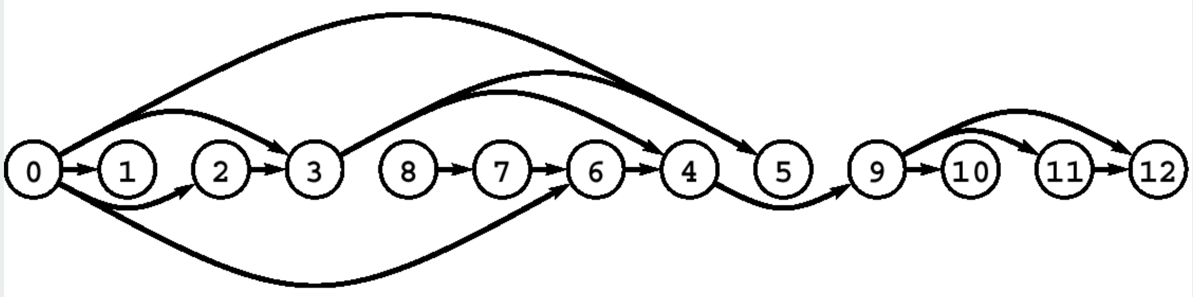


Topological Sort

DAG. Directed acyclic graph.



Topological sort. Redraw DAG so all edges point left to right.



Observation. Not possible if graph has a directed cycle.

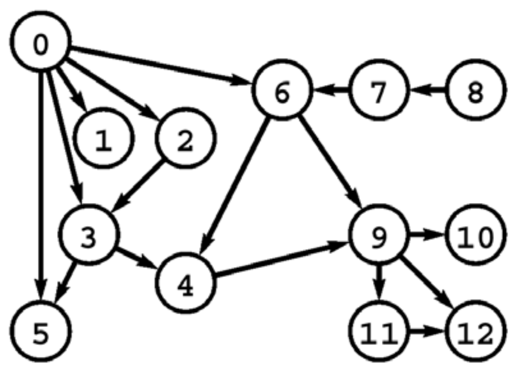
Digraph-processing challenge 3

Problem: Check that the digraph is a DAG.
If it is a DAG, do a topological sort.

Goals: linear ~(V + E) preprocessing time
provide client with vertex iterator for topological order

How difficult?

1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows
6) impossible



```
0-1
0-6
0-2
0-5
2-3
4-9
6-4
6-9
7-6
8-7
9-10
9-11
9-12
11-12
```

Topological sort in a DAG: Java implementation

```
public class TopologicalSorter
{
   private int count;
   private boolean[] marked;
   private int[] ts;

   public TopologicalSorter(Digraph G)
   {
      marked = new boolean[G.V()];
      ts = new int[G.V()];
      count = G.V();
      for (int v = 0; v < G.V(); v++)
         if (!marked[v]) tsort(G, v);
   }

   private void tsort(Digraph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) tsort(G, w);
      ts[--count] = v;
   }
}
```

standard DFS
with 5
extra lines of code

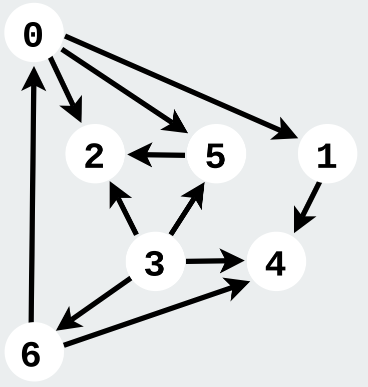
add iterator that returns
`ts[0], ts[1], ts[2]...`

Seems easy? Missed by experts for a few decades

Topological sort of a dag: trace

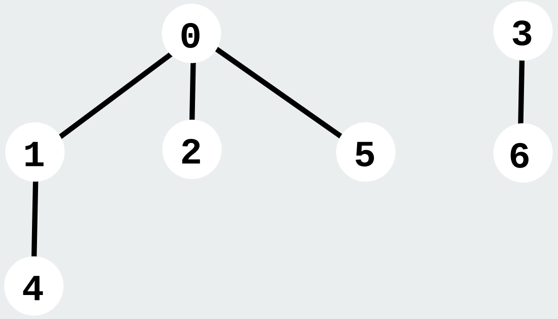
"visit" means "call `tsort()`" and "leave" means "return from `tsort()`

```
                        marked[]          ts[]
visit 0:             1 0 0 0 0 0 0    0 0 0 0 0 0 0
  visit 1:           1 1 0 0 0 0 0    0 0 0 0 0 0 0
    visit 4:         1 1 0 0 1 0 0    0 0 0 0 0 0 0
    leave 4:         1 1 0 0 1 0 0    0 0 0 0 0 0 4
  leave 1:           1 1 0 0 1 0 0    0 0 0 0 0 1 4
  visit 2:           1 1 1 0 1 0 0    0 0 0 0 0 1 4
  leave 2:           1 1 1 0 1 0 0    0 0 0 0 2 1 4
  visit 5:           1 1 1 0 1 1 0    0 0 0 0 2 1 4
    check 2:         1 1 1 0 1 1 0    0 0 0 0 2 1 4
  leave 5:           1 1 1 0 1 1 0    0 0 0 5 2 1 4
leave 0:             1 1 1 0 1 1 0    0 0 0 5 2 1 4
check 1:             1 1 1 0 1 1 0    0 0 0 5 2 1 4
check 2:             1 1 1 0 1 1 0    0 0 0 5 2 1 4
visit 3:             1 1 1 1 1 1 0    0 0 0 5 2 1 4
  check 2:           1 1 1 1 1 1 0    0 0 0 5 2 1 4
  check 4:           1 1 1 1 1 1 0    0 0 0 5 2 1 4
  check 5:           1 1 1 1 1 1 0    0 0 0 5 2 1 4
  visit 6:           1 1 1 1 1 1 1    0 0 0 5 2 1 4
  leave 6:           1 1 1 1 1 1 1    0 6 0 5 2 1 4
leave 3:             1 1 1 1 1 1 1    3 6 0 5 2 1 4
check 4:             1 1 1 1 1 1 0    3 6 0 5 2 1 4
check 5:             1 1 1 1 1 1 0    3 6 0 5 2 1 4
check 6:             1 1 1 1 1 1 0    3 6 0 5 2 1 4
```



adj SETs

```
0: 1  2  5
1: 4
2:
3: 2  4  5  6
4:
5: 2
6: 0  4
```



3 6 0 5 2 1 4

Topological sort in a DAG: correctness proof

Invariant:

`tsort(G, v)` visits all vertices reachable from v with a directed path

Proof by induction:

- w marked: vertices reachable from w are already visited
- w not marked: call `tsort(G, w)` to visit the vertices reachable from w

```
public class TopologicalSorter
{
   private int count;
   private boolean[] marked;
   private int[] ts;

   public TopologicalSorter(Digraph G)
   {
      marked = new boolean[G.V()];
      ts = new int[G.V()];
      count = G.V();
      for (int v = 0; v < G.V(); v++)
         if (!marked[v]) tsort(G, v);
   }

   private void tsort(Digraph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) tsort(G, w);
      ts[--count] = v;
   }
}
```

Therefore, algorithm is correct in placing v before all vertices visited during call to `tsort(G, v)` just before returning.

Q. How to tell whether the digraph has a cycle (is not a DAG)?

A. Use `TopologicalSorter` (exercise)

Topological sort applications.

- Causalities.
- Compilation units.
- Class inheritance.
- Course prerequisites.
- Deadlocking detection.
- Temporal dependencies.
- Pipeline of computing jobs.
- Check for symbolic link loop.
- Evaluate formula in spreadsheet.
- Program Evaluation and Review Technique / Critical Path Method

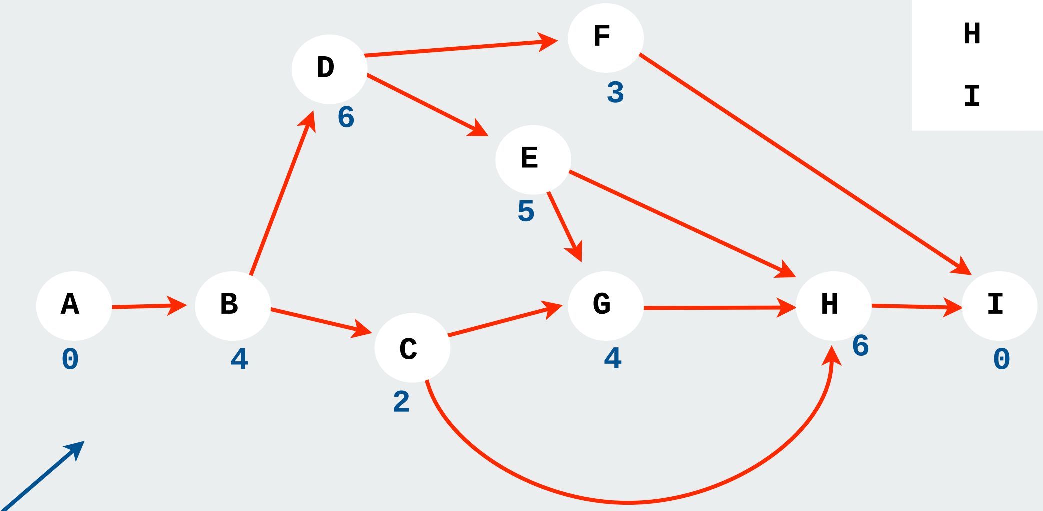
Topological sort application (weighted DAG)

Precedence scheduling

- Task v takes `time[v]` units of time.
- Can work on jobs in parallel.
- Precedence constraints:
- must finish task v before beginning task w.
- Goal: finish each task as soon as possible

index	task	time	prereq
A	begin	0	-
B	framing	4	A
C	roofing	2	B
D	siding	6	B
E	windows	5	D
F	plumbing	3	D
G	electricity	4	C, E
H	paint	6	C, E
I	finish	0	F, H

Example:



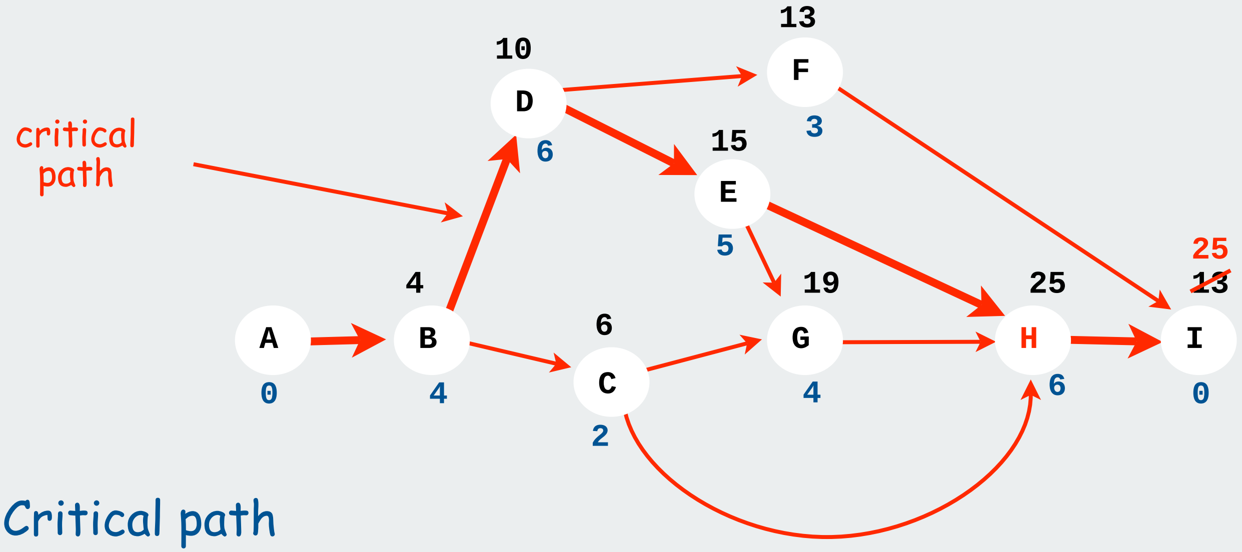
vertices labelled
A-I in topological order

Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.

- compute topological order of vertices.
- initialize `fin[v] = 0` for all vertices `v`.
- consider vertices `v` in topologically sorted order.

  for each edge `v→w`, set `fin[w]= max(fin[w], fin[v] + time[w])`



Critical path

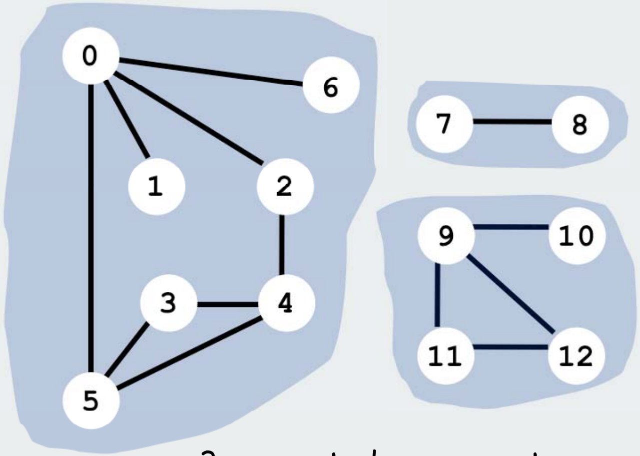
- remember vertex that set value.
- work backwards from sink

- digraph search
- transitive closure
- topological sort
- **strong components**

Strong connectivity in digraphs

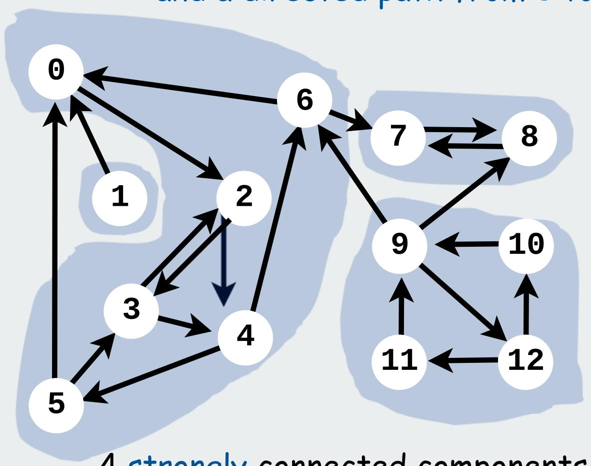
Analog to connectivity in undirected graphs

In a **Graph**, **u** and **v** are connected
when there is a path from **u** to **v**



3 connected components
(sets of mutually connected vertices)

In a **Digraph**, **u** and **v** are strongly connected
when there is a directed path from **u** to **v**
and a directed path from **v** to **u**



4 strongly connected components
(sets of mutually strongly connected vertices)

Connectivity table (easy to compute with DFS)

	0	1	2	3	4	5	6	7	8	9	10	11	12
cc	0	0	0	0	0	0	0	1	1	2	2	2	2

```
public int connected(int v, int w)
{  return cc[v] == cc[w];  }
```

constant-time client connectivity query

Strong connectivity table (how to compute?)

	0	1	2	3	4	5	6	7	8	9	10	11	12
sc	2	1	2	2	2	2	2	3	3	0	0	0	0

```
public int connected(int v, int w)
{  return cc[v] == cc[w];  }
```

constant-time client strong connectivity query

Digraph-processing challenge 4

Problem: Is there a directed cycle containing s and t ?

Equivalent: Are there directed paths from s to t and from t to s?

Equivalent: Are s and t strongly connected?

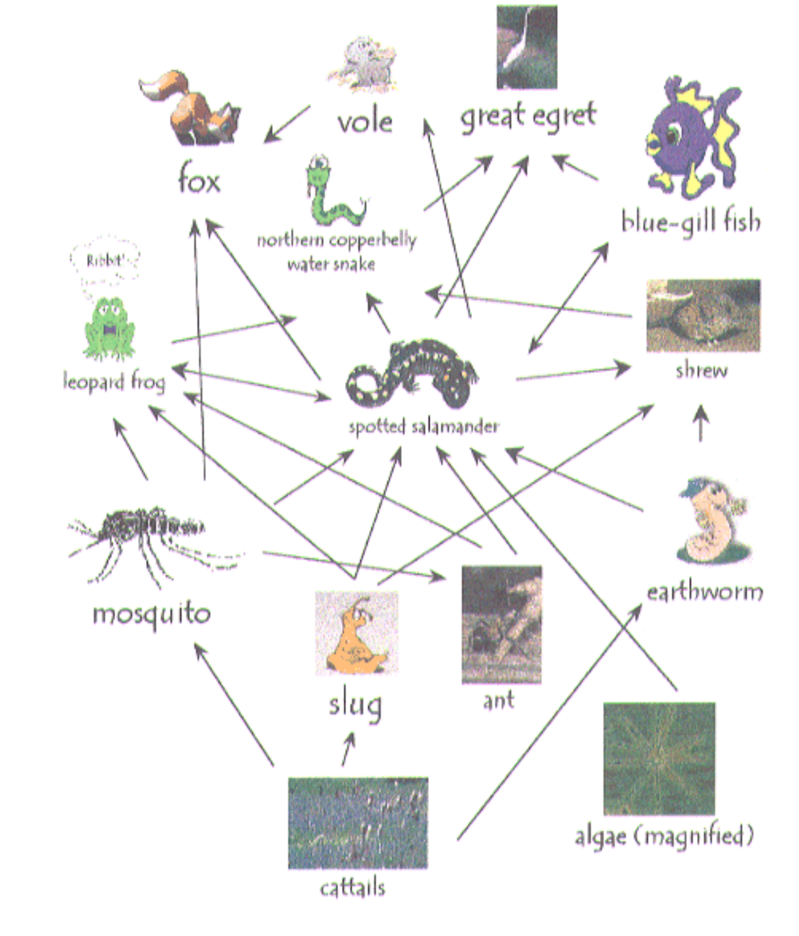
Goals: linear (V + E) preprocessing time (like for undirected graphs)
constant query time

How difficult?

1) any COS126 student could do it
2) need to be a typical diligent COS226 student
3) hire an expert
4) intractable
5) no one knows
6) impossible

Typical strong components applications

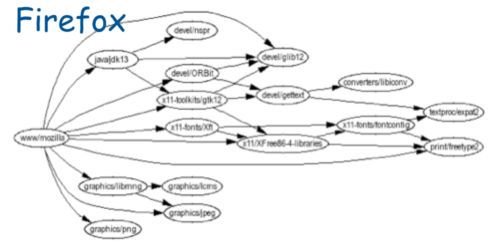
Ecological food web

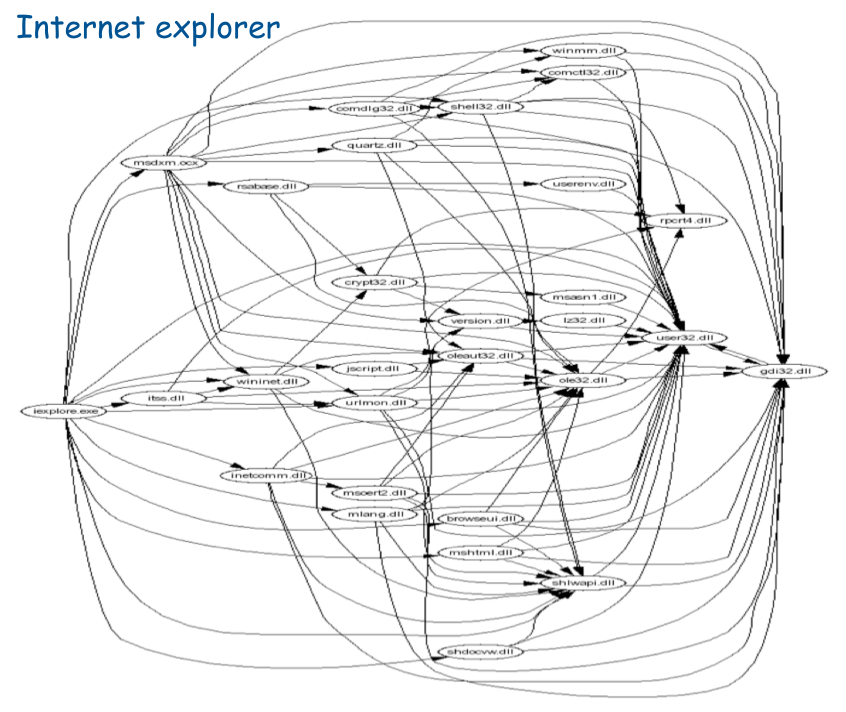


Strong component: subset with common energy flow

- source in kernel DAG: needs outside energy?
- sink in kernel DAG: heading for growth?

Software module dependency digraphs





Strong component: subset of mutually interacting modules

- approach 1: package strong components together
- approach 2: use to improve design!

Strong components algorithms: brief history

1960s: Core OR problem

- widely studied
- some practical algorithms
- complexity not understood

1972: Linear-time DFS algorithm (Tarjan)

- classic algorithm
- level of difficulty: CS226++
- demonstrated broad applicability and importance of DFS

1980s: Easy two-pass linear-time algorithm (Kosaraju)

- forgot notes for teaching algorithms class
- developed algorithm in order to teach it!
- later found in Russian scientific literature (1972)

1990s: More easy linear-time algorithms (Gabow, Mehlhorn)

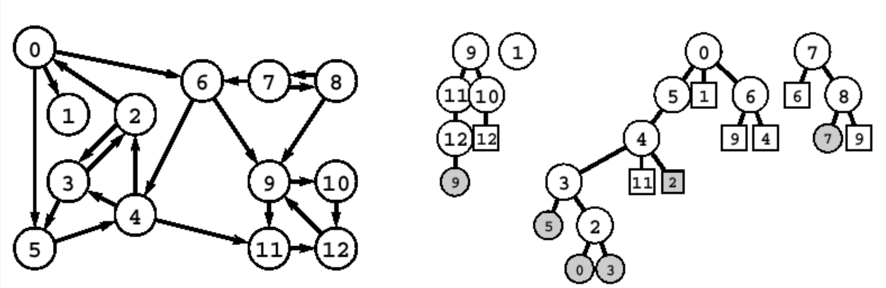
- Gabow: fixed old OR algorithm
- Mehlhorn: needed one-pass algorithm for LEDA

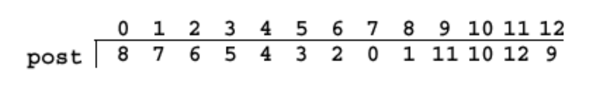
Kosaraju's algorithm

Simple (but mysterious) algorithm for computing strong components

- Run DFS on G^R and compute postorder.
- Run DFS on G, considering vertices in reverse postorder
- [has to be seen to be believed: follow example in book]

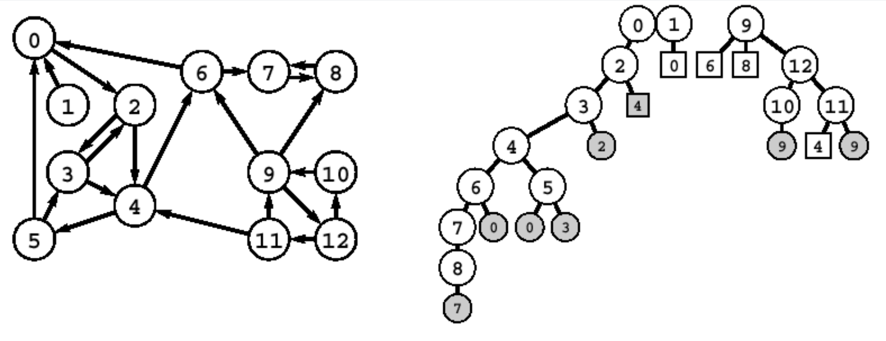
G

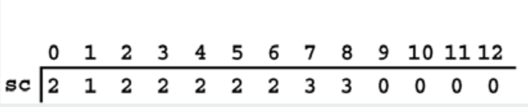




	0	1	2	3	4	5	6	7	8	9	10	11	12
post	8	7	6	5	4	3	2	0	1	11	10	12	9

G^R





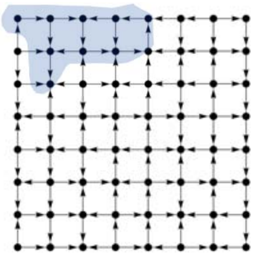
	0	1	2	3	4	5	6	7	8	9	10	11	12
sc	2	1	2	2	2	2	2	3	3	0	0	0	0

Theorem. Trees in second DFS are strong components. (!)

Proof. [stay tuned in COS 423]

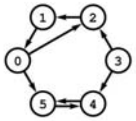
Digraph-processing summary: Algorithms of the day

Single-source reachability

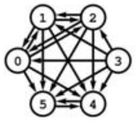


DFS

transitive closure



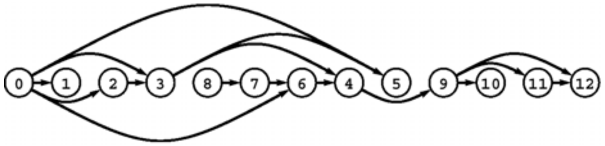
	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	0	0	0	0
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1



	0	1	2	3	4	5
0	1	1	1	0	1	1
1	1	1	1	0	1	1
2	1	1	1	0	1	1
3	1	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1

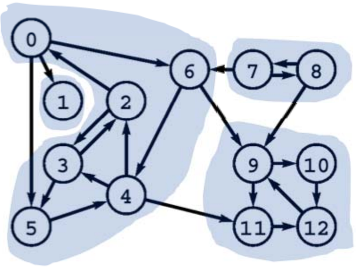
DFS from each vertex

topological sort
(DAG)



DFS

strong components



Kosaraju
DFS (twice)

Minimum Spanning Trees

- weighted graph API
- cycles and cuts
- Kruskal’s algorithm
- Prim’s algorithm
- advanced topics

References:

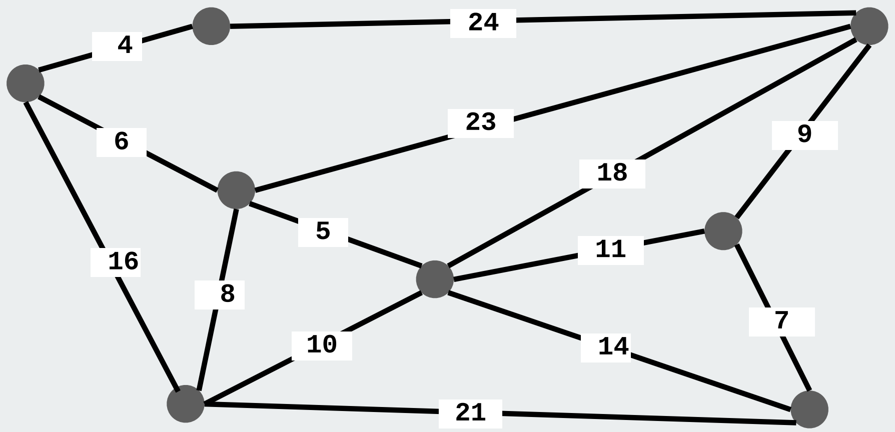
Algorithms in Java, Chapter 20

http://www.cs.princeton.edu/introalgsds/54mst

Minimum Spanning Tree

Given. Undirected graph G with positive edge weights (connected).

Goal. Find a min weight set of edges that connects all of the vertices.

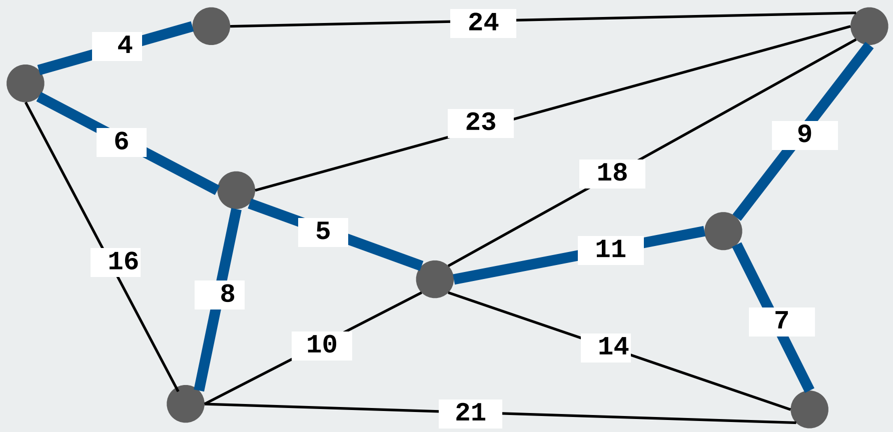


G

Minimum Spanning Tree

Given. Undirected graph G with positive edge weights (connected).

Goal. Find a min weight set of edges that connects all of the vertices.



weight(T) = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

Brute force: Try all possible spanning trees

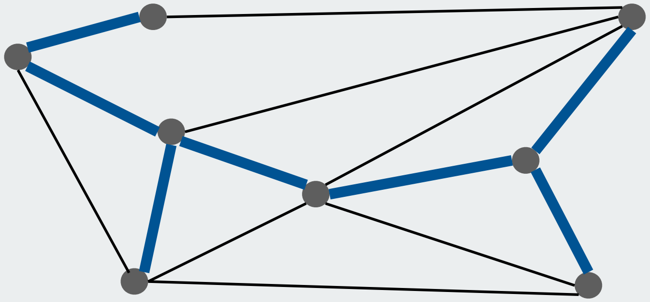
- problem 1: not so easy to implement
- problem 2: far too many of them ←

Ex: [Cayley, 1889]: V^{V-2} spanning trees on the complete graph on V vertices.

MST Origin

Otakar Boruvka (1926).

- Electrical Power Company of Western Moravia in Brno.
- Most economical construction of electrical power network.
- Concrete engineering problem is now a cornerstone problem-solving model in combinatorial optimization.



Otakar Boruvka

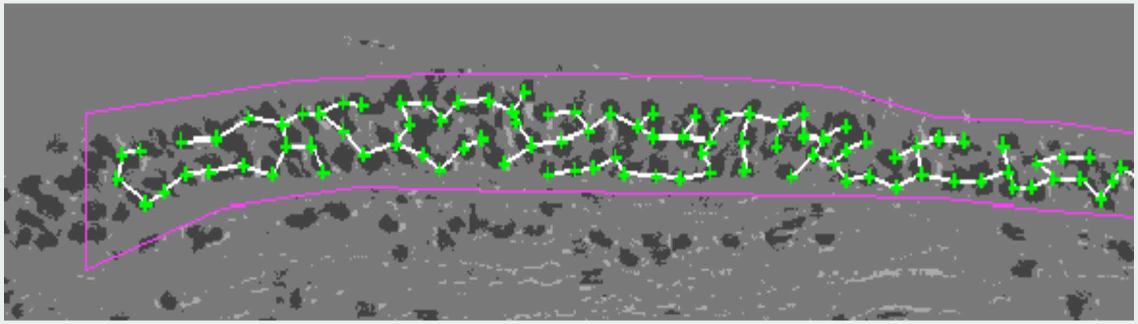
Applications

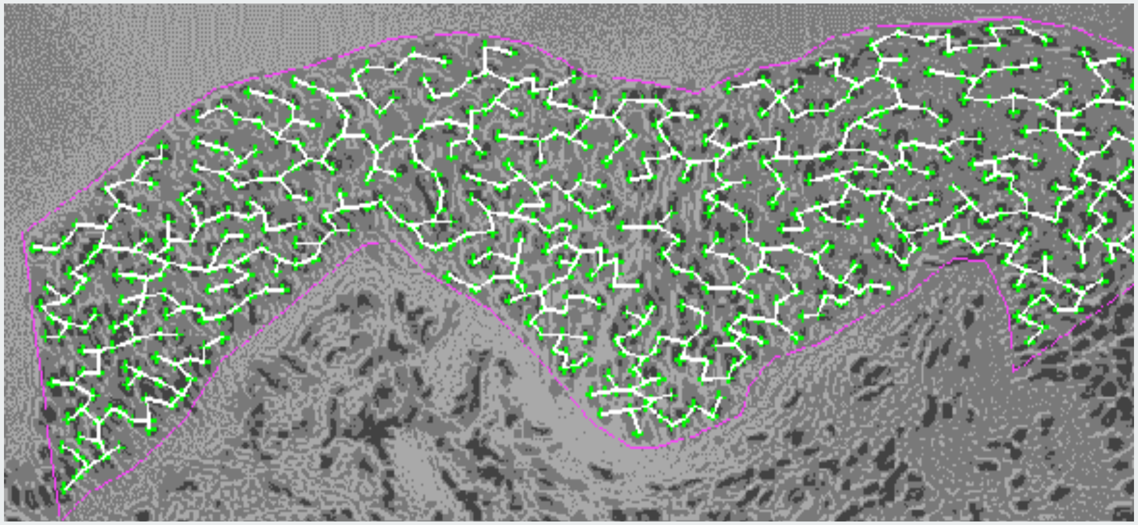
MST is fundamental problem with diverse applications.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.

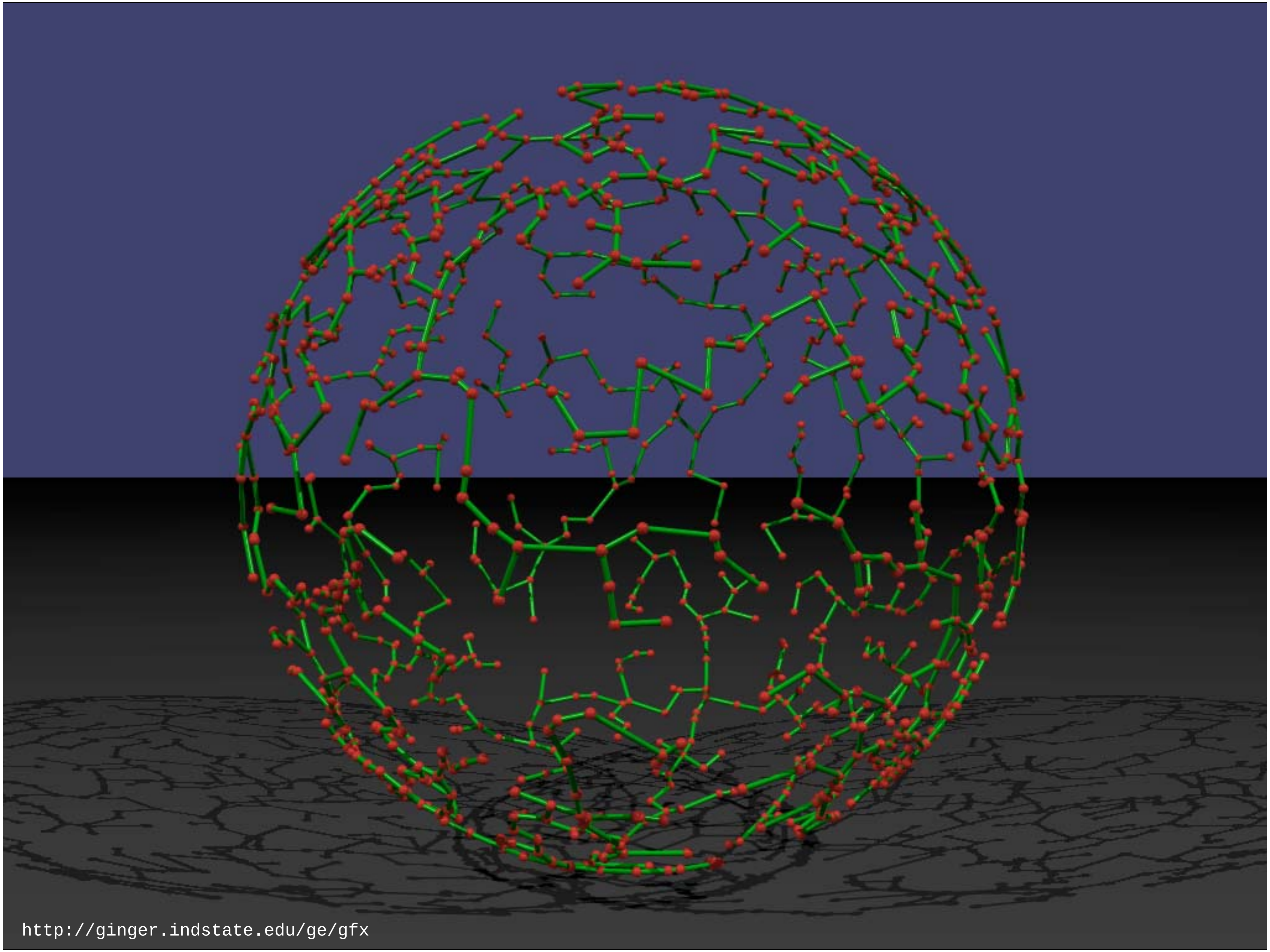
Medical Image Processing

MST describes arrangement of nuclei in the epithelium for cancer research





http://www.bccrc.ca/ci/ta01_archlevel.html



http://ginger.indstate.edu/ge/gfx

Two Greedy Algorithms

Kruskal's algorithm. Consider edges in ascending order of cost. Add the next edge to T unless doing so would create a cycle.

Prim's algorithm. Start with any vertex s and greedily grow a tree T from s. At each step, add the cheapest edge to T that has exactly one endpoint in T.

Proposition. Both greedy algorithms compute an MST.

Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit." - Gordon Gecko

- weighted graph API
- cycles and cuts
- Kruskal’s algorithm
- Prim’s algorithm
- advanced topics

Weighted Graph API

`public class WeightedGraph`

	`WeightedGraph(int V)`	create an empty graph with V vertices
`void`	`insert(Edge e)`	insert edge e
`Iterable<Edge>`	`adj(int v)`	return an iterator over edges incident to v
`int`	`V()`	return the number of vertices
`String`	`toString()`	return a string representation

iterate through all edges (once in each direction)

Weighted graph data type

Identical to **Graph.java** but use **Edge** adjacency sets instead of **int**.

```
public class WeightedGraph
{
   private int V;
   private SET<Edge>[] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = (SET<Edge>[]) new SET[V];
      for (int v = 0; v < V; v++)
         adj[v] = new SET<Edge>();
   }

   public void addEdge(Edge e)
   {
      int v = e.v, w = e.w;
      adj[v].add(e);
      adj[w].add(e);
   }

   public Iterable<Edge> adj(int v)
   {  return adj[v];  }

}
```

Weighted edge data type

```
public class Edge implements Comparable<Edge>
{
   private final int v, int w;
   private final double weight;

   public Edge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int either()
   {  return v; }

   public int other(int vertex)
   {
      if (vertex == v) return w;
      else return v;
   }

   public int weight()
   {  return weight; }


   // See next slide for edge compare methods.

}
```

Edge abstraction
needed for weights

slightly tricky accessor methods
(enables client code like this)

```
for (int v = 0; v < G.V(); v++)
{
   for (Edge e : G.adj(v))
   {
      int w = e.other(v);

      // edge v-w
   }
}
```

Weighted edge data type: compare methods

Two different compare methods for edges

- **`compareTo()`** so that edges are **`Comparable`** (for use in **`SET`**)
- **`compare()`** so that clients can compare edges by weight.

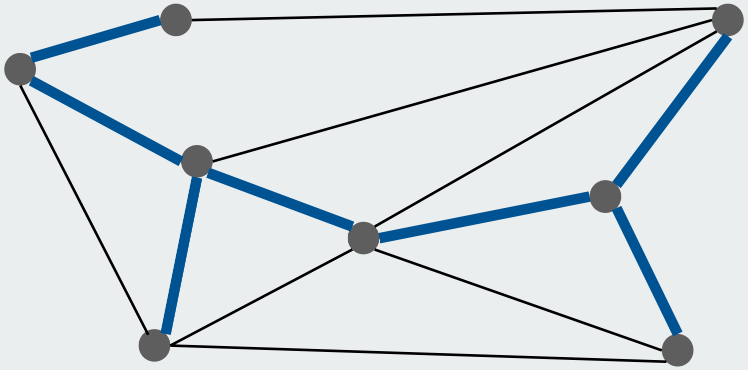
```
public final static Comparator<Edge> BY_WEIGHT = new ByWeightComparator();

private static class ByWeightComparator implements Comparator<Edge>
{
   public int compare(Edge e, Edge f)
   {
      if (e.weight < f.weight) return -1;
      if (e.weight > f.weight) return +1;
      return 0;
   }
}

   public int compareTo(Edge that)
   {
      if      (this.weight < that.weight) return -1;
      else if (this.weight > that.weight) return +1;
      else                                return  0;
   }
}
```

- weighted graph API
- **cycles and cuts**
- Kruskal’s algorithm
- Prim’s algorithm
- advanced topics

Spanning Tree

MST. Given connected graph G with positive edge weights, find a min weight set of edges that connects all of the vertices.

Def. A spanning tree of a graph G is a subgraph T that is connected and acyclic.

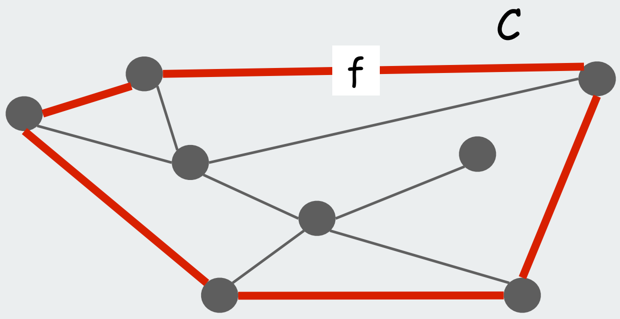


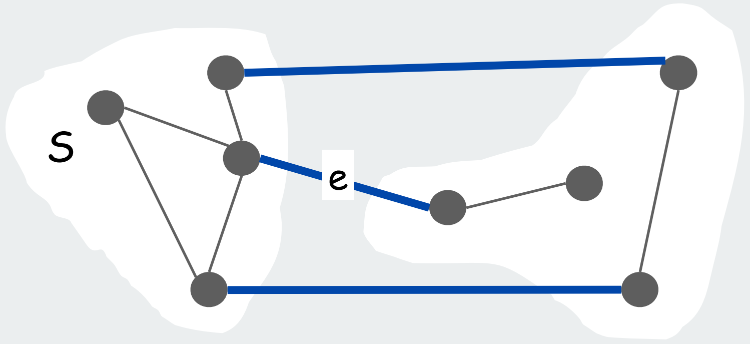
Property. MST of G is always a spanning tree.

Simplifying assumption. All edge weights w_e are distinct.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.

Cut property. Let S be any subset of vertices, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.





f is not in the MST

e is in the MST

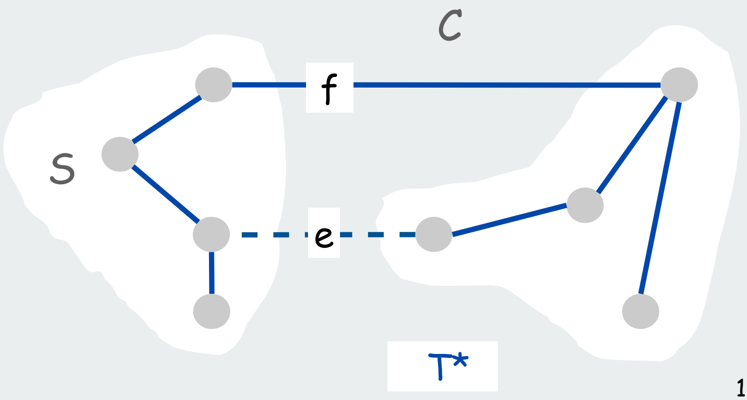
Cycle Property

Simplifying assumption. All edge weights w_e are distinct.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST T* does not contain f.

Pf. [by contradiction]

- Suppose f belongs to T*. Let's see what happens.
- Deleting f from T* disconnects T*. Let S be one side of the cut.
- Some other edge in C, say e, has exactly one endpoint in S.
- $T = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T) < \text{cost}(T^*)$.
- Contradicts minimality of T*. ▪



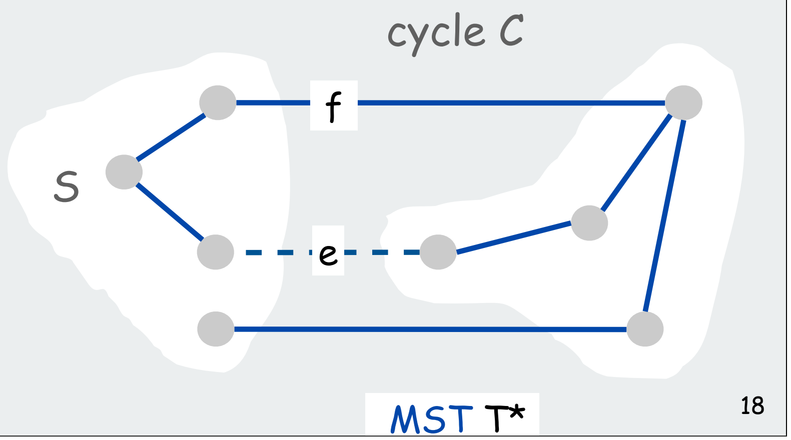
Cut Property

Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of vertices, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

Pf. [by contradiction]

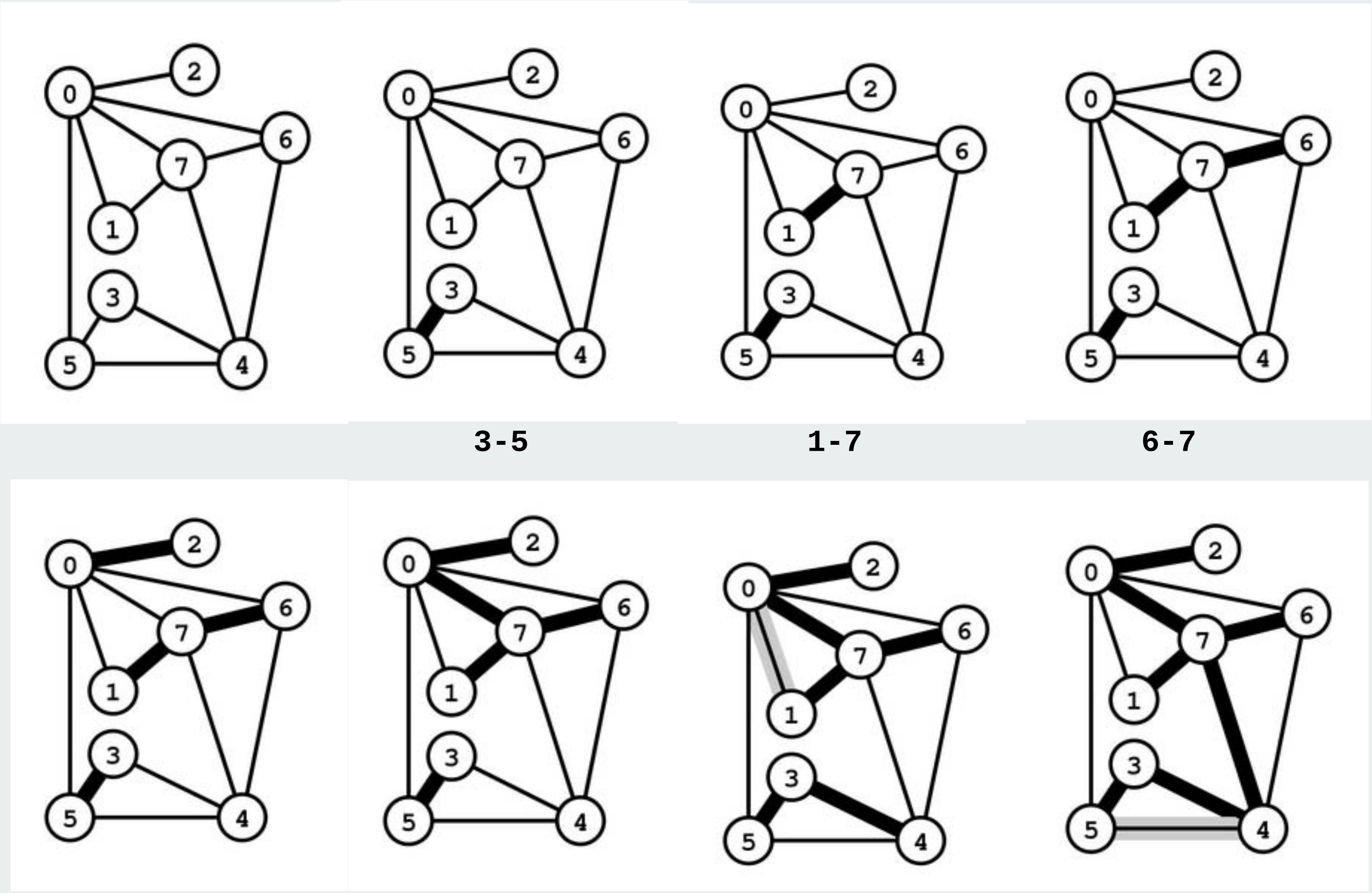
- Suppose e does not belong to T*. Let's see what happens.
- Adding e to T* creates a (unique) cycle C in T*.
- Some other edge in C, say f, has exactly one endpoint in S.
- T = T* ∪ { e } – { f } is also a spanning tree.
- Since $c_e < c_f$, cost(T) < cost(T*).
- Contradicts minimality of T*. ▪



- weighted graph API
- cycles and cuts
- **Kruskal's algorithm**
- Prim's algorithm
- advanced algorithms
- clustering

Kruskal's Algorithm: Example

Kruskal's algorithm. [Kruskal, 1956] Consider edges in ascending order of cost. Add the next edge to T unless doing so would create a cycle.



3-5

1-7

6-7

0-2

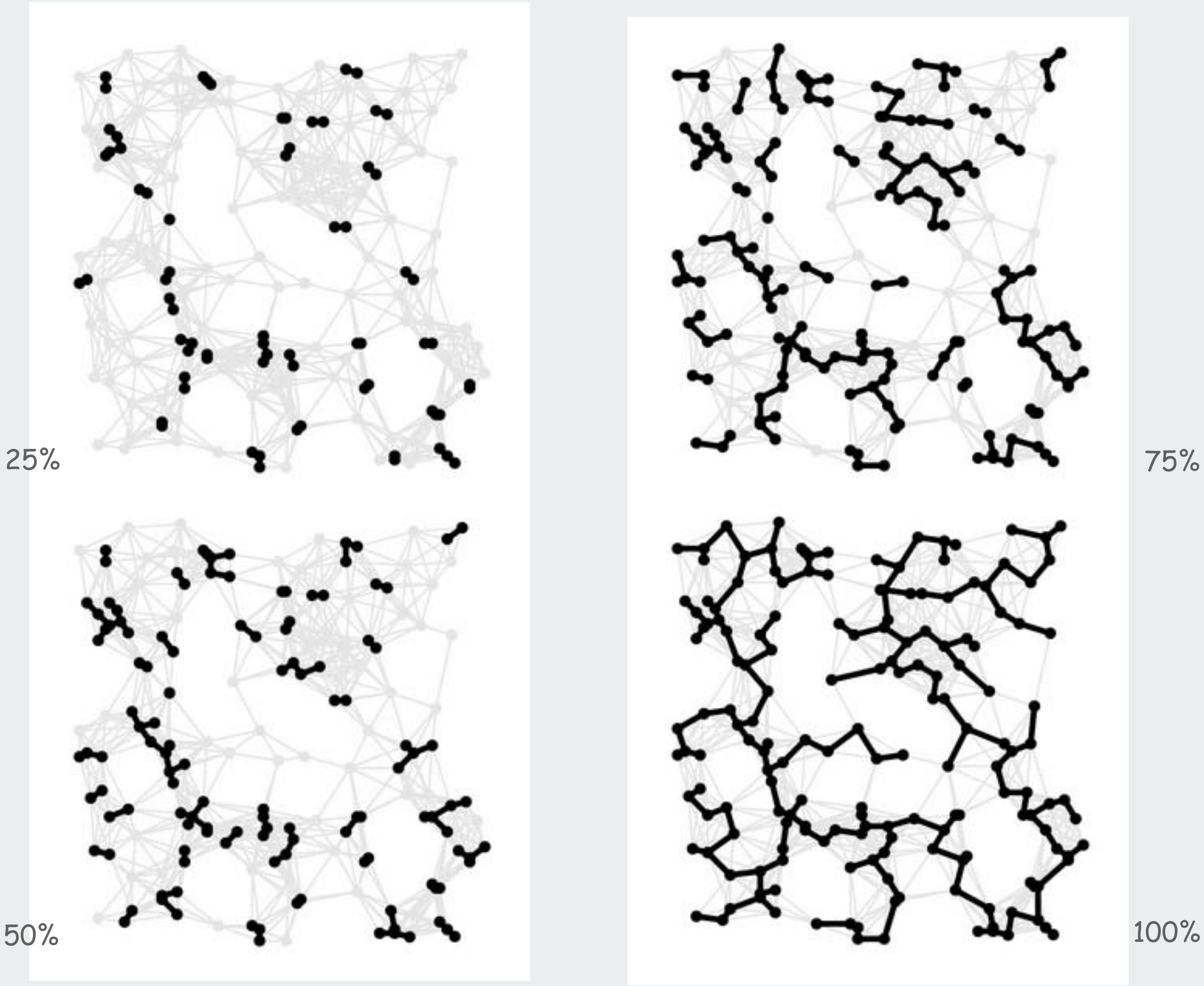
0-7

0-1 3-4

4-5 4-7

3-5	0.18
1-7	0.21
6-7	0.25
0-2	0.29
0-7	0.31
0-1	0.32
3-4	0.34
4-5	0.40
4-7	0.46
0-6	0.51
4-6	0.51
0-5	0.60

Kruskal's algorithm example

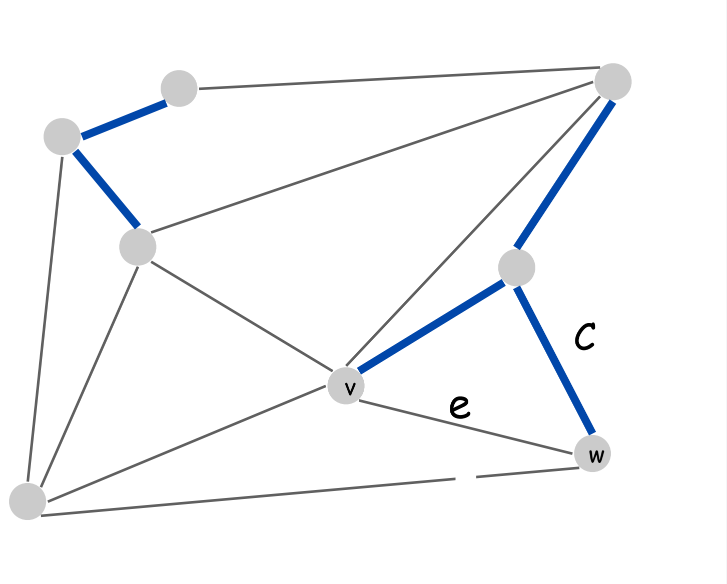


Kruskal's algorithm correctness proof

Proposition. Kruskal's algorithm computes the MST.

Pf. [case 1] Suppose that adding e to T creates a cycle C

- e is the max weight edge in C (weights come in increasing order)
- e is not in the MST (cycle property)

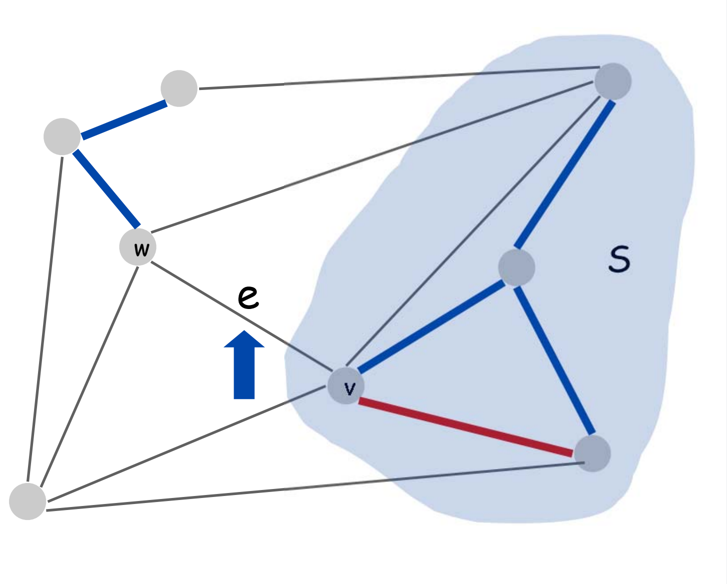


Kruskal's algorithm correctness proof

Proposition. Kruskal's algorithm computes the MST.

Pf. [case 2] Suppose that adding e = (v, w) to T does not create a cycle

- let S be the vertices in v's connected component
- w is not in S
- e is the min weight edge with exactly one endpoint in S
- e is in the MST (cut property) ■



Kruskal's algorithm implementation

Q. How to check if adding an edge to T would create a cycle?

A1. Naïve solution: use DFS.

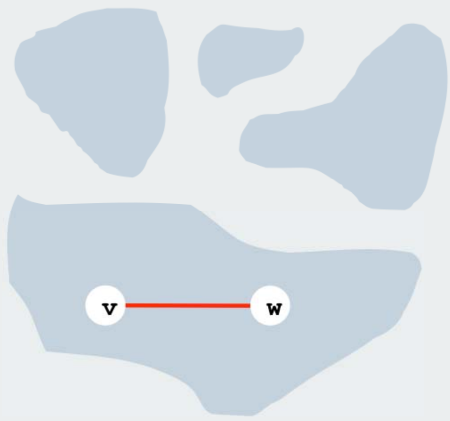
- O(V) time per cycle check.
- O(E V) time overall.

Kruskal's algorithm implementation

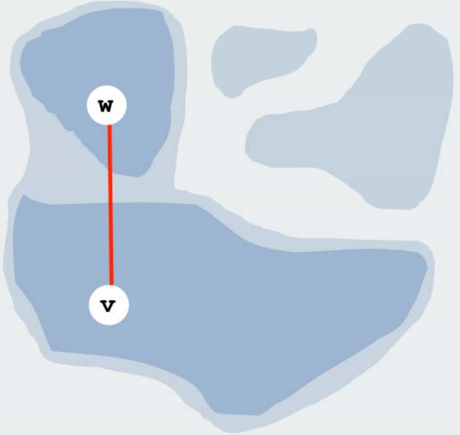
Q. How to check if adding an edge to T would create a cycle?

A2. Use the union-find data structure from lecture 1 (!).

- Maintain a set for each connected component.
- If v and w are in same component, then adding v-w creates a cycle.
- To add v-w to T, merge sets containing v and w.



Case 1: adding v-w creates a cycle



Case 2: add v-w to T and merge sets

Kruskal's algorithm: Java implementation

```
public class Kruskal
{
   private SET<Edge> mst = new SET<Edge>();

   public Kruskal(WeightedGraph G)
   {
      Edge[] edges = G.edges();
      Arrays.sort(edges, Edge.BY_WEIGHT);

      UnionFind uf = new UnionFind(G.V());
      for (Edge e: edges)
         if (!uf.find(e.either(), e.other()))
         {
            uf.unite(e.either(), e.other());
            mst.add(edge);
         }

   }

   public Iterable<Edge> mst()
   {  return mst;  }
}
```

sort edges
by weight

greedily add
edges to MST

return to client iterable
sequence of edges

Easy speedup: Stop as soon as there are V-1 edges in MST.

Kruskal's algorithm running time

Kruskal running time: Dominated by the cost of the sort.

Operation	Frequency	Time per op
sort	1	E log E
union	V	log* V †
find	E	log* V †

† amortized bound using weighted quick union with path compression

recall: log* V ≤ 5 in this universe

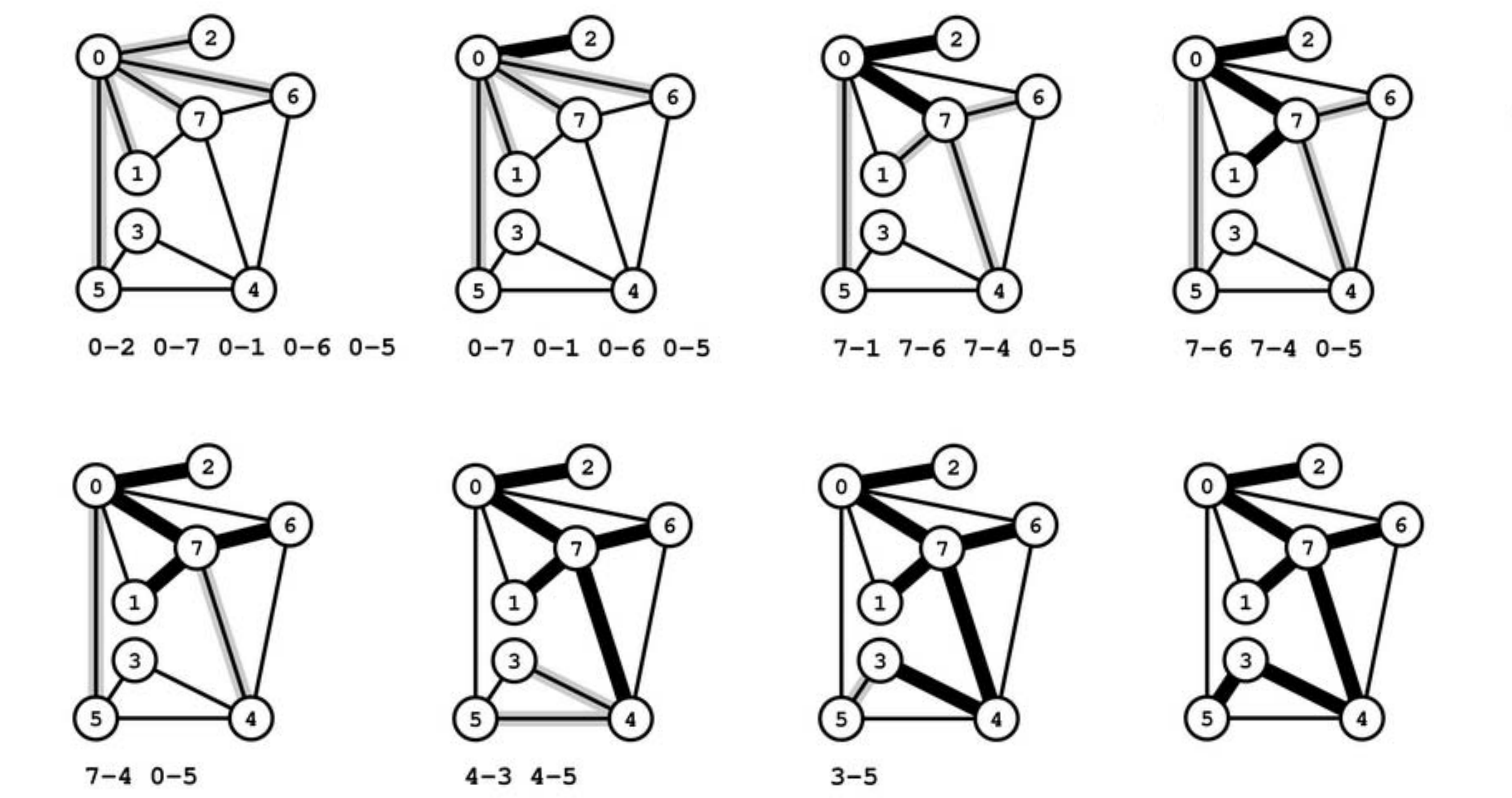
Remark 1. If edges are already sorted, time is proportional to E log* V

Remark 2. Linear in practice with PQ or quicksort partitioning
(see book: don't need full sort)

- weighted graph API
- cycles and cuts
- Kruskal’s algorithm
- **Prim’s algorithm**
- advanced topics

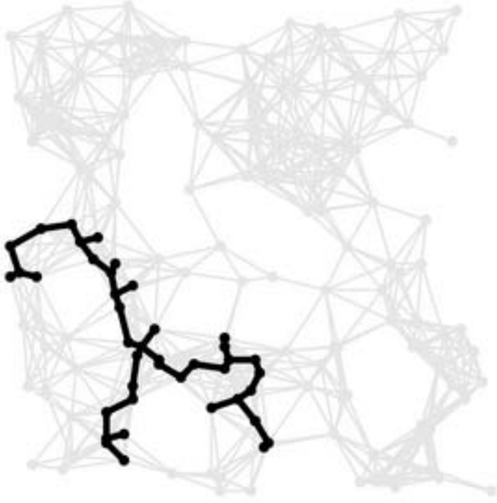
Prim's algorithm example

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]
Start with vertex 0 and greedily grow tree T. At each step, add cheapest edge that has exactly one endpoint in T.

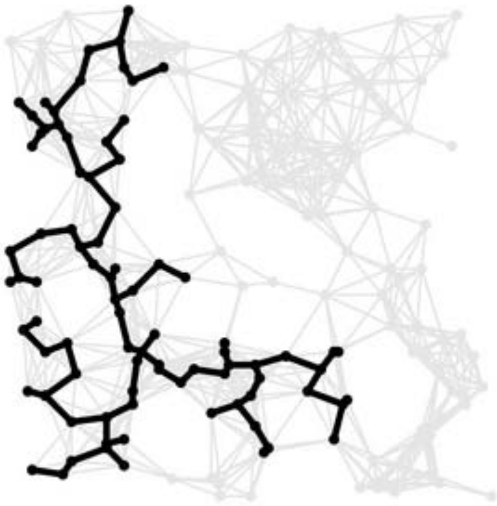


0-1	0.32
0-2	0.29
0-5	0.60
0-6	0.51
0-7	0.31
1-7	0.21
3-4	0.34
3-5	0.18
4-5	0.40
4-6	0.51
4-7	0.46
6-7	0.25

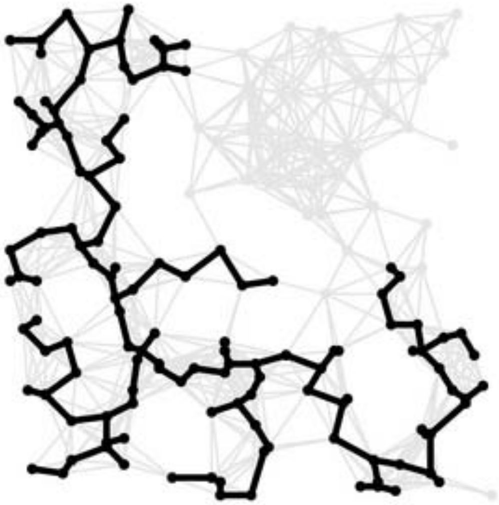
Prim's Algorithm example



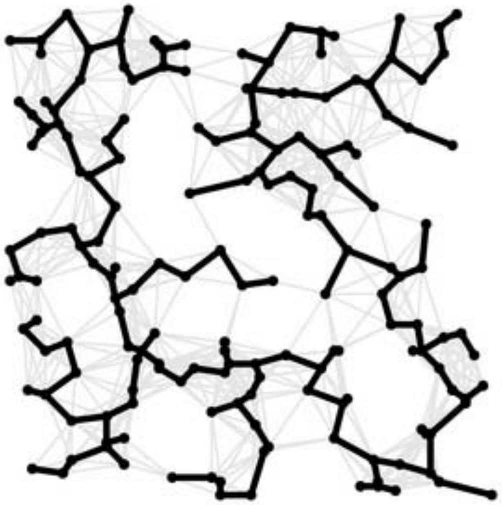
25%



50%



75%



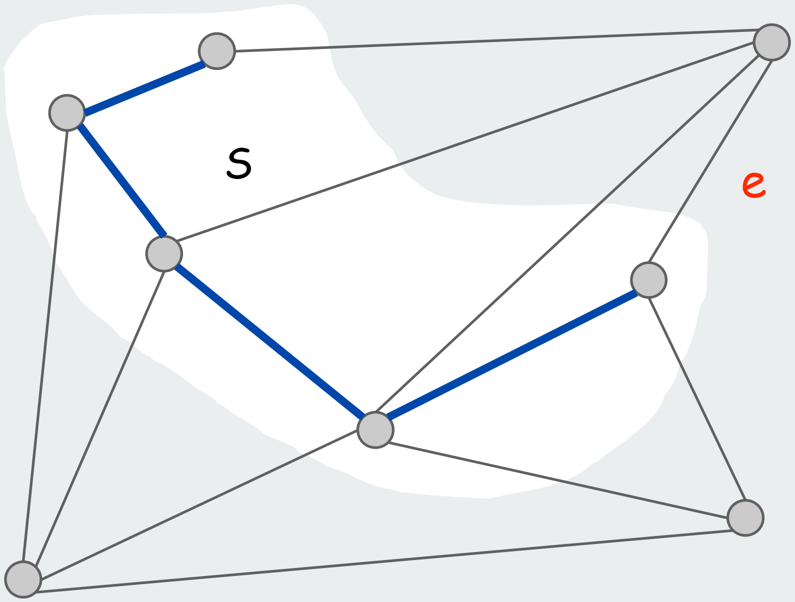
100%

Prim's algorithm correctness proof

Proposition. Prim's algorithm computes the MST.

Pf.

- Let S be the subset of vertices in current tree T.
- Prim adds the cheapest edge e with exactly one endpoint in S.
- e is in the MST (cut property) ■



Prim's algorithm implementation

Q. How to find cheapest edge with exactly one endpoint in S?

A1. Brute force: try all edges.

- O(E) time per spanning tree edge.
- O(E V) time overall.

Prim's algorithm implementation

Q. How to find cheapest edge with exactly one endpoint in S?

A2. Maintain a priority queue of vertices connected by an edge to S

- Delete min to determine next vertex v to add to S.
- Disregard v if already in S.
- Add to PQ any vertex brought closer to S by v.

Running time.

- log V steps per edge (using a binary heap).
- E log V steps overall.

Note: This is a lazy version of implementation in Algs in Java

lazy: put all adjacent vertices (that are not already in MST) on PQ
eager: first check whether vertex is already on PQ and decrease its key

Key-value priority queue

Associate a value with each key in a priority queue.

API:

```
public class MinPQplus<Key extends Comparable<Key>, Value>
```

	`MinPQplus()`	create a key-value priority queue
`void`	`put(Key key, Value val)`	put key-value pair into the priority queue
`Value`	`delMin()`	return value paired with minimal key
`Key`	`min()`	return minimal key

Implementation:

- start with same code as standard heap-based priority queue
- use a parallel array `vals[]` (value associated with `keys[i]` is `vals[i]`)
- modify `exch()` to maintain parallel arrays (do exch in `vals[]`)
- modify `delMin()` to return `Value`
- add `min()` (just returns `keys[1]`)

Lazy implementation of Prim's algorithm

```
public class LazyPrim
{
   Edge[] pred = new Edge[G.V()];
   public LazyPrim(WeightedGraph G)
   {
      boolean[] marked = new boolean[G.V()];
      double[] dist = new double[G.V()];
      MinPQplus<Double, Integer> pq;
      pq = new MinPQplus<Double, Integer>();
      dist[s] = 0.0;
      marked[s] = true;
      pq.put(dist[s], s);
      while (!pq.isEmpty())
      {
         int v = pq.delMin();
         if (marked[v]) continue;
         marked(v) = true;
         for (Edge e : G.adj(v))
         {
            int w = e.other(v);
            if (!done[w] && (dist[w] > e.weight()))
            {
               dist[w] = e.weight(); pred[w] = e;
               pq.insert(dist[w], w);
            }
         }
      }
   }
}
```

pred[v] is edge attaching **v** to MST

marks vertices in MST

distance to MST

key-value PQ

get next vertex

ignore if already in MST

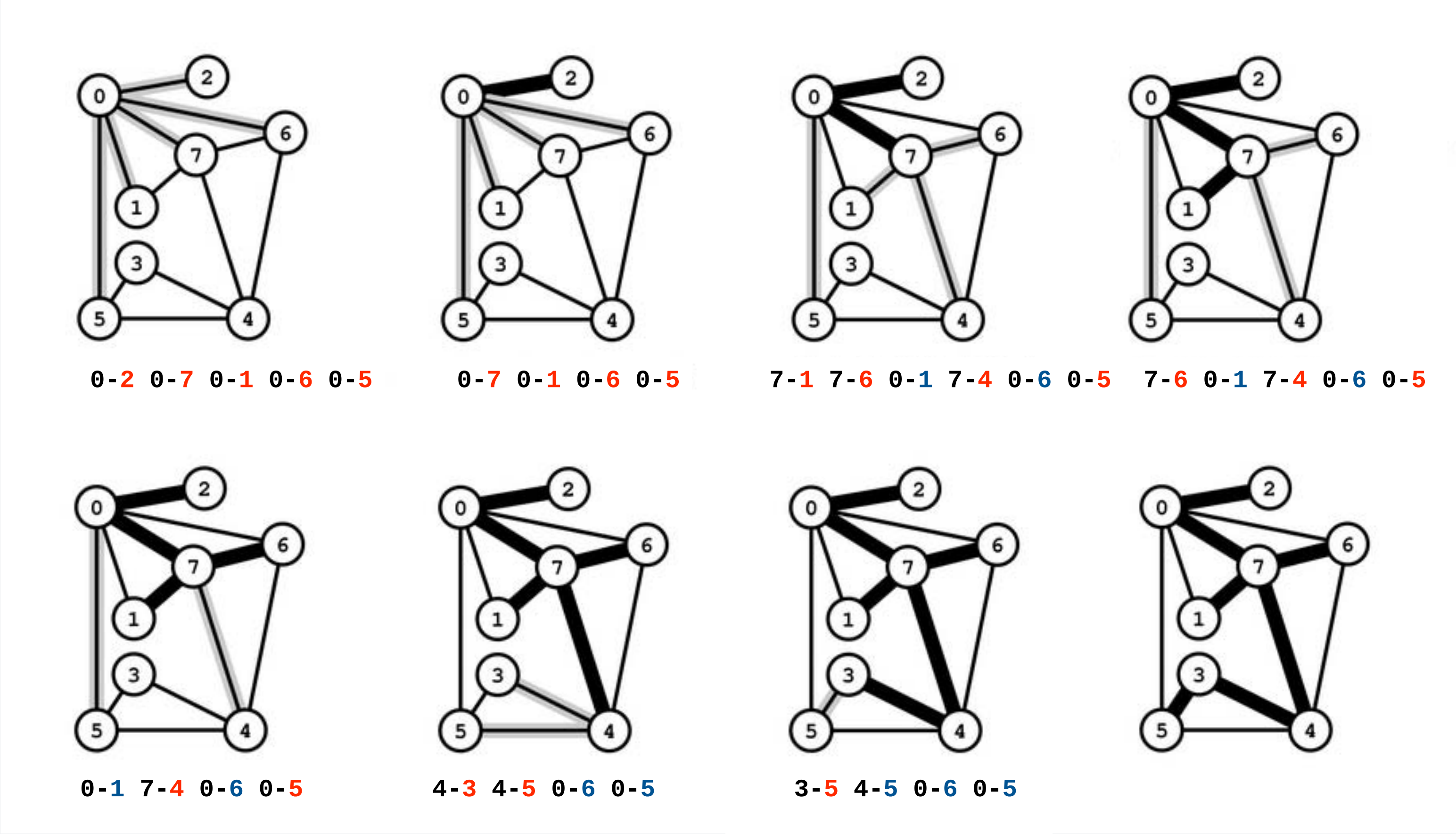
add to PQ any vertices brought closer to S by v

Prim's algorithm (lazy) example

Priority queue key is distance (edge weight); value is vertex

Lazy version leaves obsolete entries in the PQ

therefore may have multiple entries with same value



0-2 0-7 0-1 0-6 0-5

0-7 0-1 0-6 0-5

7-1 7-6 0-1 7-4 0-6 0-5

7-6 0-1 7-4 0-6 0-5

0-1 7-4 0-6 0-5

4-3 4-5 0-6 0-5

3-5 4-5 0-6 0-5

0-1	0.32
0-2	0.29
0-5	0.60
0-6	0.51
0-7	0.31
1-7	0.21
3-4	0.34
3-5	0.18
4-5	0.40
4-6	0.51
4-7	0.46
6-7	0.25

red: pq value (vertex)
blue: obsolete value

Eager implementation of Prim's algorithm

Use indexed priority queue that supports

- contains: is there a key associated with value v in the priority queue?
- decrease key: decrease the key associated with value v

[more complicated data structure, see text]

Putative "benefit": reduces PQ size guarantee from E to V

- not important for the huge sparse graphs found in practice
- PQ size is far smaller in practice
- widely used, but practical utility is debatable

Removing the distinct edge costs assumption

Simplifying assumption. All edge weights w_e are distinct.

Fact. Prim and Kruskal don't actually rely on the assumption
(our proof of correctness does)

Suffices to introduce tie-breaking rule for `compare()`.

Approach 1:

```
public int compare(Edge e, Edge f)
{
   if (e.weight < f.weight) return -1;
   if (e.weight > f.weight) return +1;
   if (e.v < f.v) return -1;
   if (e.v > f.v) return +1;
   if (e.w < f.w) return -1;
   if (e.w > f.w) return +1;
   return  0;
}
```

Approach 2: add tiny random perturbation.

- weighted graph API
- cycles and cuts
- Kruskal’s algorithm
- Prim’s algorithm
- **advanced topics**

Advanced MST theorems: does an algorithm with a linear-time guarantee exist?

Year	Worst Case	Discovered By
1975	$E \log \log V$	Yao
1976	$E \log \log V$	Cheriton-Tarjan
1984	$E \log^* V$, $E + V \log V$	Fredman-Tarjan
1986	$E \log (\log^* V)$	Gabow-Galil-Spencer-Tarjan
1997	$E\, \alpha(V) \log \alpha(V)$	Chazelle
2000	$E\, \alpha(V)$	Chazelle
2002	optimal	Pettie-Ramachandran
20xx	E	???

deterministic comparison based MST algorithms

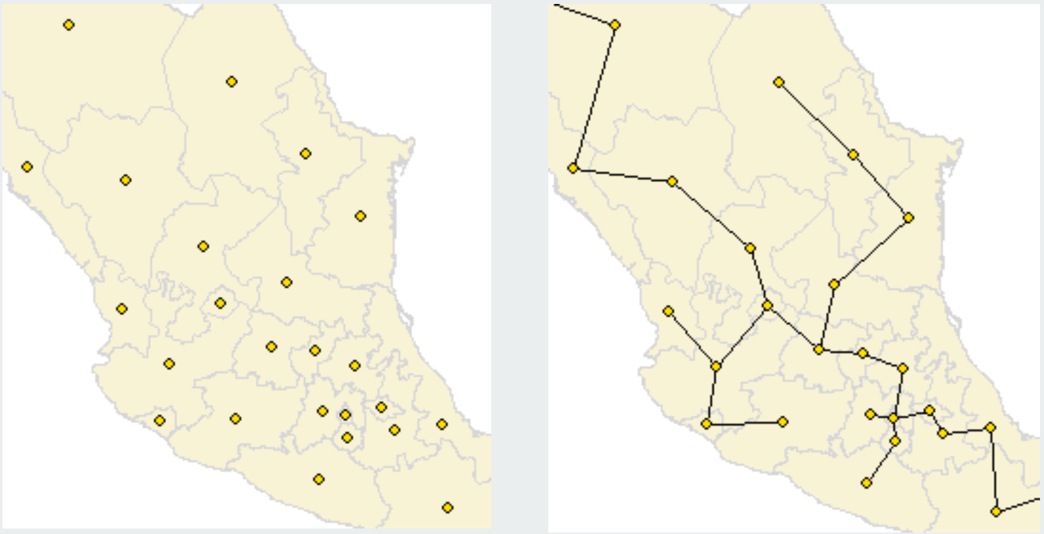
Year	Problem	Time	Discovered By
1976	Planar MST	E	Cheriton-Tarjan
1992	MST Verification	E	Dixon-Rauch-Tarjan
1995	Randomized MST	E	Karger-Klein-Tarjan

related problems

Euclidean MST

Euclidean MST. Given N points in the plane, find MST connecting them.

- Distances between point pairs are Euclidean distances.



Brute force. Compute $N^2 / 2$ distances and run Prim's algorithm.

Ingenuity. Exploit geometry and do it in O(N log N)

[stay tuned for geometric algorithms]

Scientific application: clustering

k-clustering. Divide a set of objects classify into k coherent groups.
distance function. numeric value specifying "closeness" of two objects.

Fundamental problem.
Divide into clusters so that points in different clusters are far apart.

Applications.
- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.
- Document categorization for web search.
- Similarity searching in medical image databases
- Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

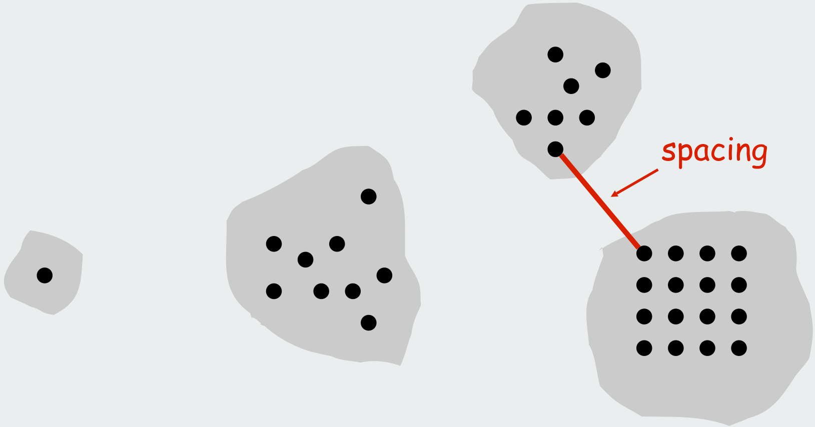
Outbreak of cholera deaths in London in 1850s.
Reference: Nina Mishra, HP Labs

k-clustering of maximum spacing

k-clustering. Divide a set of objects classify into k coherent groups.
distance function. Numeric value specifying "closeness" of two objects.

Spacing. Min distance between any pair of points in different clusters.

k-clustering of maximum spacing.
Given an integer k, find a k-clustering such that spacing is maximized.



k = 4

Single-link clustering algorithm

"Well-known" algorithm for single-link clustering:

- Form V clusters of one object each.
- Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
- Repeat until there are exactly k clusters.

Observation. This procedure is precisely Kruskal's algorithm (stop when there are k connected components).

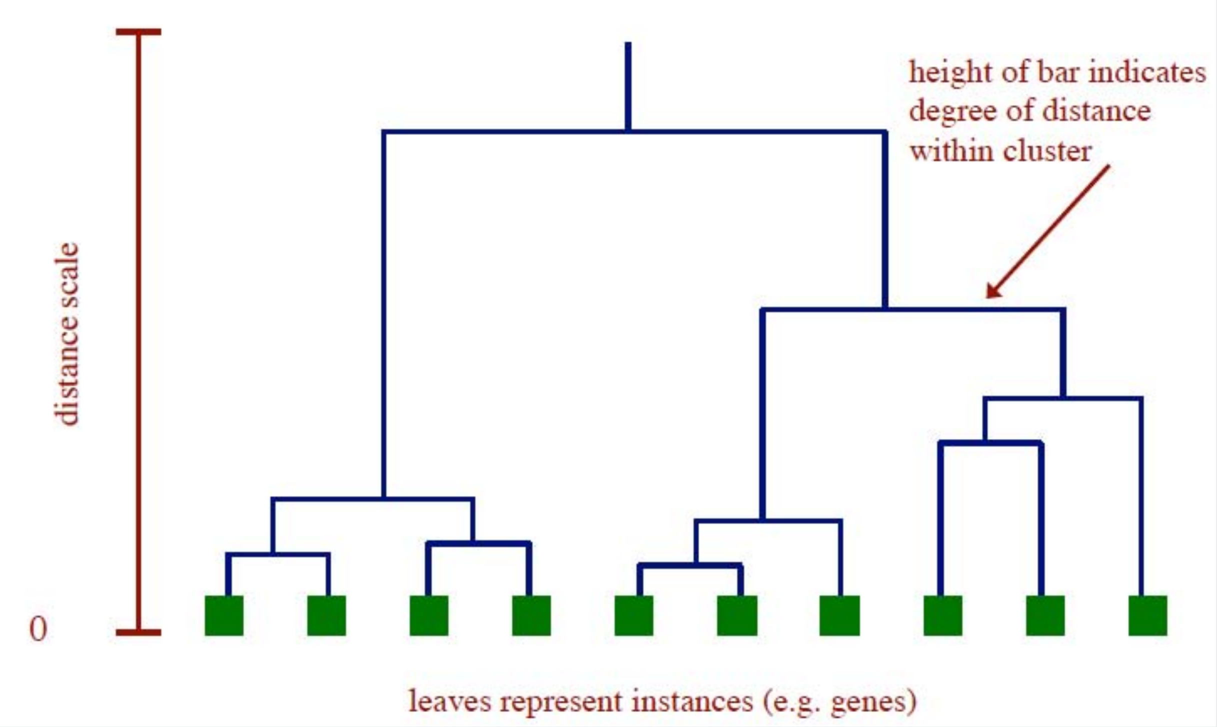
Property. Kruskal's algorithm finds a k-clustering of maximum spacing.

Clustering application: dendrograms

Dendrogram.

Scientific visualization of hypothetical sequence of evolutionary events.

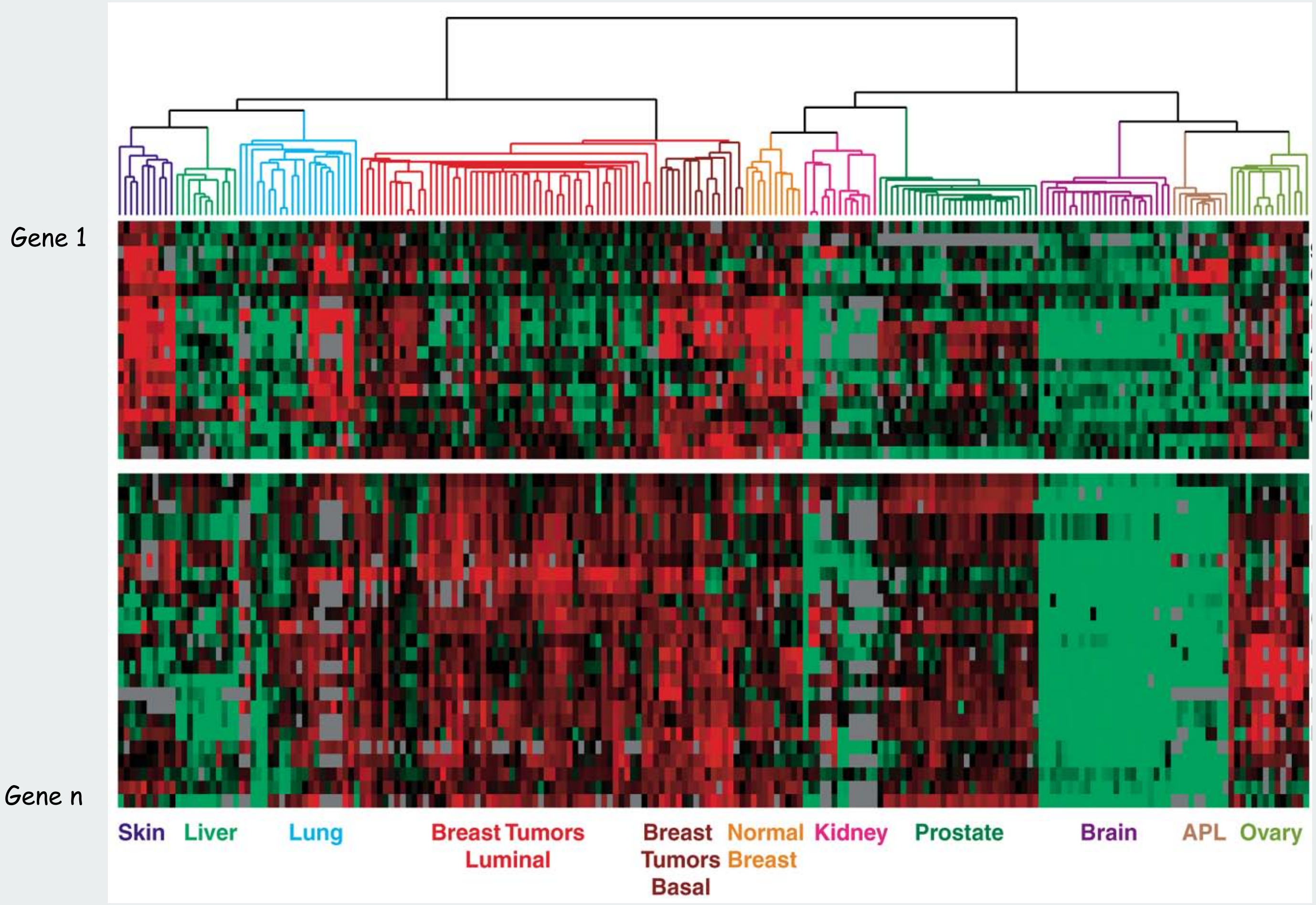
- Leaves = genes.
- Internal nodes = hypothetical ancestors.


distance scale
0
height of bar indicates degree of distance within cluster
leaves represent instances (e.g. genes)

Reference: http://www.biostat.wisc.edu/bmi576/fall-2003/lecture13.pdf

Dendrogram of cancers in human

Tumors in similar tissues cluster together.



Reference: Botstein & Brown group

⬜ gene expressed

🟩 gene not expressed

Shortest Paths

- Dijkstra's algorithm
- implementation
- negative weights

References:

Algorithms in Java, Chapter 21

http://www.cs.princeton.edu/introalgsds/55dijkstra

Edsger W. Dijkstra: a few select quotes

The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

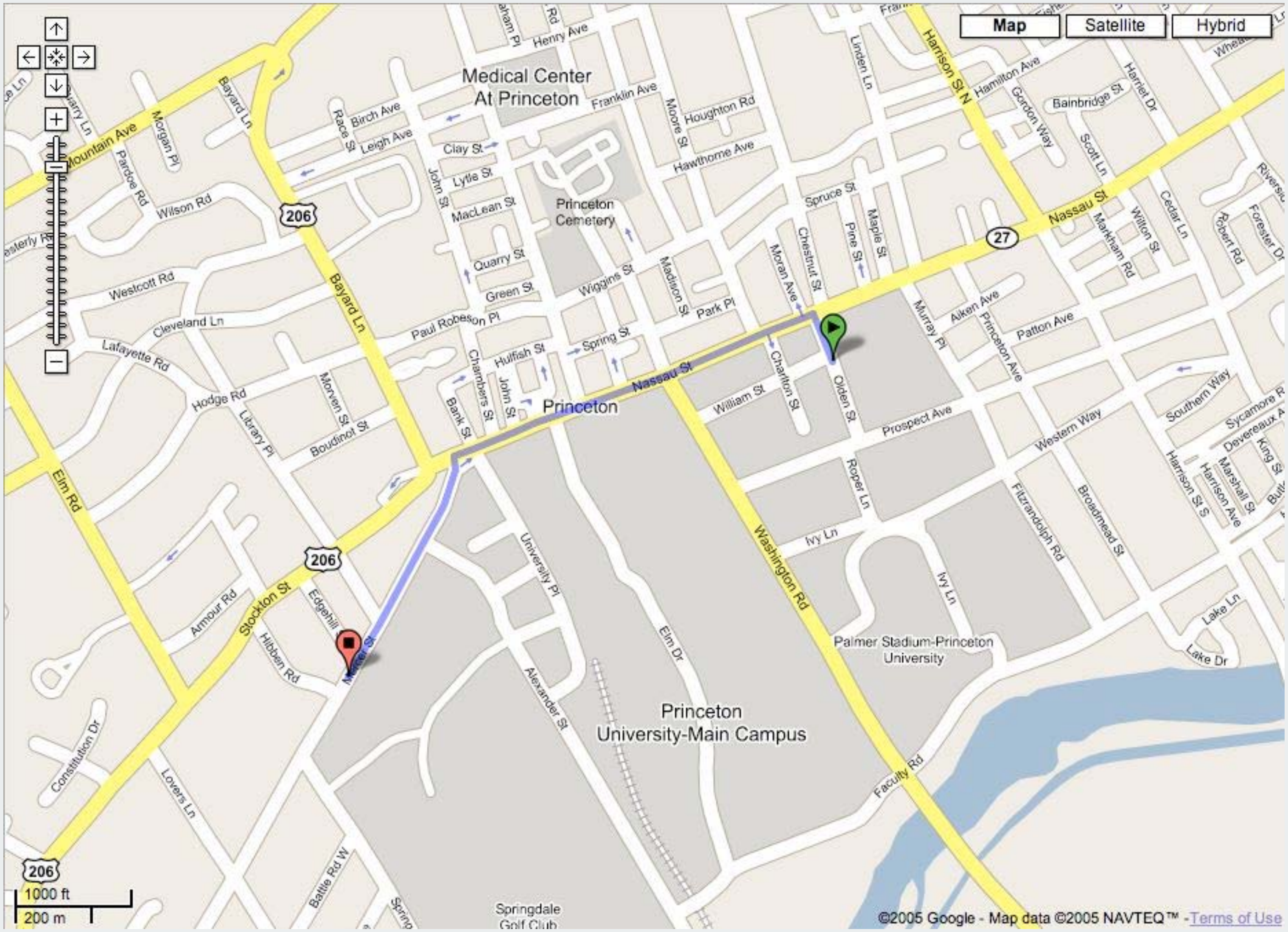
In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.

Edger Dijkstra
Turing award 1972

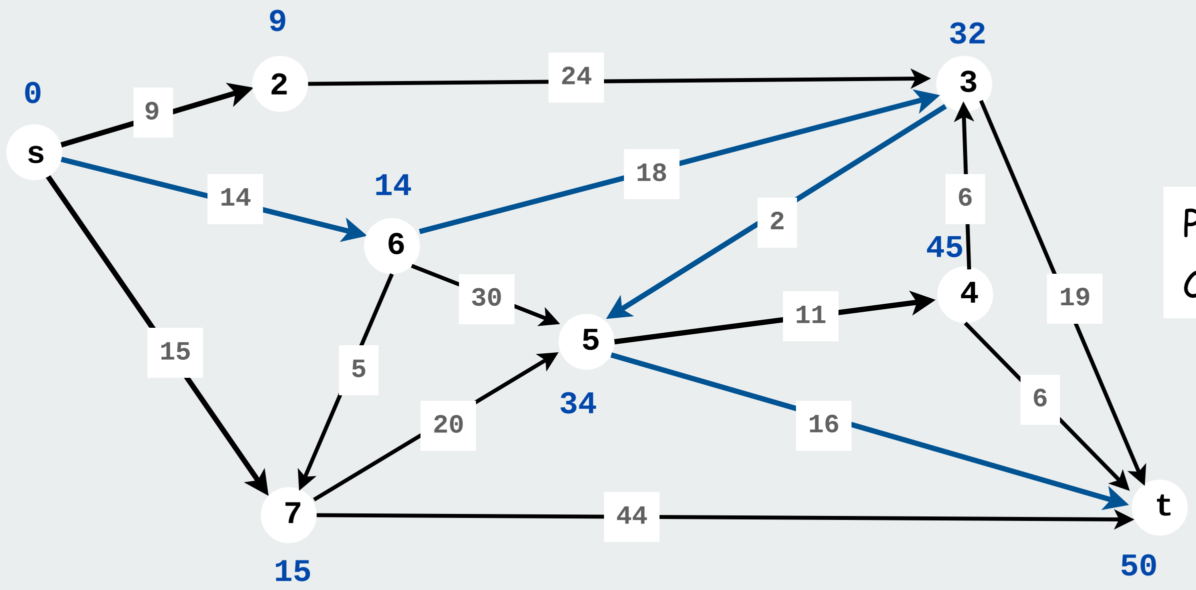
Shortest paths in a weighted digraph



Shortest paths in a weighted digraph

Given a weighted digraph, find the shortest directed path from s to t.

cost of path = sum of edge costs in path



Path: **s→6→3→5→t**

Cost: 14 + 18 + 2 + 16 = 50

Note: weights are arbitrary numbers

- not necessarily distances
- need not satisfy the triangle inequality
- Ex: airline fares [stay tuned for others]

Versions

- source-target (s-t)
- single source
- all pairs.
- nonnegative edge weights
- arbitrary weights
- Euclidean weights.

Early history of shortest paths algorithms

Shimbel (1955). Information networks.

Ford (1956). RAND, economics of transportation.

Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, Seitz (1957).
Combat Development Dept. of the Army Electronic Proving Ground.

Dantzig (1958). Simplex method for linear programming.

Bellman (1958). Dynamic programming.

Moore (1959). Routing long-distance telephone calls for Bell Labs.

Dijkstra (1959). Simpler and faster version of Ford's algorithm.

Applications

Shortest-paths is a broadly useful problem-solving model

- Maps
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Subroutine in advanced algorithms.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Approximating piecewise linear functions.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

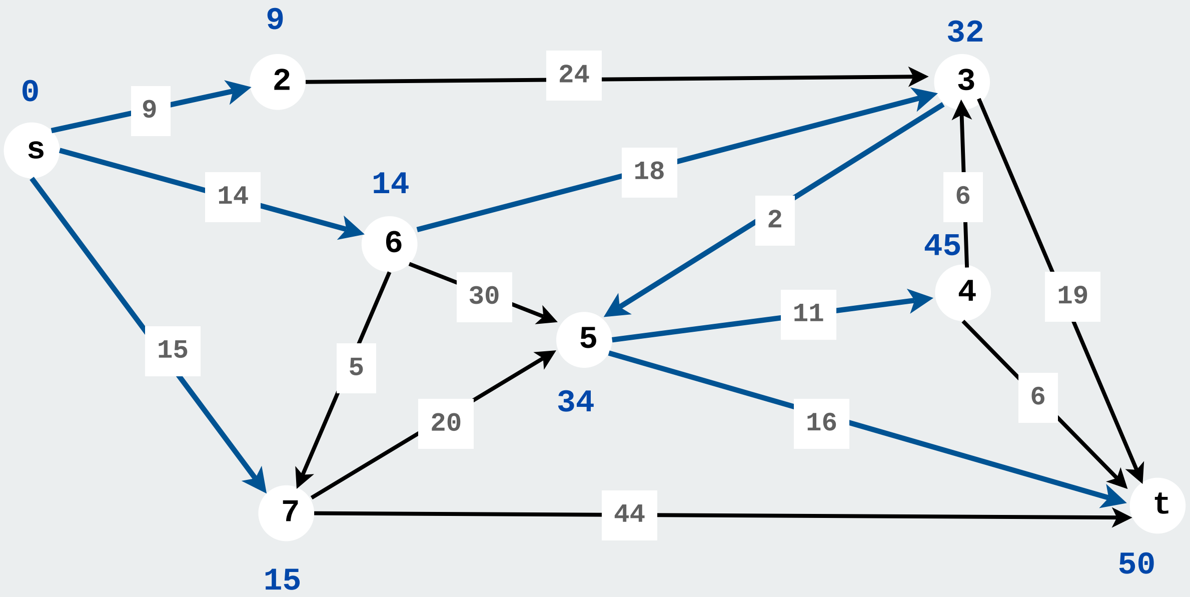
- Dijkstra's algorithm
- implementation
- negative weights

Single-source shortest-paths

Given. Weighted digraph, single source s.

Distance from s to v: length of the shortest path from s to v .

Goal. Find distance (and shortest path) from s to every other vertex.



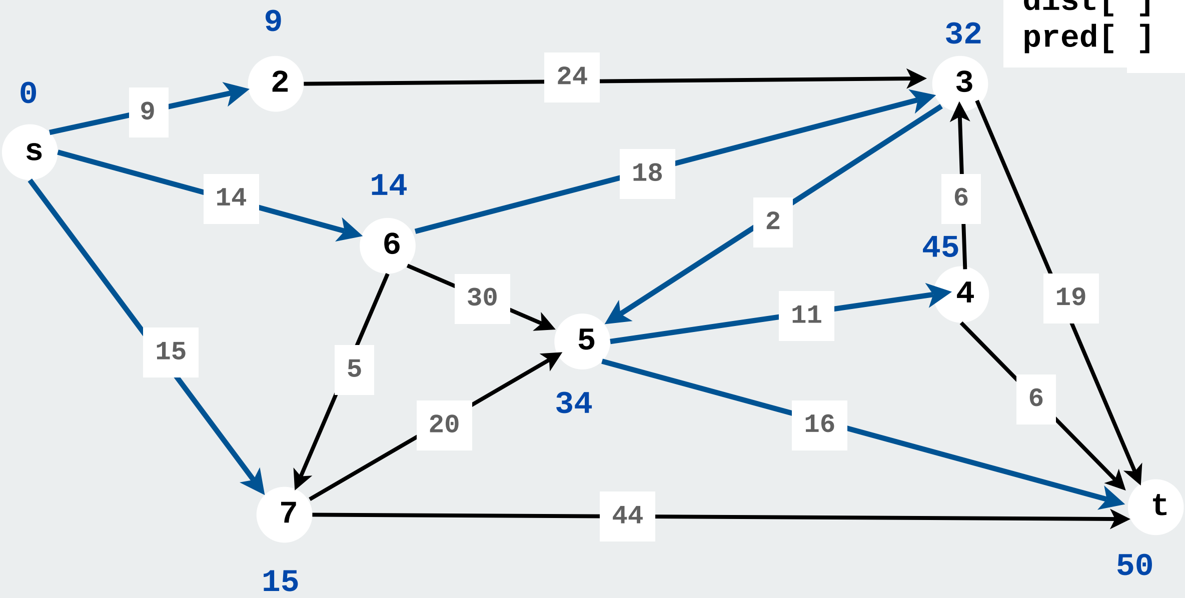
Shortest paths form a tree

Single-source shortest-paths: basic plan

Goal: Find distance (and shortest path) from s to every other vertex.

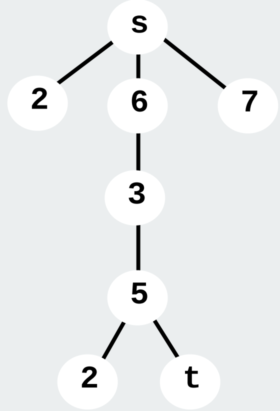
Design pattern:

- **`ShortestPaths`** class (**`WeightedDigraph`** client)
- instance variables: vertex-indexed arrays **`dist[]`** and **`pred[]`**
- client query methods return distance and path iterator



v	s	2	3	4	5	6	7	t
dist[ ]	0	9	32	45	34	14	15	50
pred[ ]	0	0	6	5	3	0	0	5

shortest path tree
(parent-link representation)



Note: Same pattern as Prim, DFS, BFS; BFS works when weights are all 1.

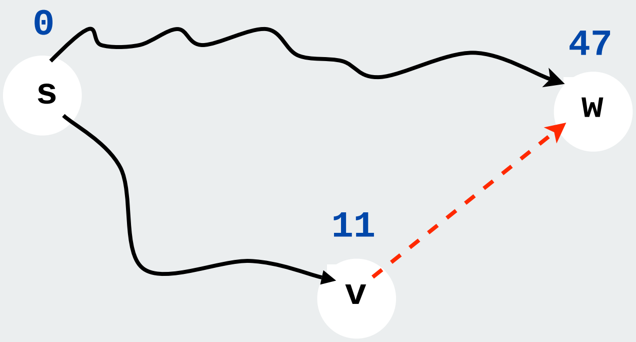
Edge relaxation

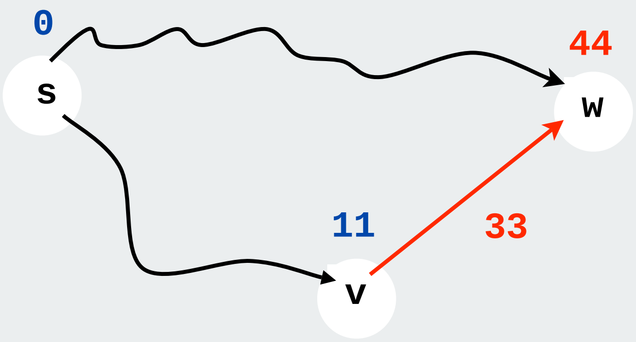
For all v, **dist[v]** is the length of some path from s to v.

Relaxation along edge **e** from v to w.

- **dist[v]** is length of some path from s to v
- **dist[w]** is length of some path from s to w
- if **v-w** gives a shorter path to w through v, update **dist[w]** and **pred[w]**

```
if (dist[w] > dist[v] + e.weight())
{
    dist[w] = dist[v] + e.weight());
    pred[w] = e;
}
```





Relaxation sets **dist[w]** to the length of a shorter path from s to w (if v-w gives one)

Dijkstra's algorithm

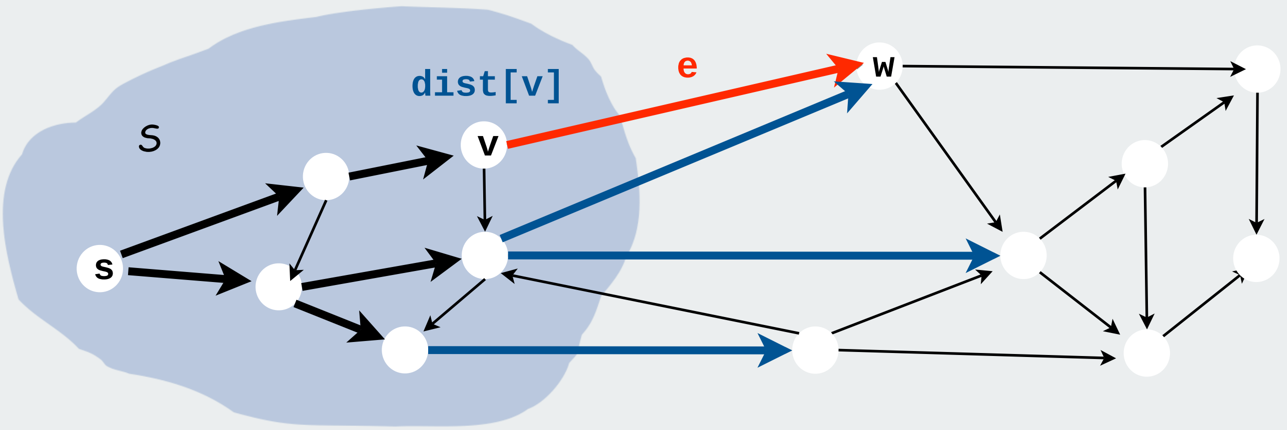
S: set of vertices for which the shortest path length from s is known.

Invariant: for v in S, **`dist[v]`** is the length of the shortest path from s to v.

Initialize S to s, **`dist[s]`** to **`0`**, **`dist[v]`** to ∞ for all other v

Repeat until S contains all vertices connected to s

- find e with v in S and w in S′ that minimizes **`dist[v] + e.weight()`**
- relax along that edge
- add w to S



Dijkstra's algorithm

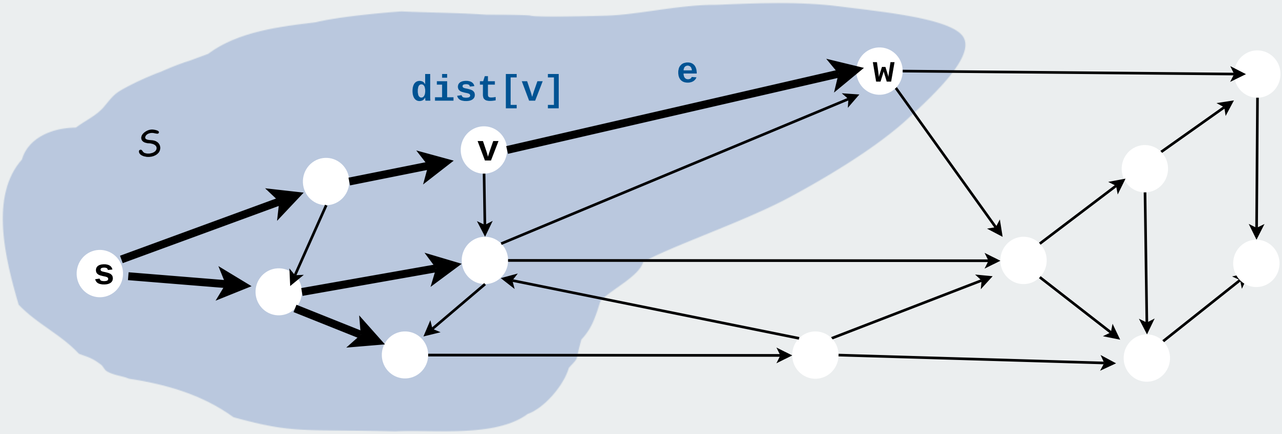
S: set of vertices for which the shortest path length from s is known.

Invariant: for v in S, `dist[v]` is the length of the shortest path from s to v.

Initialize S to s, `dist[s]` to `0`, `dist[v]` to ∞ for all other v

Repeat until S contains all vertices connected to s

- find e with v in S and w in S′ that minimizes `dist[v] + e.weight()`
- relax along that edge
- add w to S



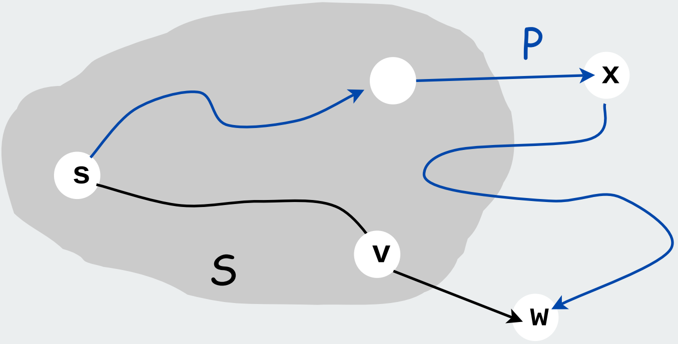
Dijkstra's algorithm proof of correctness

S: set of vertices for which the shortest path length from s is known.

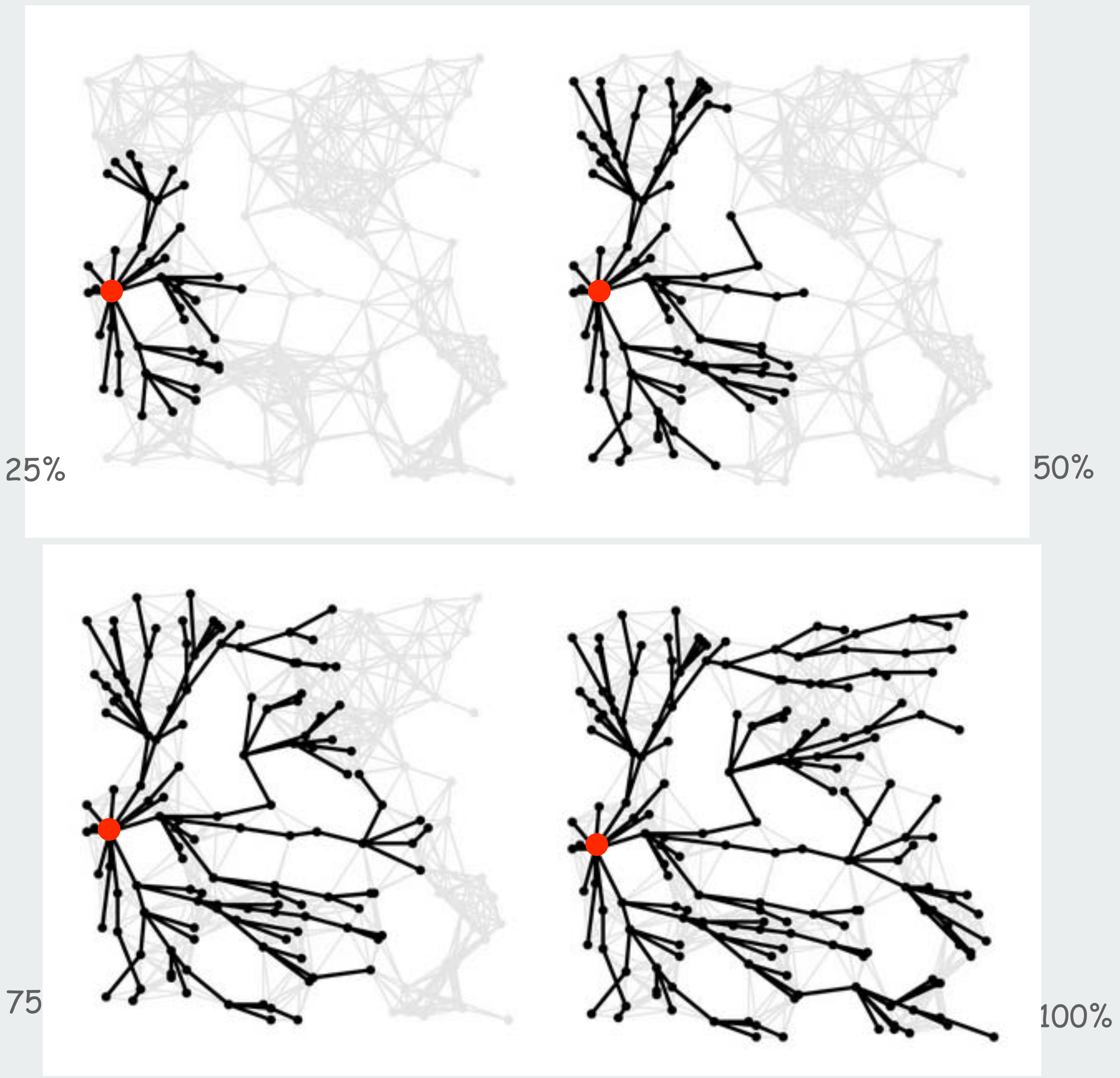
Invariant: for v in S, `dist[v]` is the length of the shortest path from s to v.

Pf. (by induction on |S|)

- Let w be next vertex added to S.
- Let P* be the s-w path through v.
- Consider any other s-w path P, and let x be first node on path outside S.
- P is already longer than P* as soon as it reaches x by greedy choice.



Shortest Path Tree



- Dijkstra’s algorithm
- **implementation**
- negative weights

Weighted directed edge data type

```
public class Edge implements Comparable<Edge>
{
   public final int v, int w;
   public final double weight;

   public Edge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int from()
   {  return v; }

   public int to()
   {  return w; }

   public int weight()
   {  return weight; }

   public int compareTo(Edge that)
   {
      if      (this.weight < that.weight) return -1;
      else if (this.weight > that.weight) return +1;
      else                                return  0;
   }
}
```

code is the same as for (undirected) **`WeightedGraph`**

except
`from()` and **`to()`** replace
`either()` and **`other()`**

Weighted digraph data type

Identical to **WeightedGraph** but just one representation of each **Edge**.

```
public class WeightedDigraph
{
   private int V;
   private SET<Edge>[] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = (SET<Edge>[]) new SET[V];
      for (int v = 0; v < V; v++)
         adj[v] = new SET<Edge>();
   }

   public void addEdge(Edge e)
   {
      int v = e.from();
      adj[v].add(e);
   }

   public Iterable<Edge> adj(int v)
   {  return adj[v];  }

}
```

Dijkstra's algorithm: implementation approach

Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v

Repeat until S contains all vertices connected to s

- find `v-w` with `v` in S and `w` in S' that minimizes `dist[v] + weight[v-w]`
- relax along that edge
- add w to S

Idea 1 (easy): Try all edges

Total running time proportional to VE

Dijkstra's algorithm: implementation approach

Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v

Repeat until S contains all vertices connected to s

- find **v-w** with **v** in S and **w** in S′ that minimizes **dist[v] + weight[v-w]**
- relax along that edge
- add w to S

Idea 2 (Dijkstra) : maintain these invariants

- for **v** in S, **dist[v]** is the length of the shortest path from s to **v**.
- for **w** in S′, **dist[w]** minimizes **dist[v] + weight[v-w]**.

Two implications

- find **v-w** in V steps (smallest **dist[]** value among vertices in S′)
- update **dist[]** in at most V steps (check neighbors of **w**)

Total running time proportional to V^2

Dijkstra's algorithm implementation

Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v

Repeat until S contains all vertices connected to s

- find **`v-w`** with **v** in S and **w** in S' that minimizes **`dist[v] + weight[v-w]`**
- relax along that edge
- add w to S

Idea 3 (modern implementations):

- for all v in S, **`dist[v]`** is the length of the shortest path from s to v.
- use a priority queue to find the edge to relax

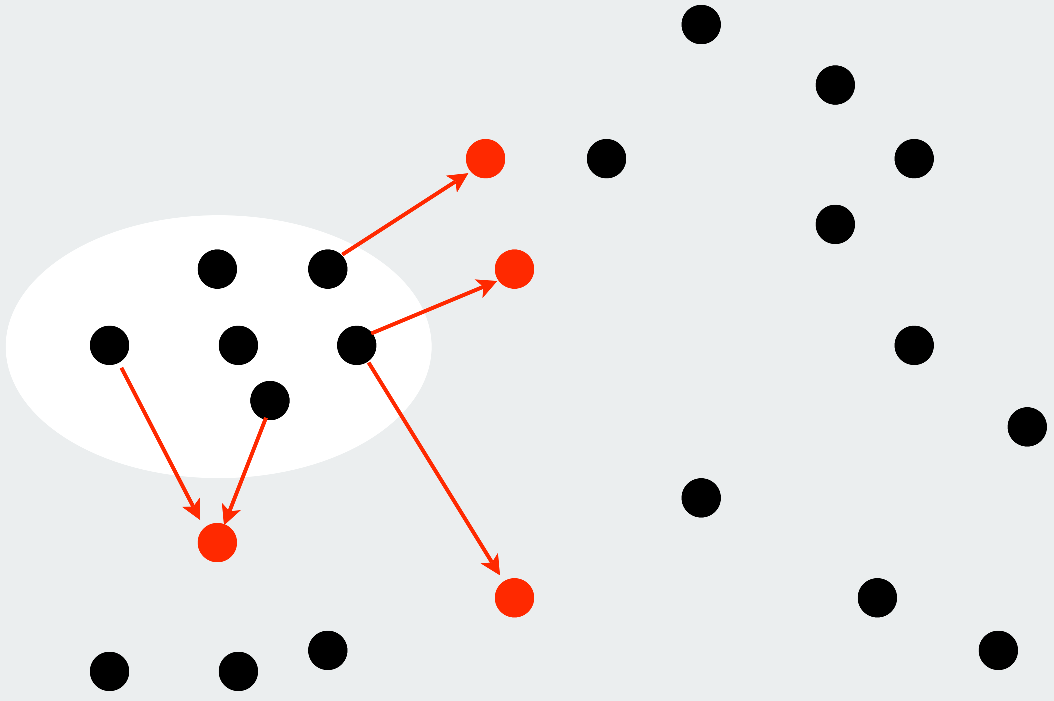
Total running time proportional to E lg E

	sparse	dense
easy	V^2	EV
Dijkstra	V^2	V^2
modern	E lg E	E lg E

Dijkstra's algorithm implementation

Q. What goes onto the priority queue?

A. Fringe vertices connected by a single edge to a vertex in S



Starting to look familiar?

Lazy implementation of Prim's MST algorithm

```
public class LazyPrim
{
   Edge[] pred = new Edge[G.V()];
   public LazyPrim(WeightedGraph G)
   {
      boolean[] marked = new boolean[G.V()];
      double[] dist = new double[G.V()];
      for (int v = 0; v < G.V(); v++)
         dist[v] = Double.POSITIVE_INFINITY;
      MinPQplus<Double, Integer> pq;
      pq = new MinPQplus<Double, Integer>();
      dist[s] = 0.0;
      pq.put(dist[s], s);
      while (!pq.isEmpty())
      {
         int v = pq.delMin();
         if (marked[v]) continue;
         marked(v) = true;
         for (Edge e : G.adj(v))
         {
            int w = e.other(v);
            if (!marked[w] && (dist[w] > e.weight() ))
            {
               dist[w] = e.weight();
               pred[w] = e;
               pq.insert(dist[w], w);
            }
         }
      }
   }
}
```

marks vertices in MST

distance to MST

edges to MST

key-value PQ

get next vertex

ignore if already in MST

add to PQ any vertices
brought closer to S by v

Lazy implementation of Dijkstra's SPT algorithm

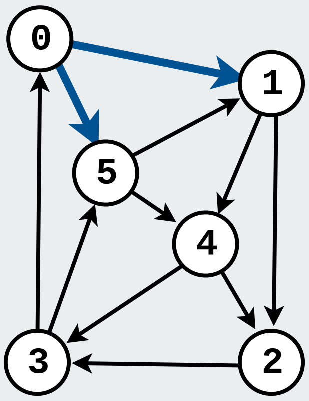
```
public class LazyDijkstra
{
   double[] dist = new double[G.V()];
   Edge[] pred = new Edge[G.V()];
   public LazyDijkstra(WeightedDigraph G, int s)
   {
      boolean[] marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         dist[v] = Double.POSITIVE_INFINITY;
      MinPQplus<Double, Integer> pq;
      pq = new MinPQplus<Double, Integer>();
      dist[s] = 0.0;
      pq.put(dist[s], s);
      while (!pq.isEmpty())
      {
         int v = pq.delMin();
         if (marked[v]) continue;
         marked(v) = true;
         for (Edge e : G.adj(v))
         {
            int w = e.to();
            if (dist[w] > dist[v] + e.weight())
            {
               dist[w] = dist[v] + e.weight();
               pred[w] = e;
               pq.insert(dist[w], w);
            }
         }
      }
   }
}
```

code is the same as Prim's (!!)

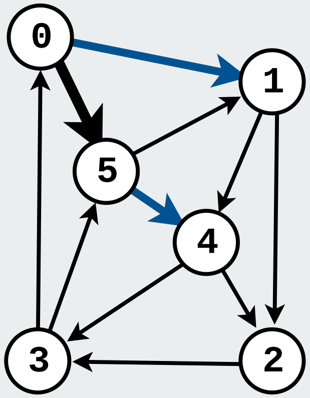
except
- **WeightedDigraph**, not **WeightedGraph**
- weight is distance to **s**, not to tree
- add client query for distances

Dijkstra's algorithm example

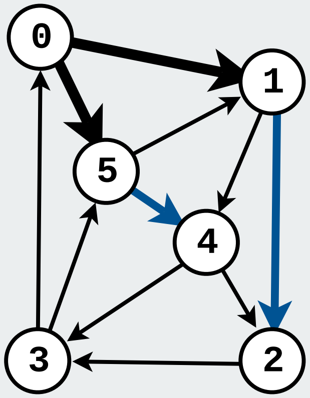
Dijkstra's algorithm. [Dijkstra 1957]
Start with vertex 0 and greedily grow tree T. At each step,
add cheapest path ending in an edge that has exactly one endpoint in T.



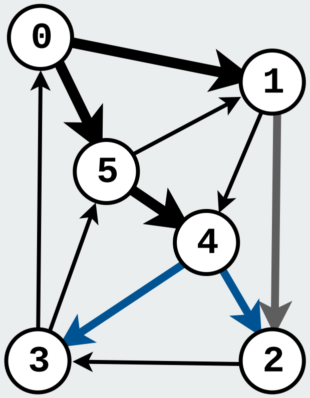
0-5 .29 0-1 .41



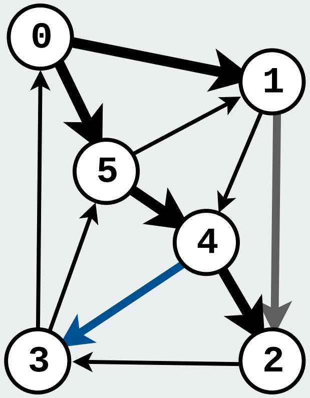
0-1 .41 5-4 .50



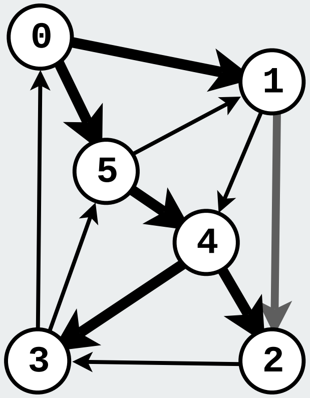
5-4 .50 1-2 .92



4-2 .82 4-3 .86 1-2 .92



4-3 .86 1-2 .92



1-2 .92

0-1	0.41
0-5	0.29
1-2	0.51
1-4	0.32
2-3	0.50
3-0	0.45
3-5	0.38
4-2	0.32
4-3	0.36
5-1	0.29
5-4	0.21

Eager implementation of Dijkstra's algorithm

Use indexed priority queue that supports

- contains: is there a key associated with value v in the priority queue?
- decrease key: decrease the key associated with value v

[more complicated data structure, see text]

Putative "benefit": reduces PQ size guarantee from E to V

- no signficant impact on time since lg E < 2lg V
- extra space not important for huge sparse graphs found in practice [PQ size is far smaller than E or even V in practice]
- widely used, but practical utility is debatable (as for Prim's)

Improvements to Dijkstra's algorithm

Use a d-way heap (Johnson, 1970s)

- easy to implement
- reduces costs to $E\ d \log_d V$
- indistinguishable from linear for huge sparse graphs found in practice

Use a Fibonacci heap (Sleator-Tarjan, 1980s)

- very difficult to implement
- reduces worst-case costs (in theory) to $E + V \lg V$
- not quite linear (in theory)
- practical utility questionable

Find an algorithm that provides a linear worst-case guarantee?
[open problem]

Dijkstra's Algorithm: performance summary

Fringe implementation directly impacts performance

Best choice depends on sparsity of graph.

- 2,000 vertices, 1 million edges. heap 2-3x slower than array
- 100,000 vertices, 1 million edges. heap gives 500x speedup.
- 1 million vertices, 2 million edges. heap gives 10,000x speedup.

Bottom line.

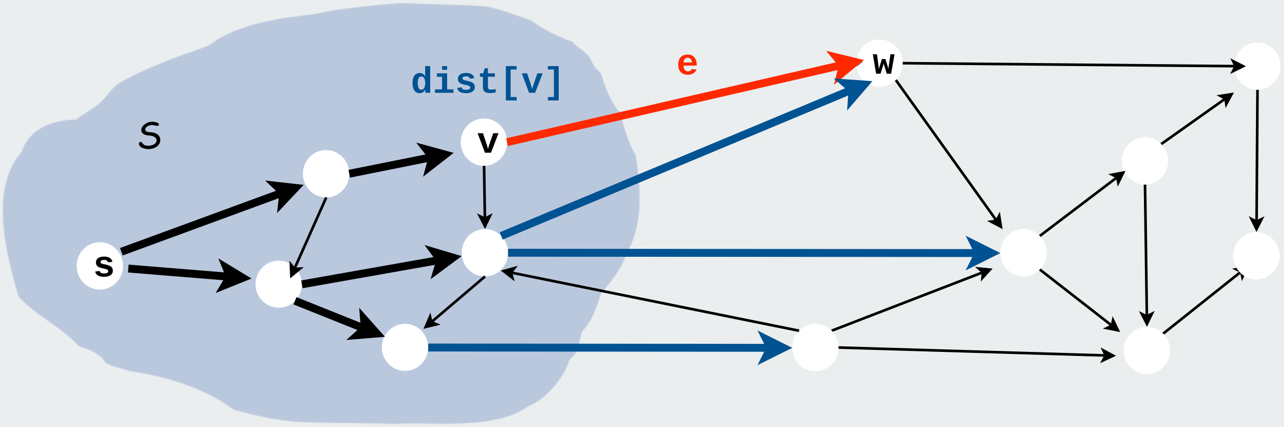
- array implementation optimal for dense graphs
- binary heap far better for sparse graphs
- d-way heap worth the trouble in performance-critical situations
- Fibonacci heap best in theory, but not worth implementing

Priority-first search

Insight: All of our graph-search methods are the same algorithm!

Maintain a set of explored vertices S
Grow S by exploring edges with exactly one endpoint leaving S.

DFS. Take edge from vertex which was discovered most recently.
BFS. Take from vertex which was discovered least recently.
Prim. Take edge of minimum weight.
Dijkstra. Take edge to vertex that is closest to s.
... Gives simple algorithm for many graph-processing problems

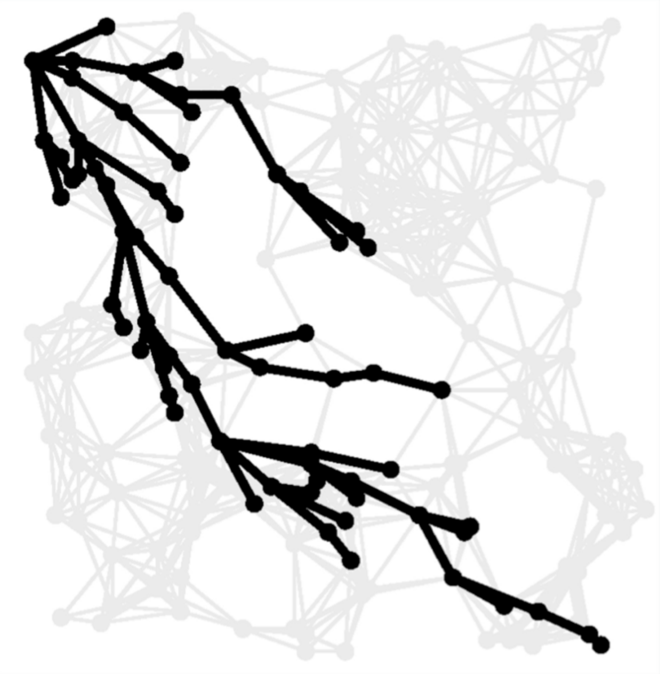


Challenge: express this insight in (re)usable Java code

Priority-first search: application example

Shortest `s-t` paths in Euclidean graphs (maps)

- Vertices are points in the plane.
- Edge weights are Euclidean distances.



A sublinear algorithm.

- Assume graph is already in memory.
- Start Dijkstra at s.
- Stop when you reach `t`.

Even better: exploit geometry

- For edge `v-w`, use weight `d(v, w) + d(w, t) – d(v, t)`.
- Proof of correctness for Dijkstra still applies.
- In practice only O($V^{1/2}$) vertices examined.
- Special case of A* algorithm

Euclidean distance

[Practical map-processing programs precompute many of the paths.]

- Dijkstra's algorithm
- implementation
- negative weights

Shortest paths application: Currency conversion

Currency conversion. Given currencies and exchange rates, what is best way to convert one ounce of gold to US dollars?

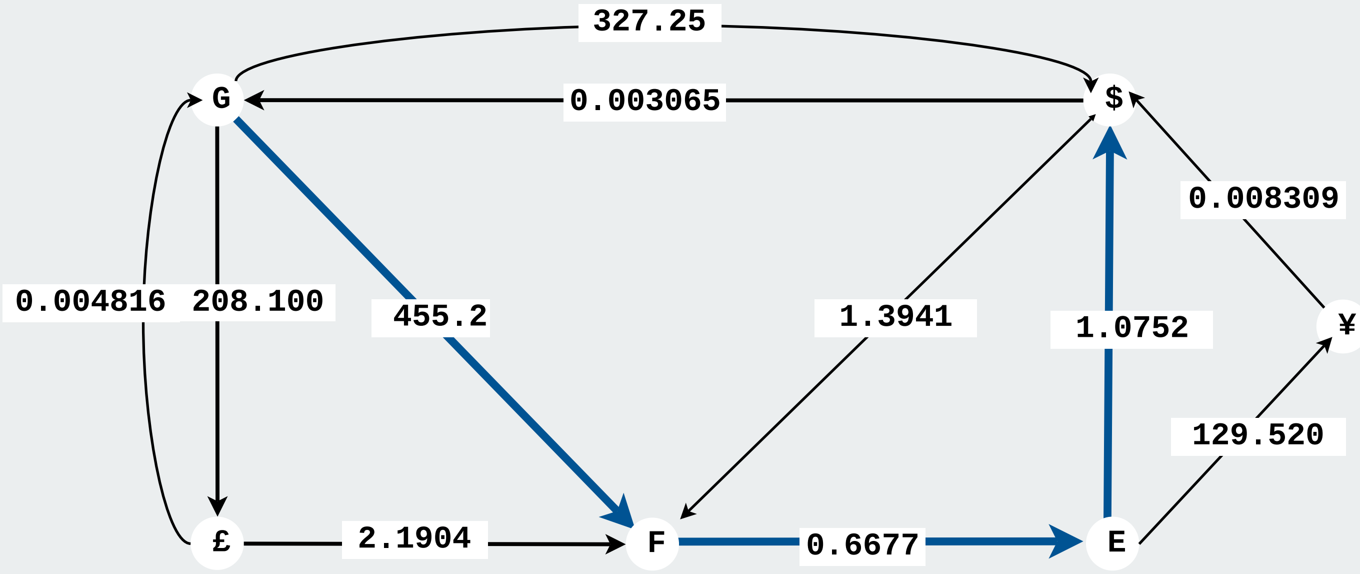
- 1 oz. gold ⇒ $327.25.
- 1 oz. gold ⇒ £208.10 ⇒ ⇒ $327.00. [208.10 × 1.5714]
- 1 oz. gold ⇒ 455.2 Francs ⇒ 304.39 Euros ⇒ $327.28. [455.2 × .6677 × 1.0752]

Currency	£	Euro	¥	Franc	$	Gold
UK Pound	1.0000	0.6853	0.005290	0.4569	0.6368	208.100
Euro	1.4599	1.0000	0.007721	0.6677	0.9303	304.028
Japanese Yen	189.050	129.520	1.0000	85.4694	120.400	39346.7
Swiss Franc	2.1904	1.4978	0.011574	1.0000	1.3941	455.200
US Dollar	1.5714	1.0752	0.008309	0.7182	1.0000	327.250
Gold (oz.)	0.004816	0.003295	0.0000255	0.002201	0.003065	1.0000

Shortest paths application: Currency conversion

Graph formulation.

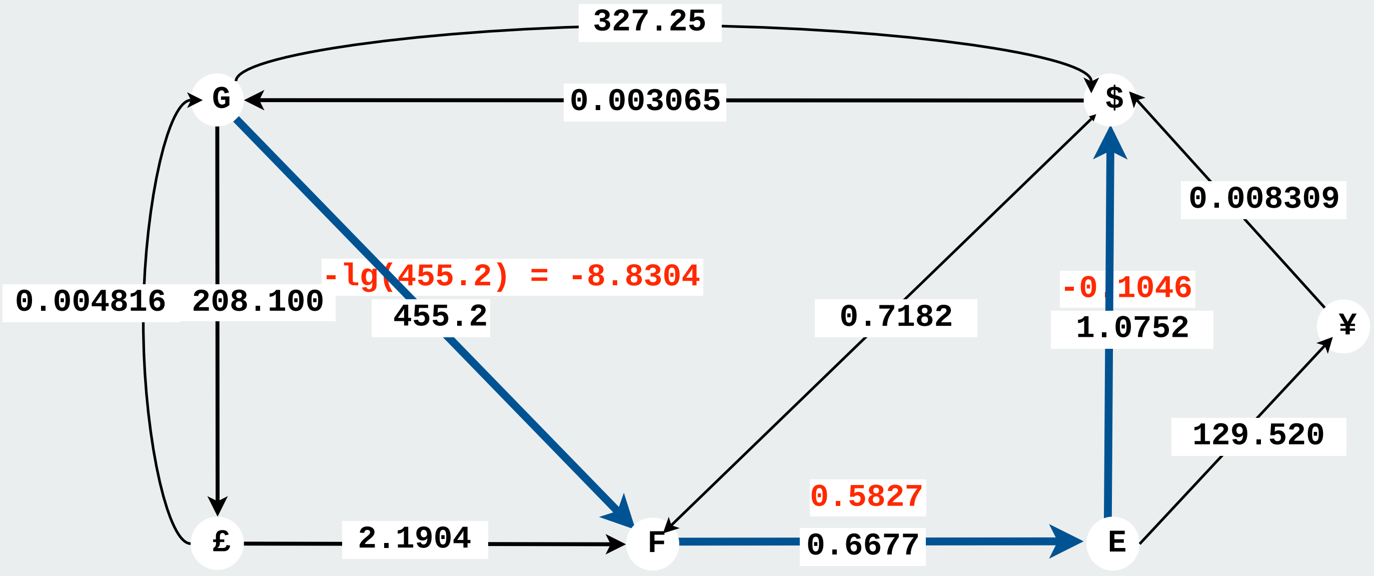
- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find path that maximizes product of weights.



Shortest paths application: Currency conversion

Reduce to shortest path problem by taking logs

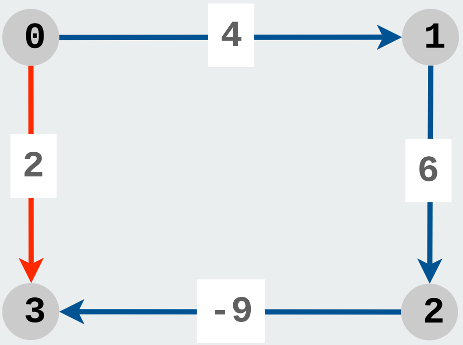
- Let weight(v-w) = − lg (exchange rate from currency v to w)
- multiplication turns to addition
- Shortest path with costs c corresponds to best exchange sequence.



Challenge. Solve shortest path problem with negative weights.

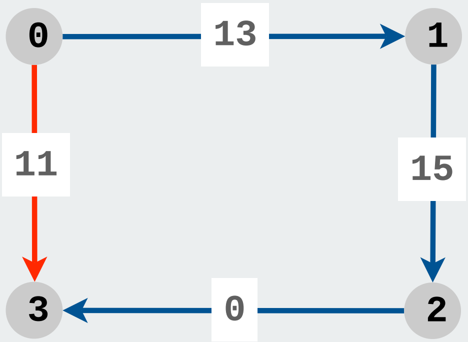
Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex **3** immediately after **0**.
But shortest path from **0** to **3** is **0**→**1**→**2**→**3**.

Re-weighting. Adding a constant to every edge weight also doesn't work.

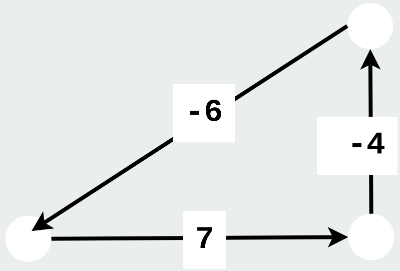


Adding 9 to each edge changes the shortest path because it adds 9 to each segment, wrong thing to do for paths with many segments.

Bad news: need a different algorithm.

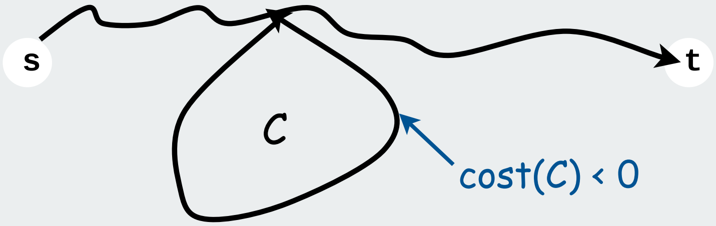
Shortest paths with negative weights: negative cycles

Negative cycle. Directed cycle whose sum of edge weights is negative.



Observations.

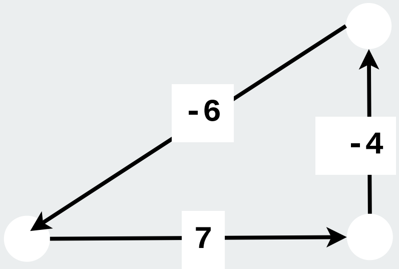
- If negative cycle C on path from s to t, then shortest path can be made arbitrarily negative by spinning around cycle
- There exists a shortest s-t path that is simple.



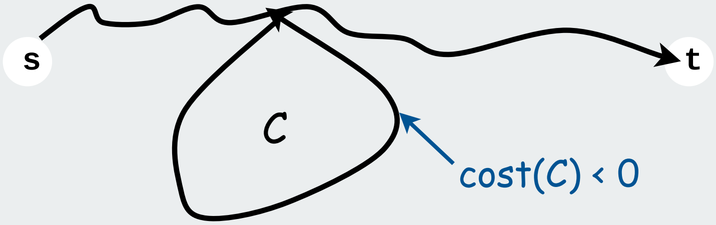
Worse news: need a different problem

Shortest paths with negative weights

Problem 1. Does a given digraph contain a negative cycle?



Problem 2. Find the shortest simple path from s to t.



Bad news: Problem 2 is intractable

Good news: Can solve problem 1 in O(VE) steps

Good news: Same algorithm solves problem 2 if no negative cycle

Bellman-Ford algorithm

- detects a negative cycle if any exist
- finds shortest simple path if no negative cycle exists

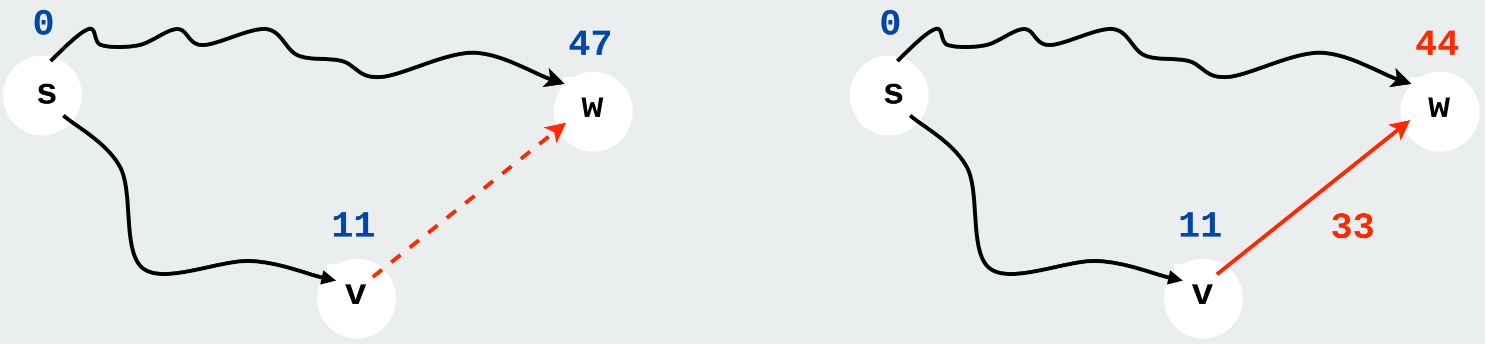
Edge relaxation

For all v, `dist[v]` is the length of some path from s to v.

Relaxation along edge e from v to w.

- `dist[v]` is length of some path from s to v
- `dist[w]` is length of some path from s to w
- if v-w gives a shorter path to w through v, update `dist[w]` and `pred[w]`

```
if (dist[w] > dist[v] + e.weight())
{
    dist[w] = dist[v] + e.weight());
    pred[w] = e;
}
```



Relaxation sets `dist[w]` to the length of a shorter path from s to w (if v-w gives one)

Shortest paths with negative weights: dynamic programming algorithm

A simple solution that works!

- Initialize `dist[v] = ∞`, `dist[s]= 0`.
- Repeat `V` times: relax each edge `e`.

```
for (int i = 1; i <= G.V(); i++)
   for (int v = 0; v < G.V(); v++)
      for (Edge e : G.adj(v))
      {
         int w = e.to();
         if (dist[w] > dist[v] + e.weight())
         {
             dist[w] = dist[v] + e.weight())
             pred[w] = e;
         }
      }
```

phase i

relax v-w

Shortest paths with negative weights: dynamic programming algorithm

Running time proportional to E V

Invariant. At end of phase **i**, **dist[v]** ≤ length of any path from s to v using at most **i** edges.

Theorem. If there are no negative cycles, upon termination **dist[v]** is the length of the shortest path from from s to v.

and **pred[]** gives the shortest paths

Shortest paths with negative weights: Bellman-Ford-Moore algorithm

Observation. If **`dist[v]`** doesn't change during phase **`i`**,
no need to relax any edge leaving **v** in phase **`i+1`**.

FIFO implementation.
Maintain queue of vertices whose distance changed.

be careful to keep at most one copy of each vertex on queue

Running time.

- still could be proportional to EV in worst case
- much faster than that in practice

Shortest paths with negative weights: Bellman-Ford-Moore algorithm

Initialize **dist[v] =** ∞ and **marked[v]= false** for all vertices **v**.

```
Queue<Integer> q = new Queue<Integer>();
marked[s] = true;
dist[s] = 0;
q.enqueue(s);


while (!q.isEmpty())
{
   int v = q.dequeue();
   marked[v] = false;
   for (Edge e : G.adj(v))
   {
      int w = e.target();
      if (dist[w] > dist[v] + e.weight())
      {
          dist[w] = dist[v] + e.weight();
          pred[w] = e;
          if (!marked[w])
          {
             marked[w] = true;
             q.enqueue(w);
          }
      }
   }
}
```

Single Source Shortest Paths Implementation: Cost Summary

	algorithm	worst case	typical case
nonnegative costs	Dijkstra (classic)	V^2	V^2
	Dijkstra (heap)	E lg E	Ⓔ
no negative cycles	Dynamic programming	EV	EV
	Bellman-Ford-Moore	EV	Ⓔ

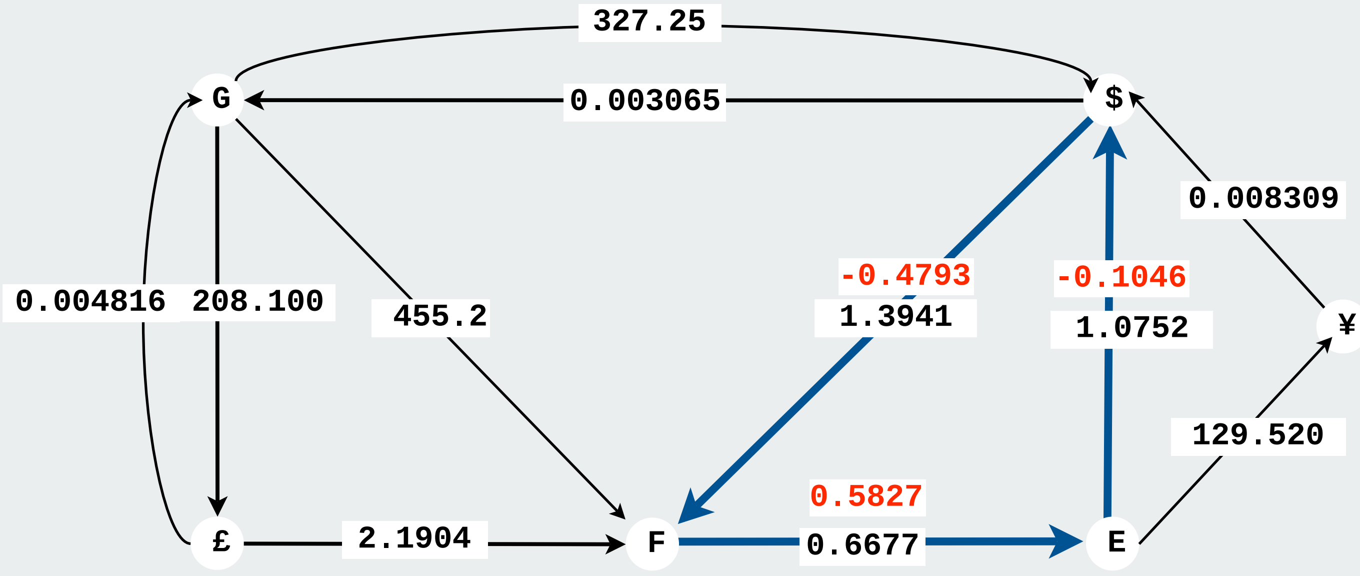
Remark 1. Negative weights makes the problem harder.

Remark 2. Negative cycles makes the problem intractable.

Shortest paths application: arbitrage

Is there an arbitrage opportunity in currency graph?

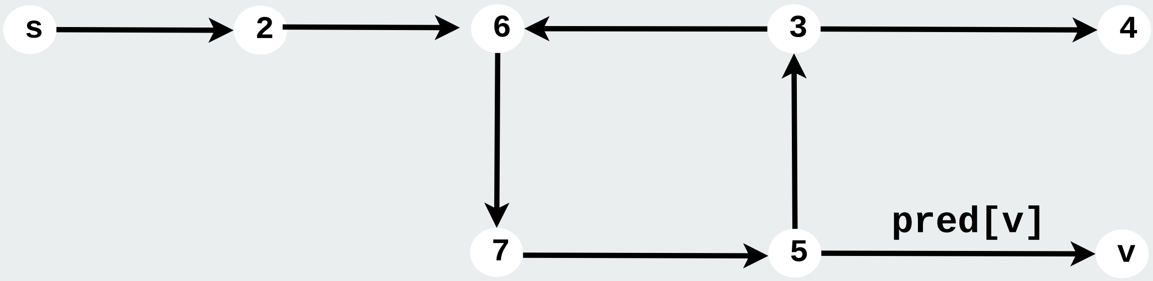
- Ex: $1 ⇒ 1.3941 Francs ⇒ 0.9308 Euros ⇒ $1.00084.
- Is there a negative cost cycle?
- Fastest algorithm is valuable!



```
-0.4793 + 0.5827 - 0.1046 < 0
```

Negative cycle detection

If there is a negative cycle reachable from s.
Bellman-Ford-Moore gets stuck in loop, updating vertices in cycle.

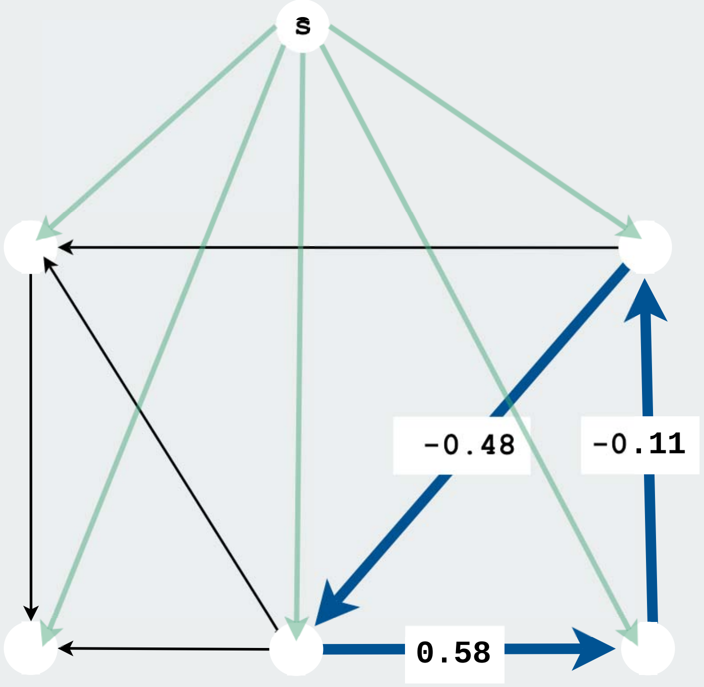


Finding a negative cycle. If any vertex v is updated in phase V,
there exists a negative cycle, and we can trace back **pred[v]** to find it.

Negative cycle detection

Goal. Identify a negative cycle (reachable from any vertex).

Solution. Add 0-weight edge from artificial source s to each vertex v. Run Bellman-Ford from vertex s.



Shortest paths summary

Dijkstra's algorithm

- easy and optimal for dense digraphs
- PQ/ST data type gives near optimal for sparse graphs

Priority-first search

- generalization of Dijkstra's algorithm
- encompasses DFS, BFS, and Prim
- enables easy solution to many graph-processing problems

Negative weights

- arise in applications
- make problem intractable in presence of negative cycles (!)
- easy solution using old algorithms otherwise

Shortest-paths is a broadly useful problem-solving model

Geometric Algorithms

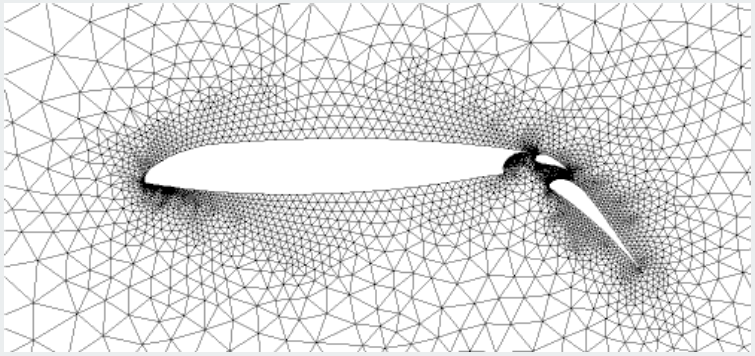
- primitive operations
- convex hull
- closest pair
- voronoi diagram

References:
Algorithms in C (2nd edition), Chapters 24-25
http://www.cs.princeton.edu/introalgsds/71primitives
http://www.cs.princeton.edu/introalgsds/72hull

Geometric Algorithms

Applications.

- Data mining.
- VLSI design.
- Computer vision.
- Mathematical models.
- Astronomical simulation.
- Geographic information systems.
- Computer graphics (movies, games, virtual reality).
- Models of physical world (maps, architecture, medical imaging).



airflow around an aircraft wing

Reference: http://www.ics.uci.edu/~eppstein/geom.html

History.

- Ancient mathematical foundations.
- Most geometric algorithms less than 25 years old.

- primitive operations
- convex hull
- closest pair
- voronoi diagram

Geometric Primitives

Point: two numbers (x, y).

Line: two numbers a and b [ax + by = 1] ← any line not through origin

Line segment: two points.

Polygon: sequence of points.

Primitive operations.

- Is a point inside a polygon?
- Compare slopes of two lines.
- Distance between two points.
- Do two line segments intersect?
- Given three points p_1, p_2, p_3, is p_1-p_2-p_3 a counterclockwise turn?

Other geometric shapes.

- Triangle, rectangle, circle, sphere, cone, …
- 3D and higher dimensions sometimes more complicated.

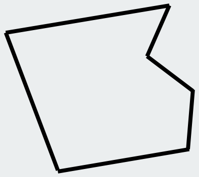
Intuition

Warning: intuition may be misleading.

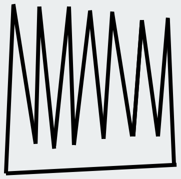
- Humans have spatial intuition in 2D and 3D.
- Computers do not.
- Neither has good intuition in higher dimensions!

Is a given polygon simple?

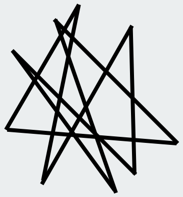
no crossings



1	6	5	8	7	2
7	8	6	4	2	1



1	15	14	13	12	11	10	9	8	7	6	5	4	3	2
1	2	18	4	18	4	19	4	19	4	20	3	20	3	20



1	10	3	7	2	8	8	3	4
6	5	15	1	11	3	14	2	16

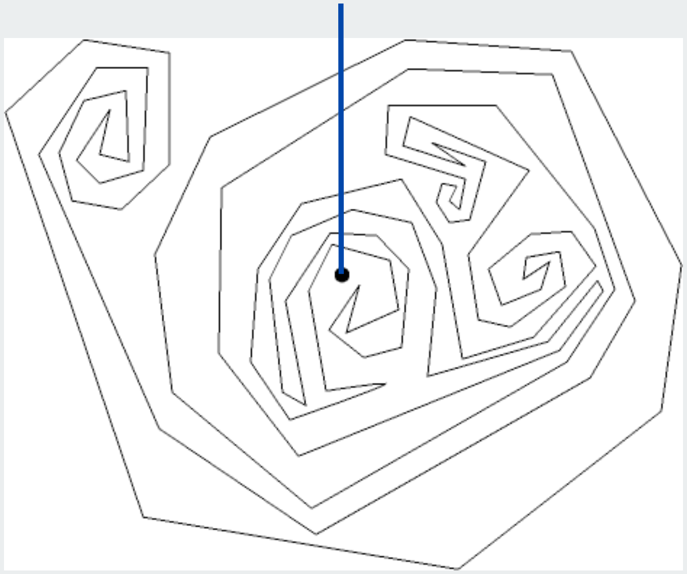
we think of this

algorithm sees this

Polygon Inside, Outside

Jordan curve theorem. [Veblen 1905] Any continuous simple closed curve cuts the plane in exactly two pieces: the inside and the outside.

Is a point inside a simple polygon?



http://www.ics.uci.edu/~eppstein/geom.html

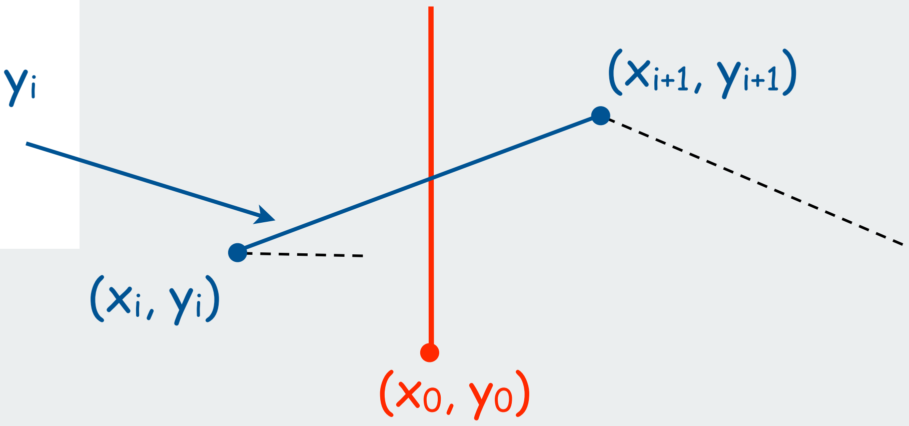
Application. Draw a filled polygon on the screen.

Polygon Inside, Outside: Crossing Number

Does line segment intersect ray?

$$y_0 = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x_0 - x_i) + y_i$$

$$x_i \le x_0 \le x_{i+1}$$

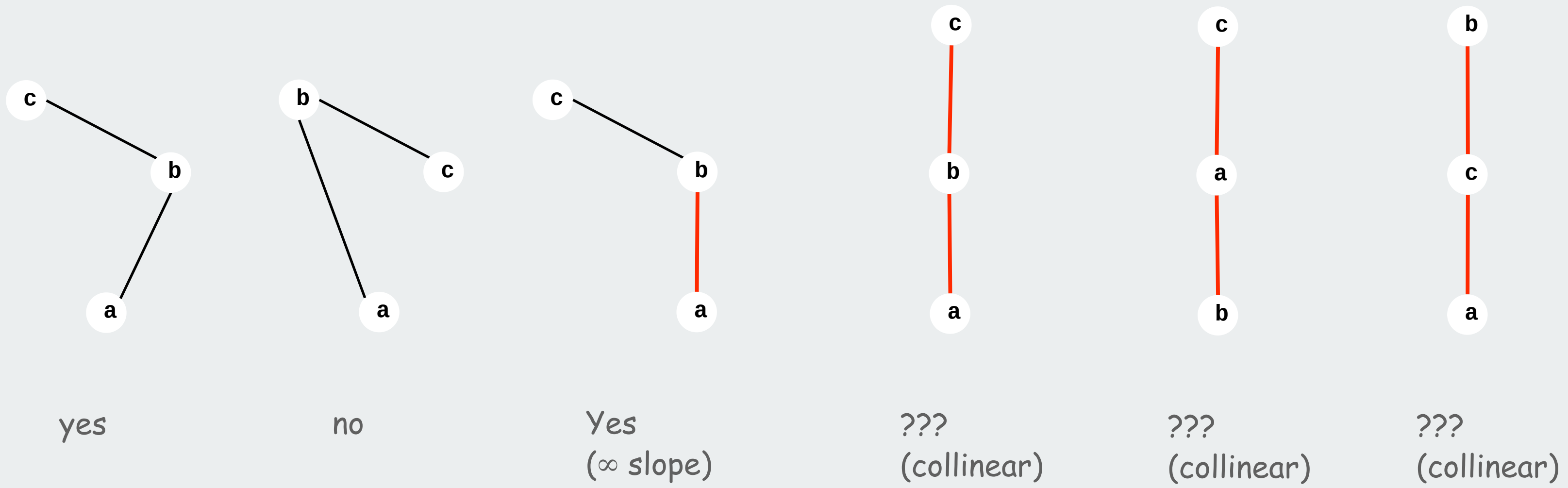


```
public boolean contains(double x0, double y0)
{
    int crossings = 0;
    for (int i = 0; i < N; i++)
    {
       double  slope = (y[i+1] - y[i]) / (x[i+1] - x[i]);
       boolean cond1 = (x[i] <= x0) && (x0 < x[i+1]);
       boolean cond2 = (x[i+1] <= x0) && (x0 < x[i]);
       boolean above = (y0 < slope * (x0 - x[i]) + y[i]);
       if ((cond1 || cond2)  && above ) crossings++;
    }
    return ( crossings % 2 != 0 );
 }
```

Implementing CCW

CCW. Given three point a, b, and c, is a-b-c a counterclockwise turn?

- Analog of comparisons in sorting.
- Idea: compare slopes.



Lesson. Geometric primitives are tricky to implement.

- Dealing with degenerate cases.
- Coping with floating point precision.

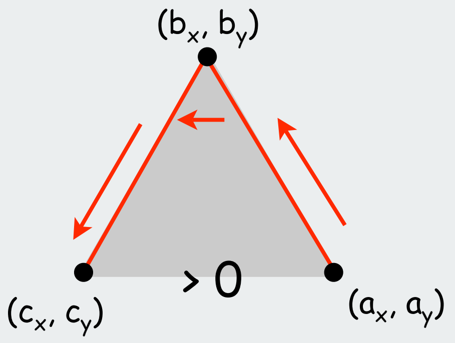
Implementing CCW

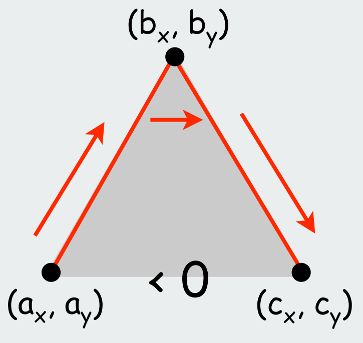
CCW. Given three point a, b, and c, is a-b-c a counterclockwise turn?

- Determinant gives twice area of triangle.

$$2 \times Area(a, b, c) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

- If area > 0 then a-b-c is counterclockwise.
- If area < 0, then a-b-c is clockwise.
- If area = 0, then a-b-c are collinear.





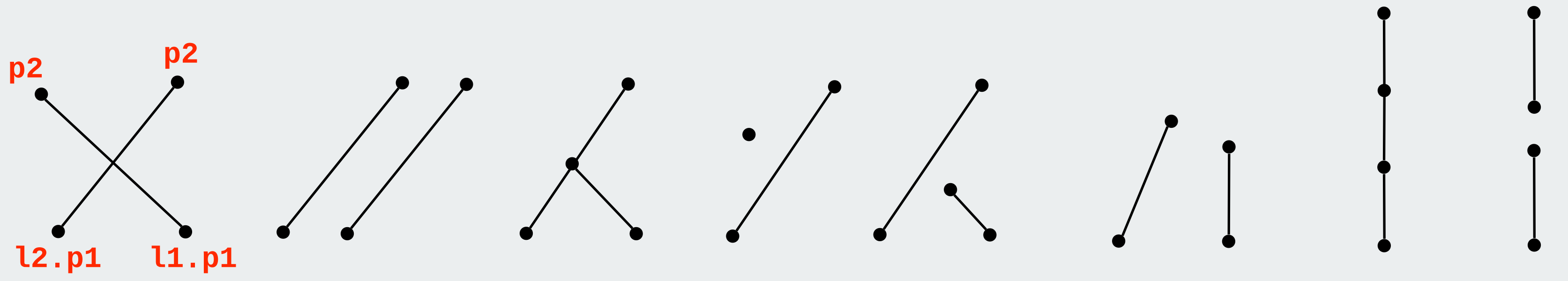
Immutable Point ADT

```
public final class Point
{
   public final int x;
   public final int y;

   public Point(int x, int y)
   {  this.x = x; this.y = y;  }


   public double distanceTo(Point q)
   {  return Math.hypot(this.x - q.x, this.y - q.y);  }

   public static int ccw(Point a, Point b, Point c)
   {
      double area2 = (b.x-a.x)*(c.y-a.y) - (b.y-a.y)*(c.x-a.x);
      if else (area2 < 0) return -1;
      else if (area2 > 0) return +1;
      else if (area2 > 0  return  0;
   }

   public static boolean collinear(Point a, Point b, Point c)
   {
      return ccw(a, b, c) == 0;
   }
}
```

Sample ccw client: Line intersection

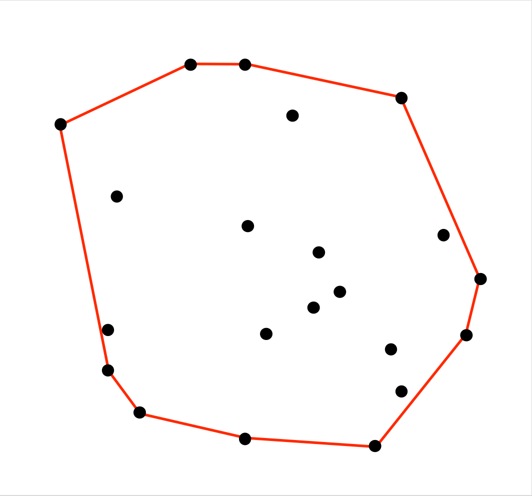
Intersect: Given two line segments, do they intersect?

- Idea 1: find intersection point using algebra and check.
- Idea 2: check if the endpoints of one line segment are on different "sides" of the other line segment.
- 4 ccw computations.



not handled

```
public static boolean intersect(Line l1, Line l2)
{
   int test1, test2;
   test1 = Point.ccw(l1.p1, l1.p2, l2.p1)
           * Point.ccw(l1.p1, l1.p2, l2.p2);
   test2 = Point.ccw(l2.p1, l2.p2, l1.p1)
           * Point.ccw(l2.p1, l2.p2, l1.p2);
   return (test1 <= 0) && (test2 <= 0);
}
```

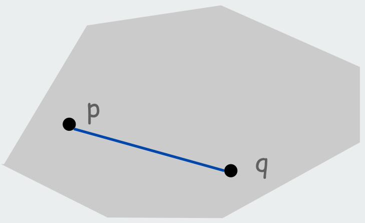


- primitive operations
- convex hull
- closest pair
- voronoi diagram

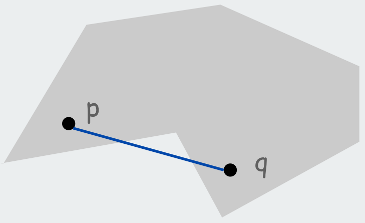
Convex Hull

A set of points is convex if for any two points p and q in the set, the line segment pq is completely in the set.

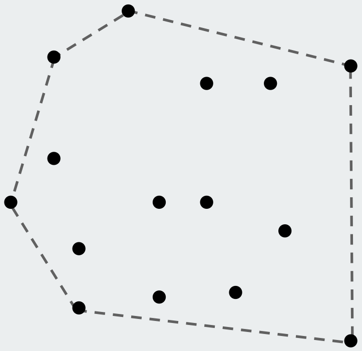
Convex hull. Smallest convex set containing all the points.



convex



not convex



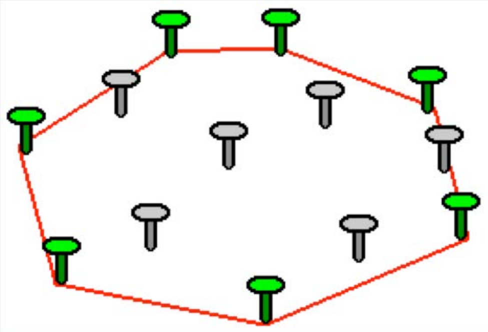
convex hull

Properties.

- "Simplest" shape that approximates set of points.
- Shortest (perimeter) fence surrounding the points.
- Smallest (area) convex polygon enclosing the points.

Mechanical Solution

Mechanical algorithm. Hammer nails perpendicular to plane; stretch elastic rubber band around points.



http://www.dfanning.com/math_tips/convexhull_1.gif

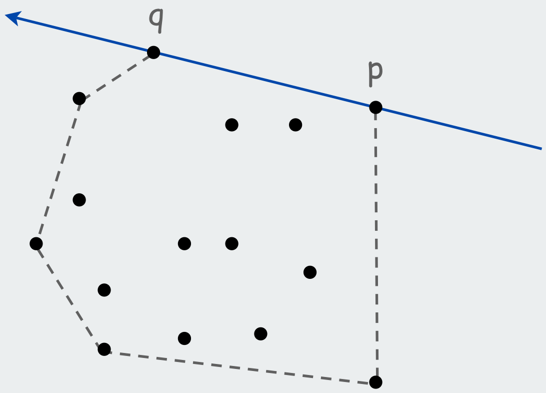
Brute-force algorithm

Observation 1.

Edges of convex hull of P connect pairs of points in P.

Observation 2.

p-q is on convex hull if all other points are counterclockwise of $\overrightarrow{pq}$.



O(N^3) algorithm.

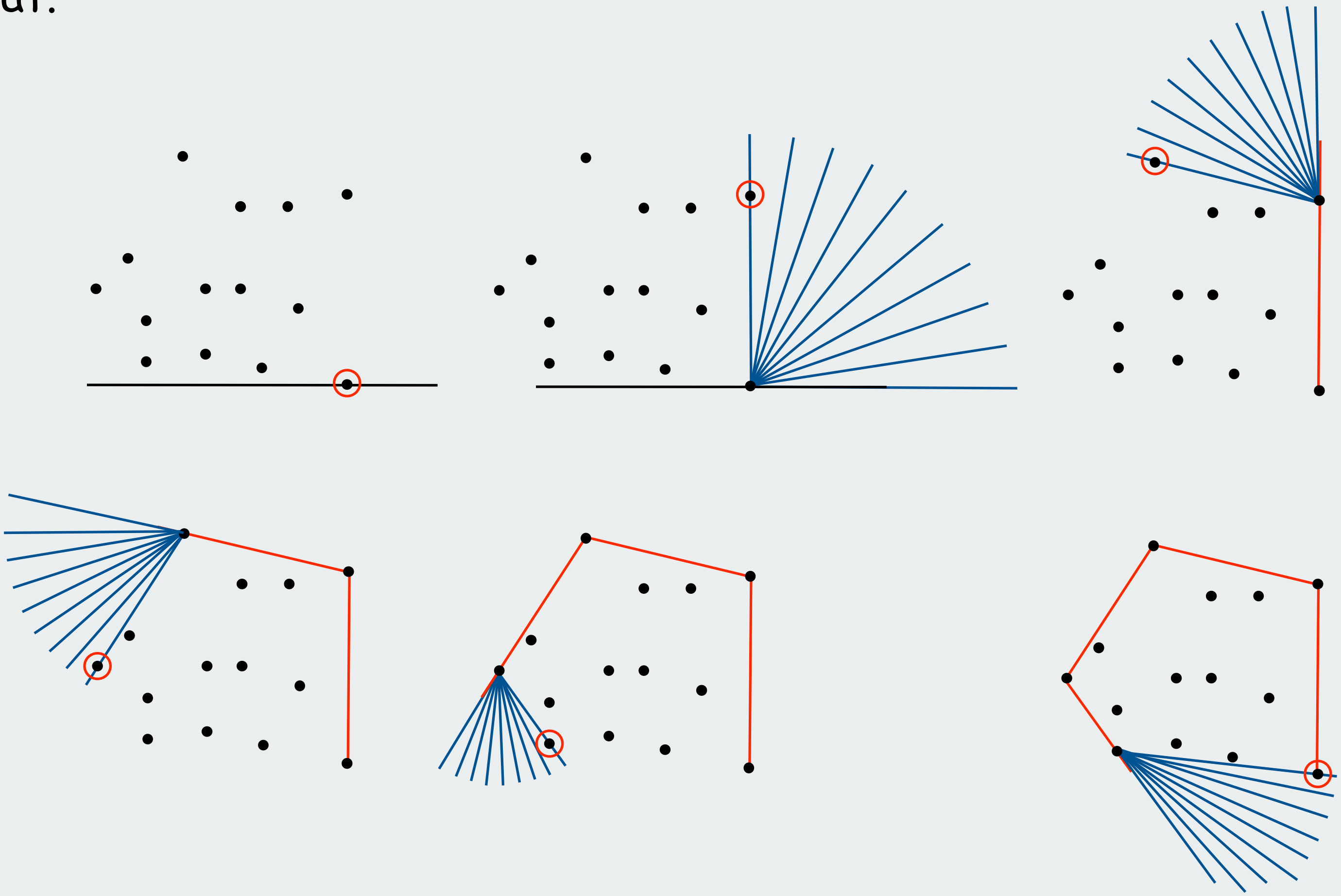
For all pairs of points p and q in P

- compute ccw(p, q, x) for all other x in P
- p-q is on hull if all values positive

Package Wrap (Jarvis March)

Package wrap.

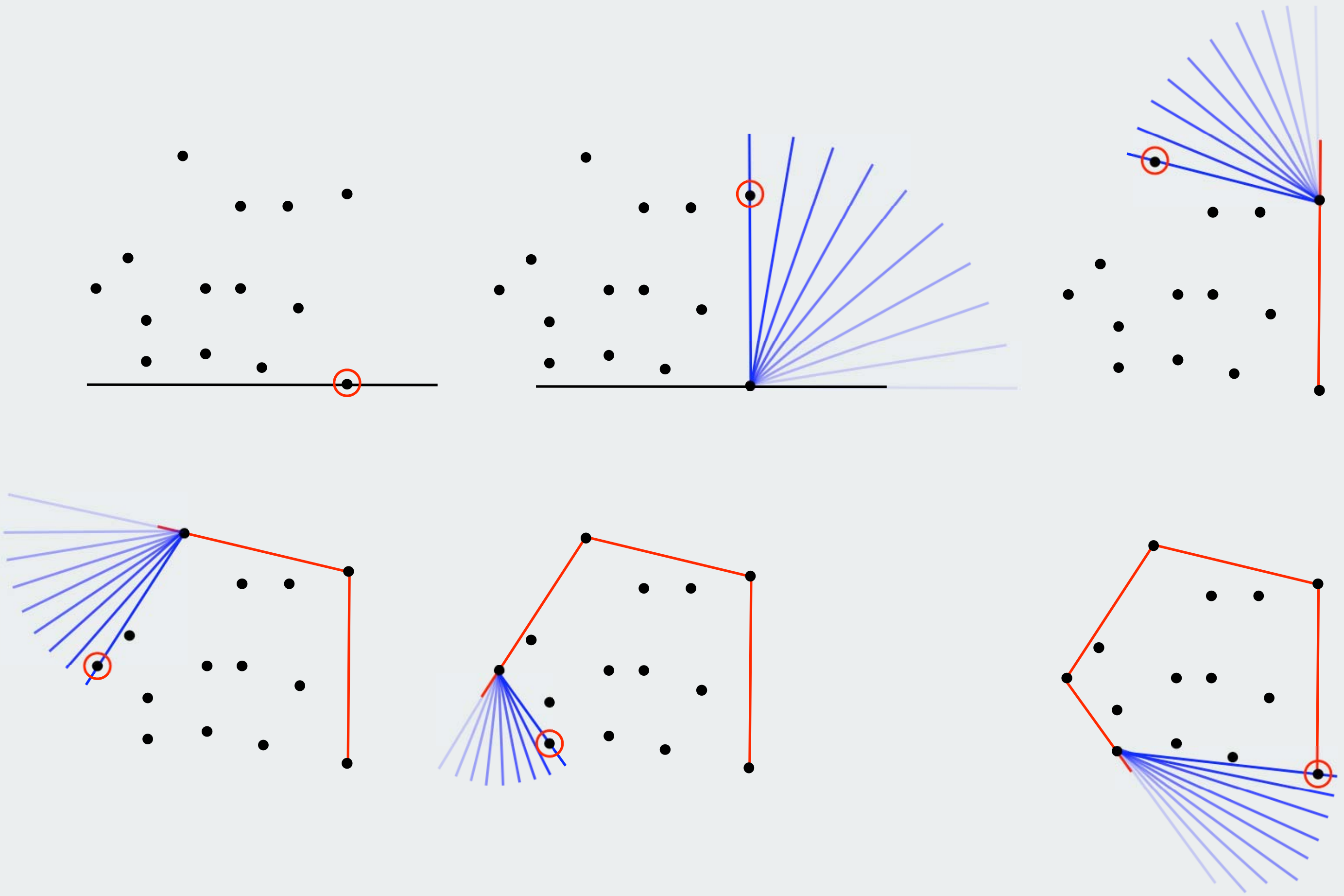
- Start with point with smallest y-coordinate.
- Rotate sweep line around current point in ccw direction.
- First point hit is on the hull.
- Repeat.



Package Wrap (Jarvis March)

Implementation.

- Compute angle between current point and all remaining points.
- Pick smallest angle larger than current angle.
- $\Theta(N)$ per iteration.



How Many Points on the Hull?

Parameters.

- N = number of points.
- h = number of points on the hull.

Package wrap running time. $\Theta(N\,h)$ per iteration.

How many points on hull?

- Worst case: h = N.
- Average case: difficult problems in stochastic geometry.

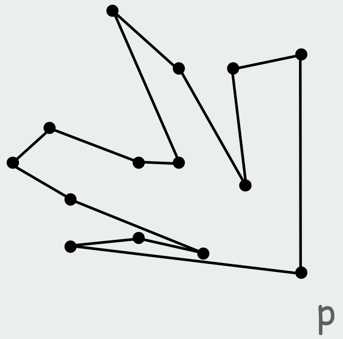
  in a disc: $h = N^{1/3}$.

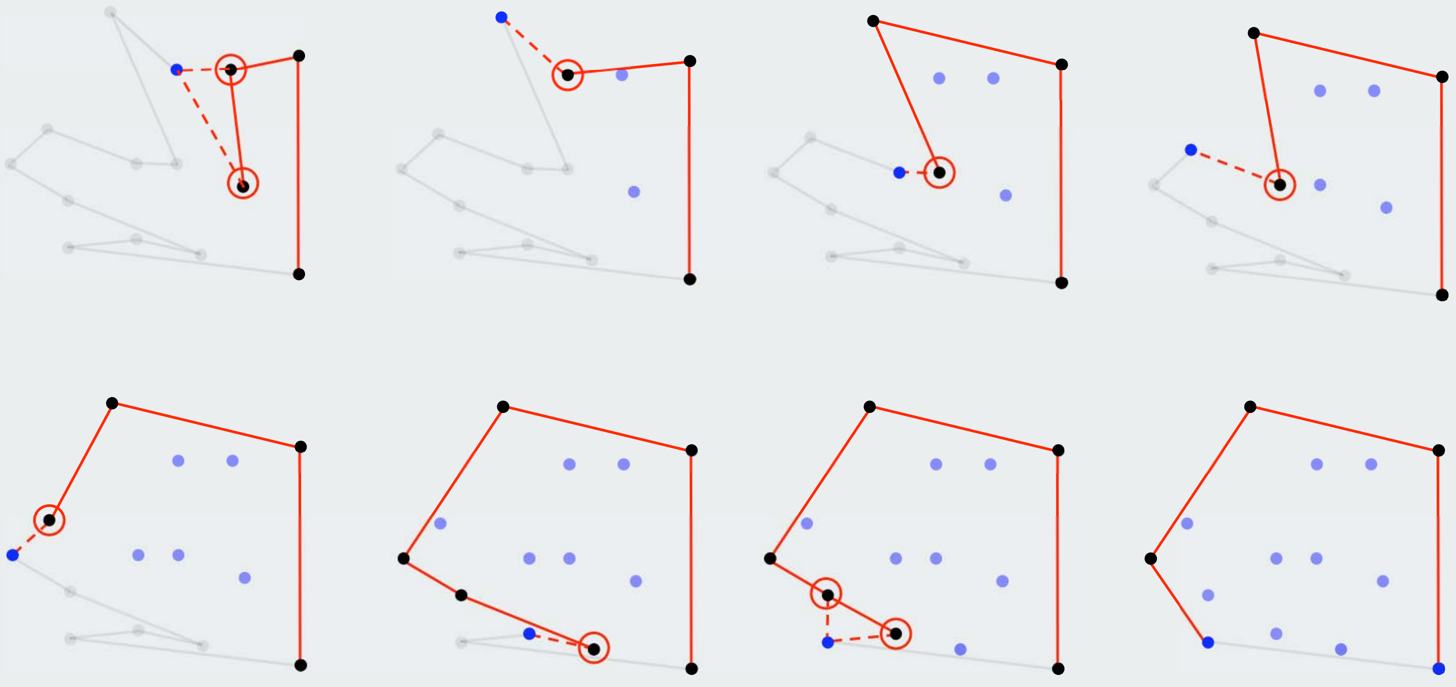
  in a convex polygon with O(1) edges: h = log N.

Graham Scan: Example

Graham scan.

- Choose point p with smallest y-coordinate.
- Sort points by polar angle with p to get simple polygon.
- Consider points in order, and discard those that would create a clockwise turn.





Graham Scan: Example

Implementation.

- Input: `p[1], p[2], …, p[N]` are points.
- Output: `M` and rearrangement so that `p[1],...,p[M]` is convex hull.

```
// preprocess so that p[1] has smallest y-coordinate
// sort by angle with p[1]

points[0] = points[N];  // sentinel
int M = 2;
for (int i = 3; i <= N; i++)
{
   while (Point.ccw(p[M-1], p[M], p[i]) <= 0) M--;
   M++;
   swap(points, M, i);
}
```

discard points that would create clockwise turn

add i to putative hull

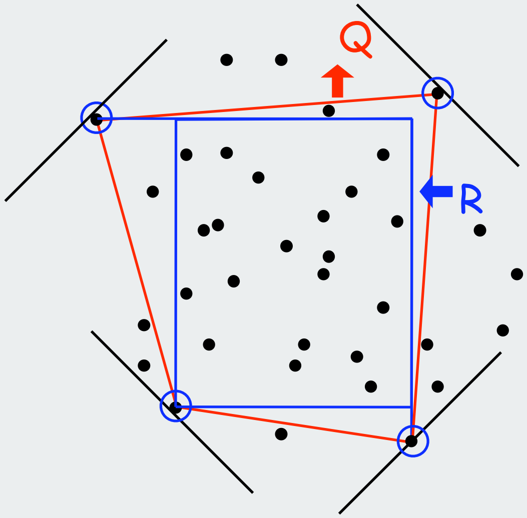
Running time. O(N log N) for sort and O(N) for rest.

why?

Quick Elimination

Quick elimination.

- Choose a quadrilateral Q or rectangle R with 4 points as corners.
- Any point inside cannot be on hull
  - 4 ccw tests for quadrilateral
  - 4 comparisons for rectangle

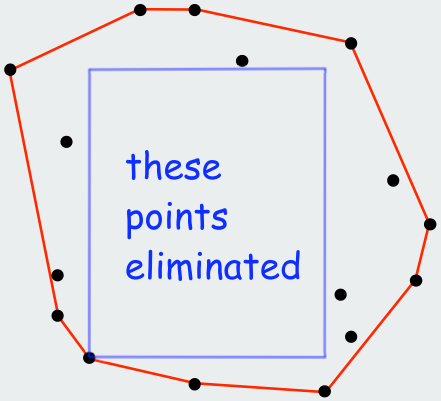


Three-phase algorithm

- Pass through all points to compute R.
- Eliminate points inside R.
- Find convex hull of remaining points.

In practice

can eliminate almost all points in linear time.



Convex Hull Algorithms Costs Summary

Asymptotic cost to find h-point hull in N-point set

algorithm	growth of running time
Package wrap	N h
Graham scan	N log N
Quickhull	N log N
Mergehull	N log N
Sweep line	N log N
Quick elimination	N †
Best in theory	N log h

output sensitive

† assumes "reasonable" point distribution

Convex Hull: Lower Bound

Models of computation.

- Comparison based: compare coordinates.
  (impossible to compute convex hull in this model of computation)

```
(a.x < b.x) || ((a.x == b.x) && (a.y < b.y)))
```

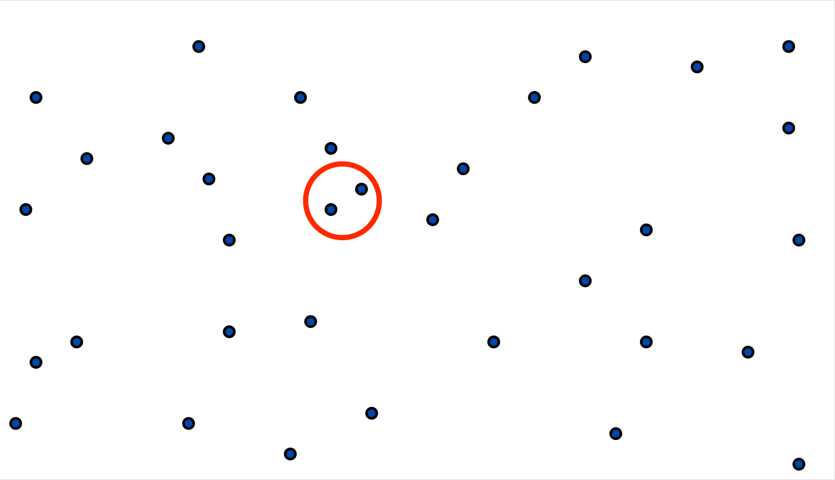
- Quadratic decision tree model: compute any quadratic function of the coordinates and compare against 0.

```
(a.x*b.y - a.y*b.x + a.y*c.x - a.x*c.y + b.x*c.y - c.x*b.y) < 0
```

Theorem. [Andy Yao, 1981] In quadratic decision tree model, any convex hull algorithm requires $\Omega(N \log N)$ ops.

even if hull points are not required to be output in counterclockwise order

higher degree polynomial tests don't help either [Ben-Or, 1983]



- primitive operations
- convex hull
- **closest pair**
- voronoi diagram

Closest pair problem

Given: N points in the plane

Goal: Find a pair with smallest Euclidean distance between them.

Fundamental geometric primitive.

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbor, Euclidean MST, Voronoi.

↖ fast closest pair inspired fast algorithms for these problems

Brute force.

Check all pairs of points p and q with $\Theta(N^2)$ distance calculations.

1-D version. O(N log N) easy if points are on a line.

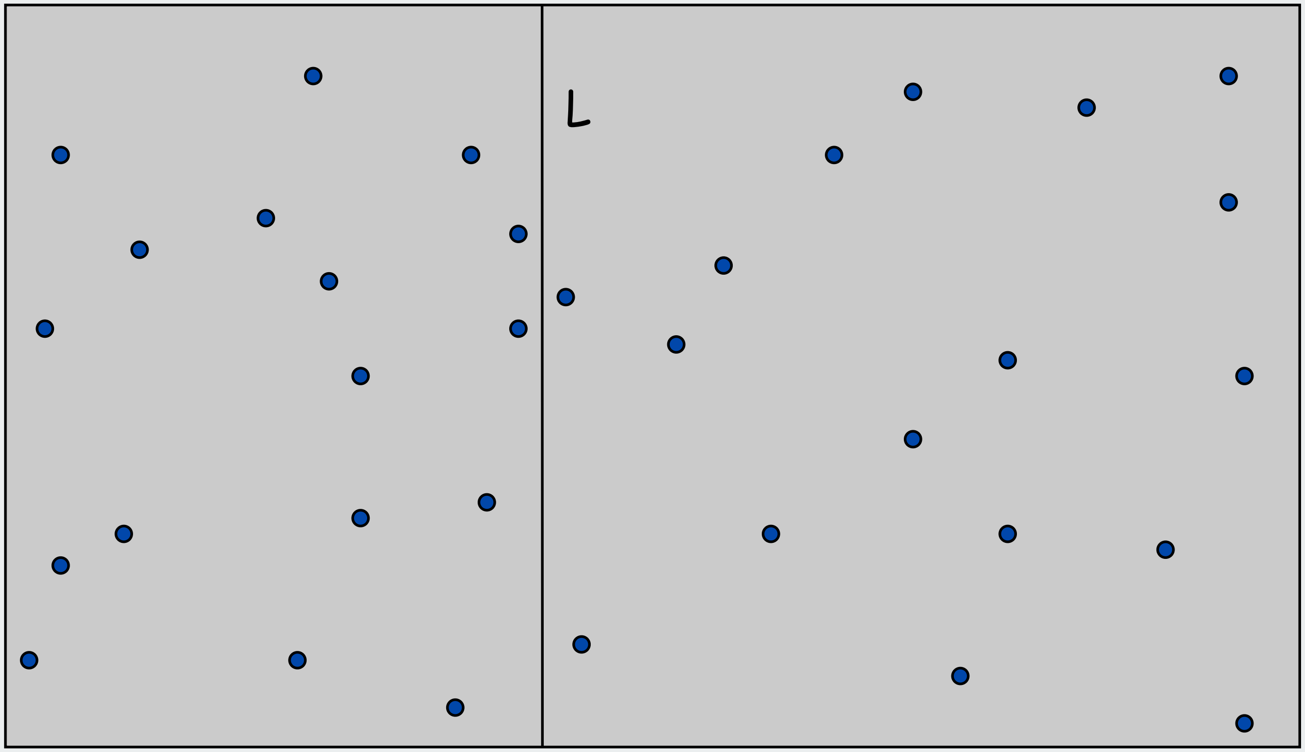
Degeneracies complicate solutions. ← as usual for geometric algs

[assumption for lecture: no two points have same x coordinate]

Closest Pair of Points

Algorithm.

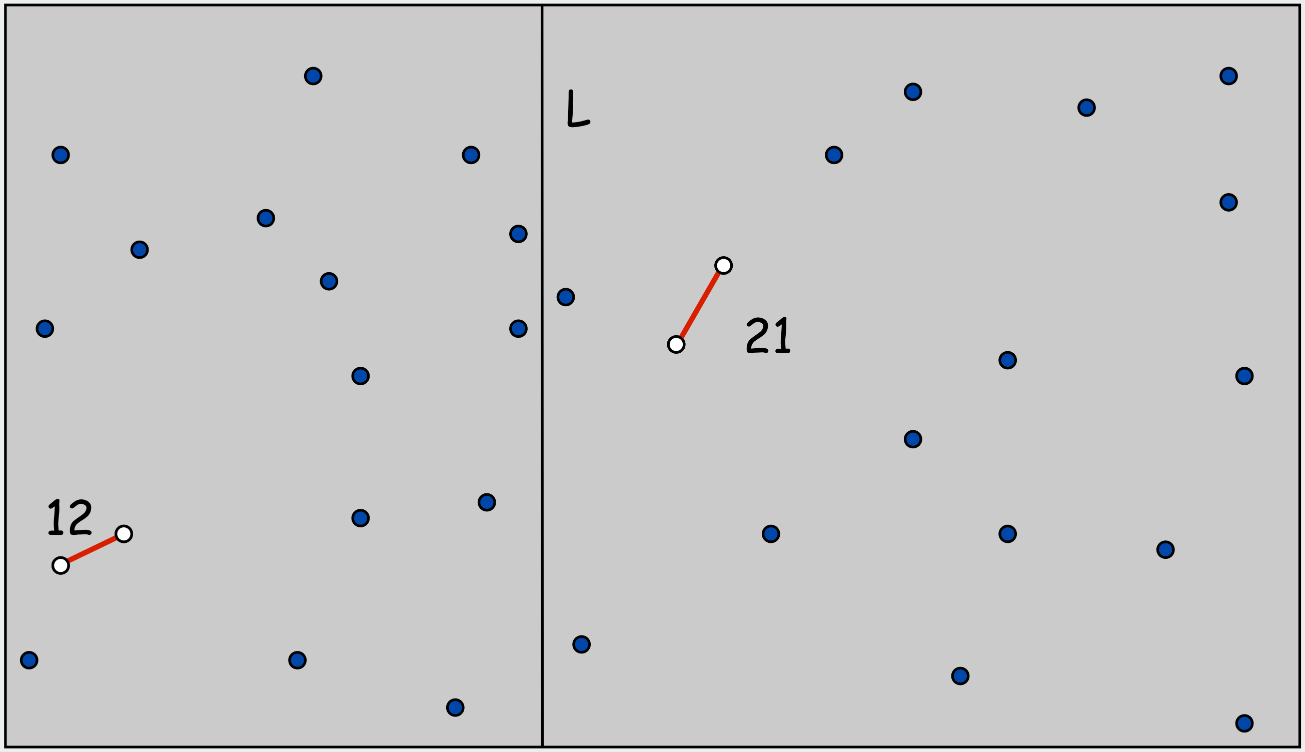
- Divide: draw vertical line L so that roughly $\frac{1}{2}$N points on each side.



Closest Pair of Points

Algorithm.

- Divide: draw vertical line L so that roughly $\frac{1}{2}$N points on each side.
- Conquer: find closest pair in each side recursively.

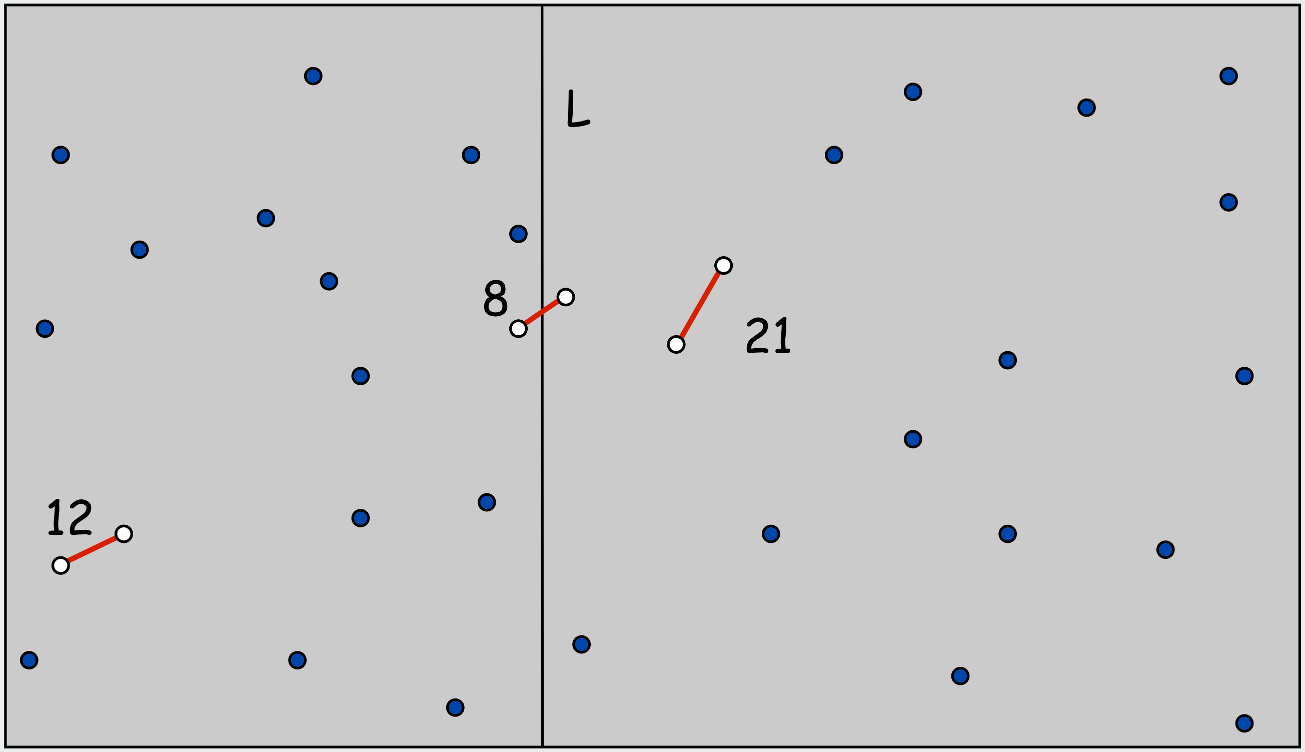


Closest Pair of Points

Algorithm.

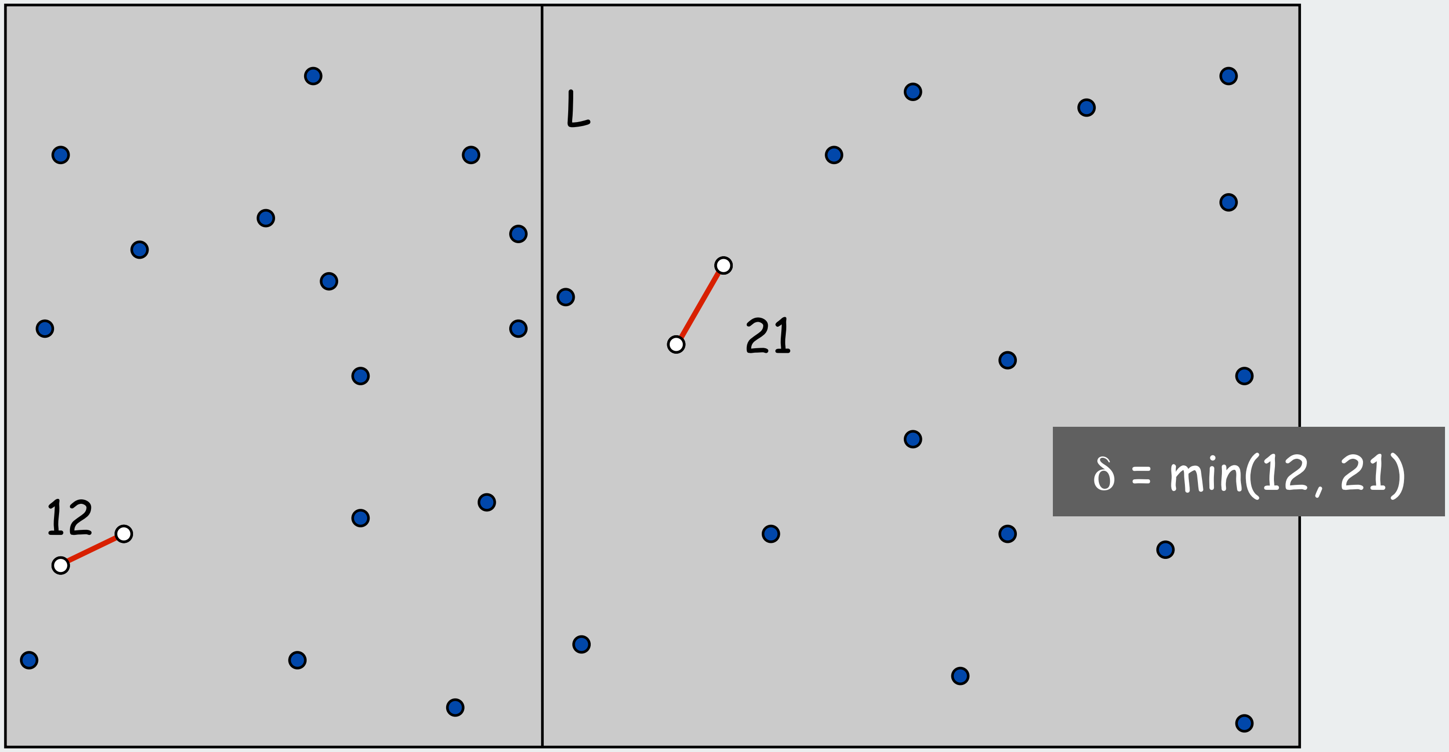
- Divide: draw vertical line L so that roughly $\frac{1}{2}N$ points on each side.
- Conquer: find closest pair in each side recursively.
- Combine: find closest pair with one point in each side.
- Return best of 3 solutions.

↖ seems like $\Theta(N^2)$



Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.

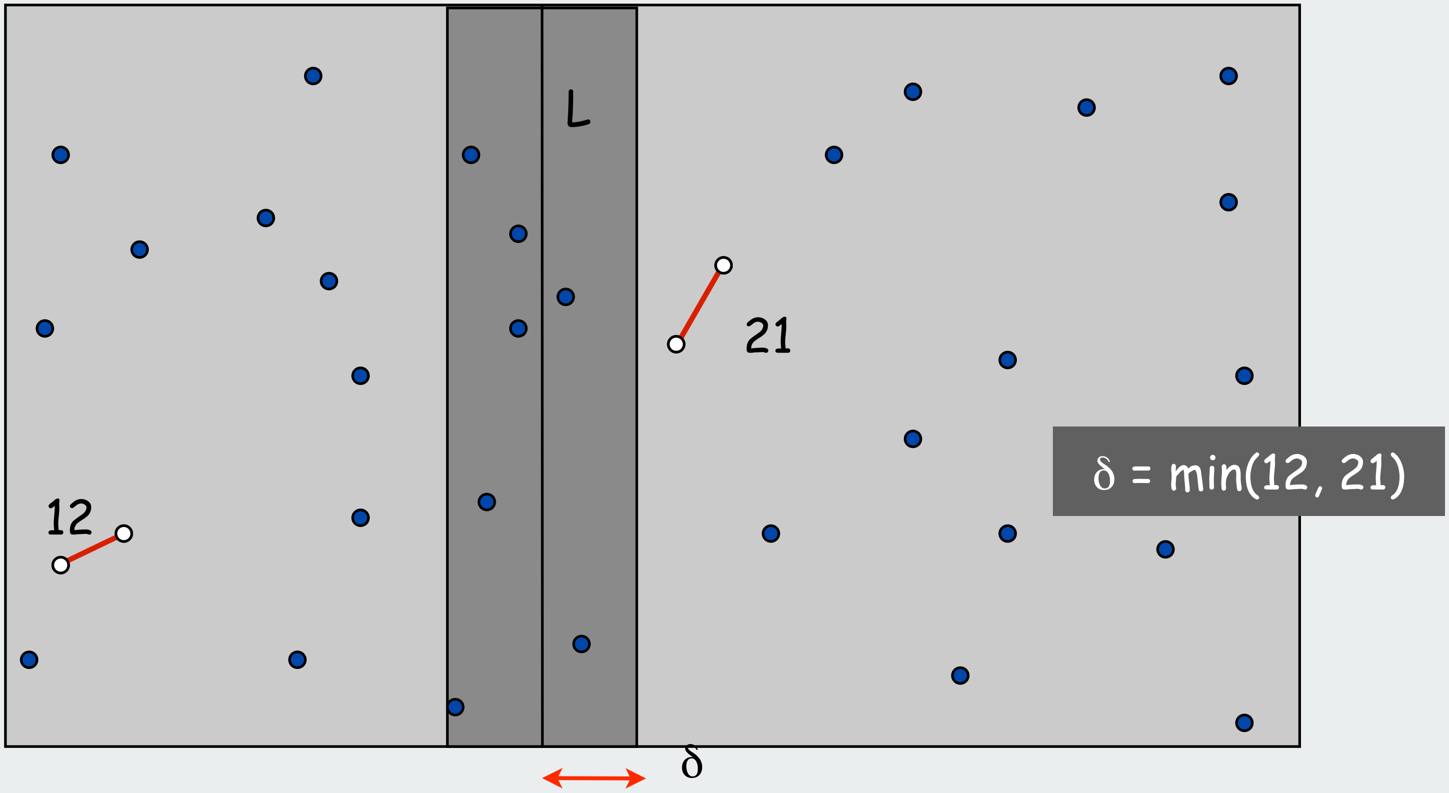


δ = min(12, 21)

Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.

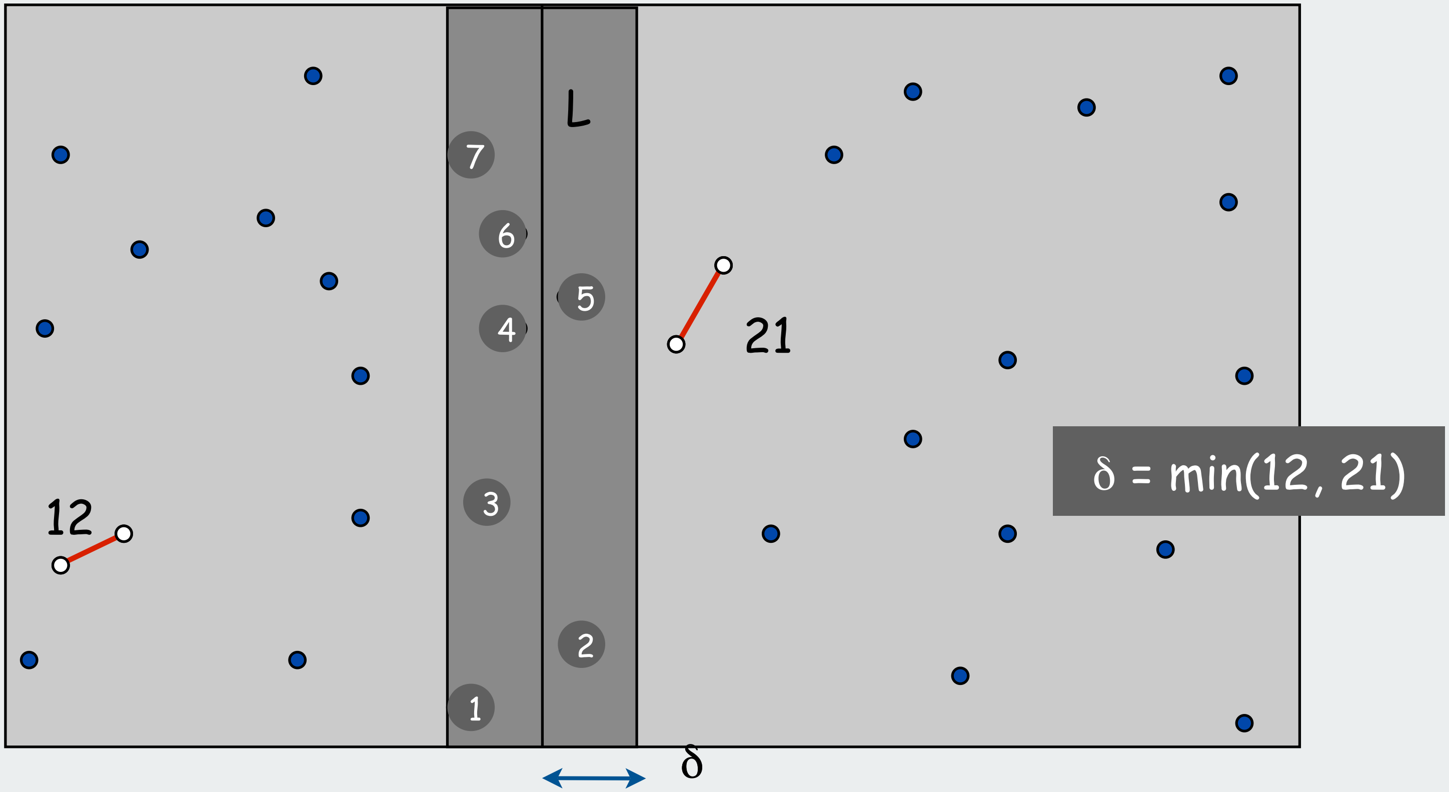
- Observation: only need to consider points within δ of line L.



Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.

- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.

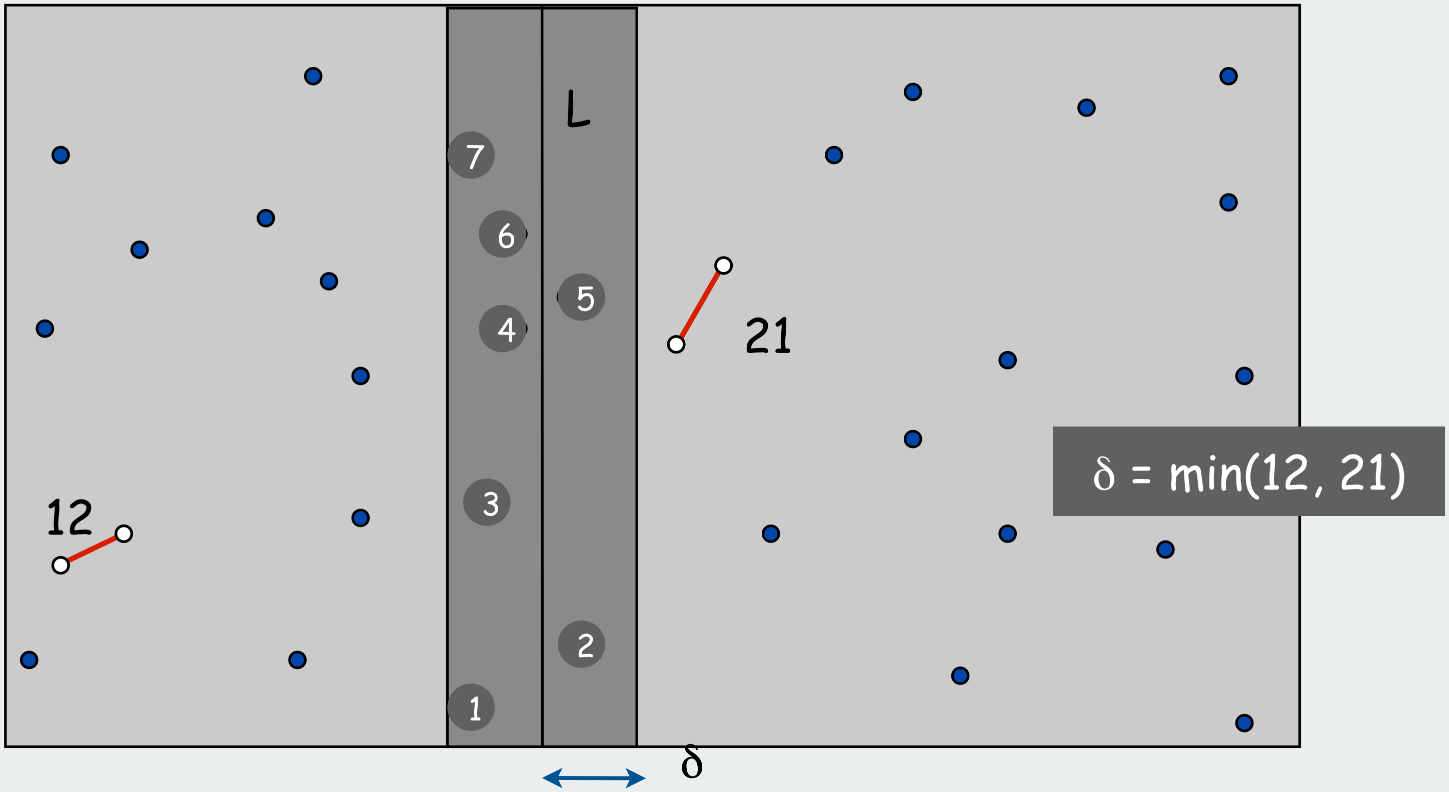


δ = min(12, 21)

Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.

- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.
- Only check distances of those within 11 positions in sorted list!



δ = min(12, 21)

Closest Pair of Points

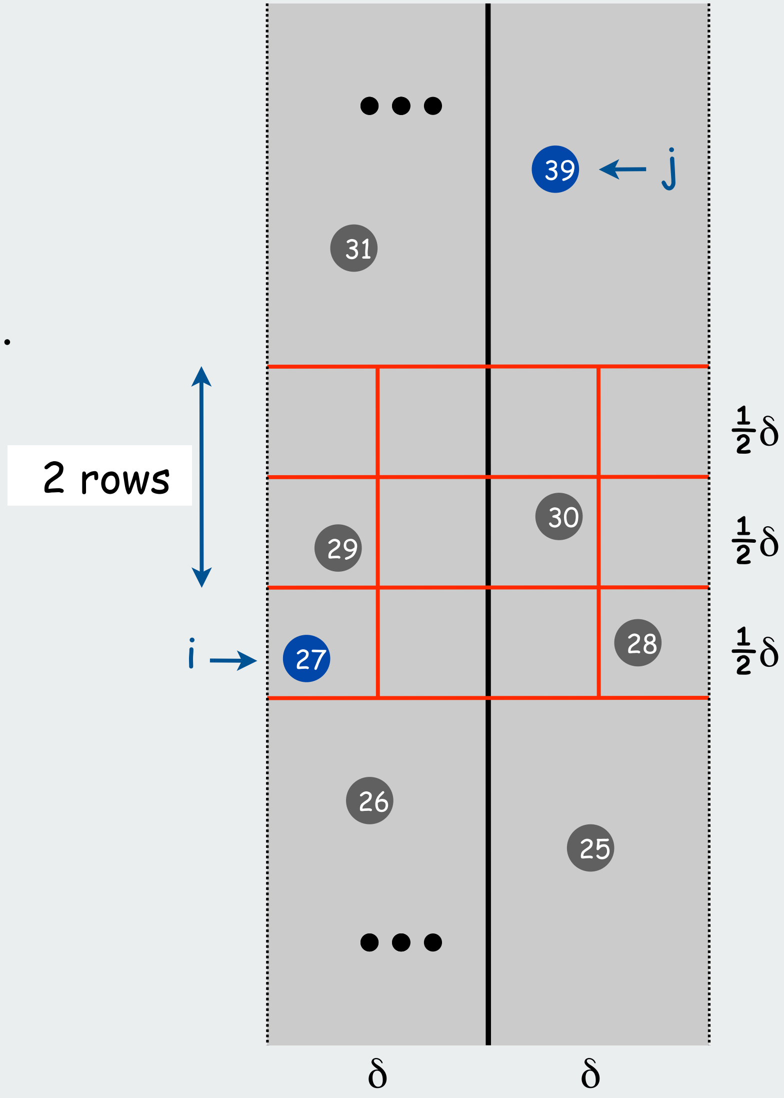
Def. Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.

Claim. If $|i - j| \geq 12$, then the distance between s_i and s_j is at least δ.

Pf.

- No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box.
- Two points at least 2 rows apart have distance $\geq 2(\frac{1}{2}\delta)$. ▪

Fact. Still true if we replace 12 with 7.



Closest Pair Algorithm

```
Closest-Pair($p_1$, …, $p_n$)
{
   Compute separation line L such that half the points
   are on one side and half on the other side.                      O(N log N)


   $\delta_1$ = Closest-Pair(left half)
   $\delta_2$ = Closest-Pair(right half)                            2T(N / 2)
   $\delta$  = min($\delta_1$, $\delta_2$)

   Delete all points further than $\delta$ from separation line L   O(N)


   Sort remaining points by y-coordinate.                           O(N log N)


   Scan points in y-order and compare distance between
   each point and next 11 neighbors. If any of these                O(N)
   distances is less than $\delta$, update $\delta$.

   return $\delta$.
}
```

Closest Pair of Points: Analysis

Algorithm gives upper bound on running time

Recurrence

$$T(N) \le 2T(N/2) + O(N \log N)$$

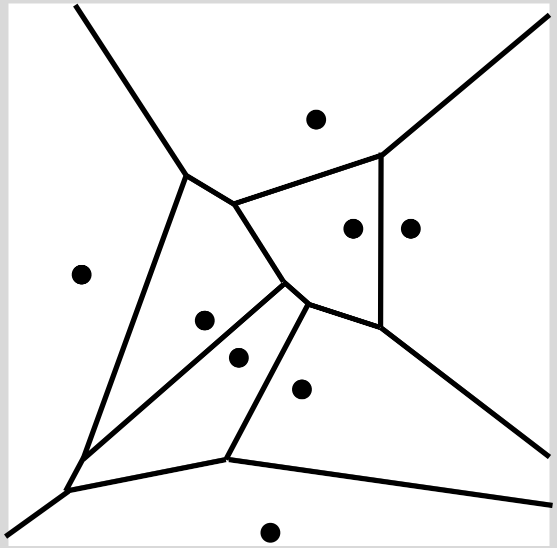
Solution

$$T(N) = O(N (\log N)^2)$$

avoid sorting by y-coordinate from scratch

Upper bound. Can be improved to O(N log N).

Lower bound. In quadratic decision tree model, any algorithm for closest pair requires $\Omega(N \log N)$ steps.

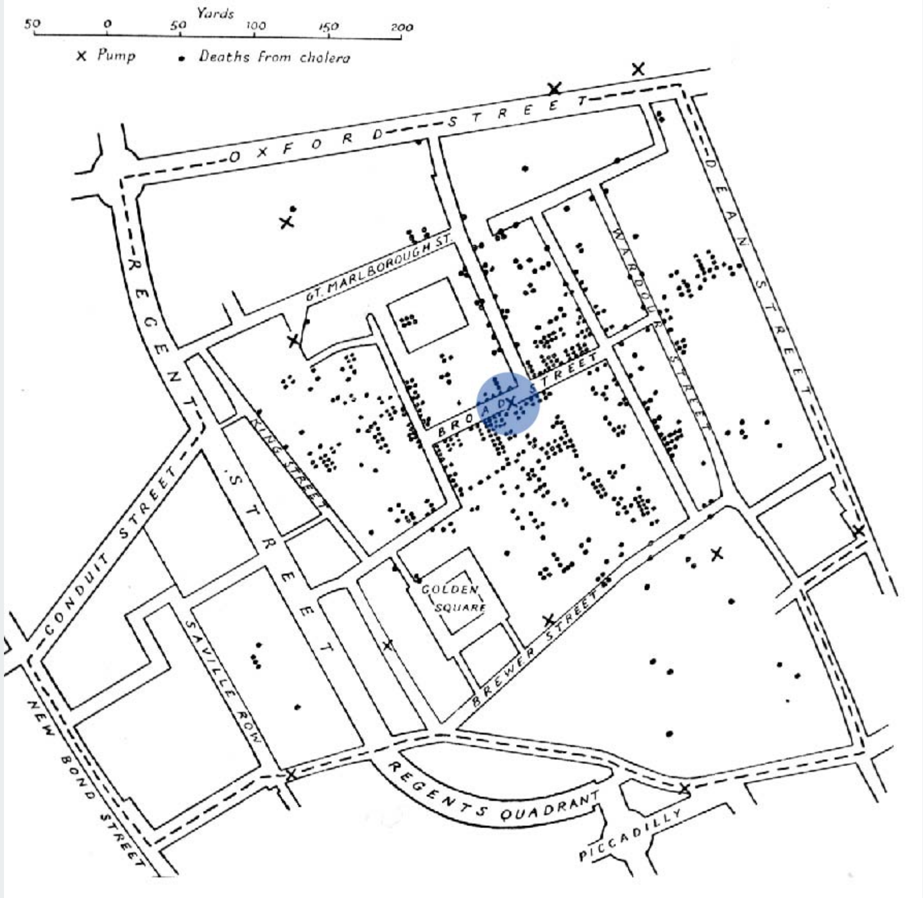


- primitive operations
- convex hull
- closest pair
- **voronoi diagrams**

1854 Cholera Outbreak, Golden Square, London

Life-or-death question:

Given a new cholera patient p, which water pump is closest to p's home?



http://content.answers.com/main/content/wp/en/c/c7/Snow-cholera-map.jpg

Nearest-neighbor problem

Input.

N Euclidean points.

Nearest neighbor problem.

Given a query point p, which one of original N points is closest to p?

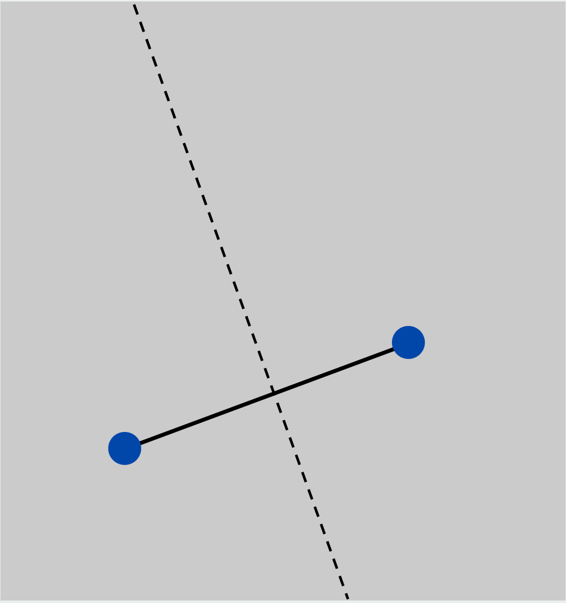
Algorithm	Preprocess	Query
Brute	1	N
Goal	N log N	log N

Voronoi Diagram

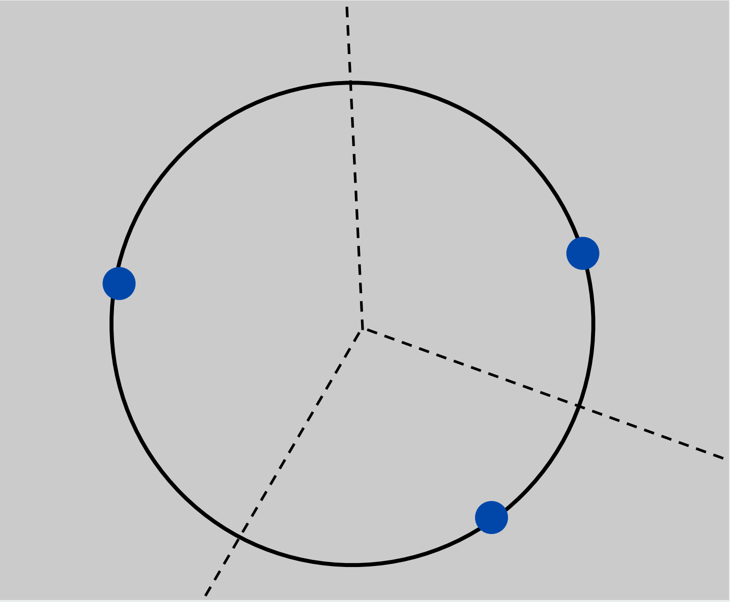
Voronoi region. Set of all points closest to a given point.

Voronoi diagram. Planar subdivision delineating Voronoi regions.

Fact. Voronoi edges are perpendicular bisector segments.



Voronoi of 2 points
(perpendicular bisector)



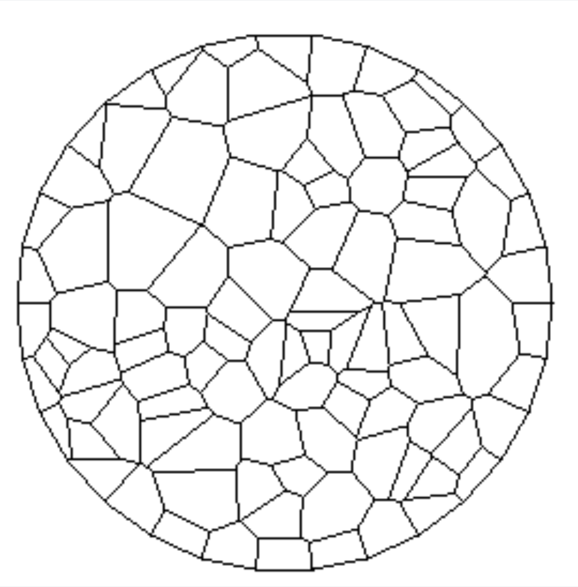
Voronoi of 3 points
(passes through circumcenter)

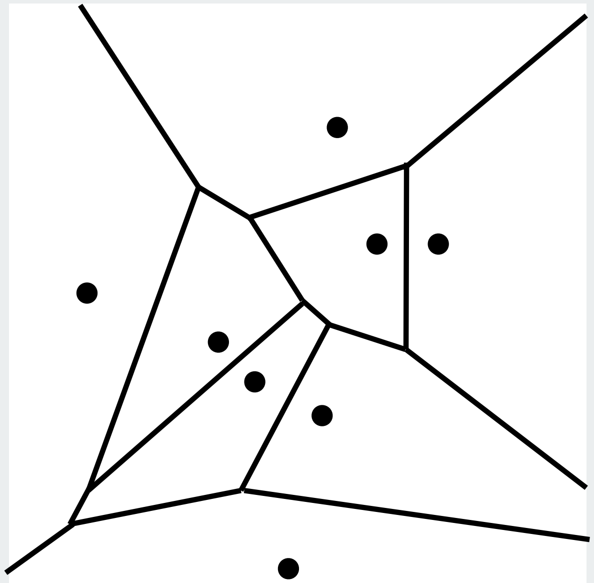
Voronoi Diagram

Voronoi region. Set of all points closest to a given point.

Voronoi diagram. Planar subdivision delineating Voronoi regions.

Fact. Voronoi edges are perpendicular bisector segments.





Quintessential nearest neighbor data structure.

Voronoi Diagram: Applications

Toxic waste dump problem. N homes in a region. Where to locate nuclear power plant so that it is far away from any home as possible?

looking for largest empty circle
(center must lie on Voronoi diagram)

Path planning. Circular robot must navigate through environment with N obstacle points. How to minimize risk of bumping into a obstacle?

robot should stay on Voronoi diagram of obstacles

Reference: J. O'Rourke. Computational Geometry.

Voronoi Diagram: More Applications

Anthropology. Identify influence of clans and chiefdoms on geographic regions.

Astronomy. Identify clusters of stars and clusters of galaxies.

Biology, Ecology, Forestry. Model and analyze plant competition.

Cartography. Piece together satellite photographs into large "mosaic" maps.

Crystallography. Study Wigner-Setiz regions of metallic sodium.

Data visualization.   Nearest neighbor interpolation of 2D data.

Finite elements. Generating finite element meshes which avoid small angles.

Fluid dynamics. Vortex methods for inviscid incompressible 2D fluid flow.

Geology. Estimation of ore reserves in a deposit using info from bore holes.

Geo-scientific modeling. Reconstruct 3D geometric figures from points.

Marketing. Model market of US metro area at individual retail store level.

Metallurgy. Modeling "grain growth" in metal films.

Physiology. Analysis of capillary distribution in cross-sections of muscle tissue.

Robotics. Path planning for robot to minimize risk of collision.

Typography. Character recognition, beveled and carved lettering.

Zoology.   Model and analyze the territories of animals.

References: http://voronoi.com,   http://www.ics.uci.edu/~eppstein/geom.html

Scientific Rediscoveries

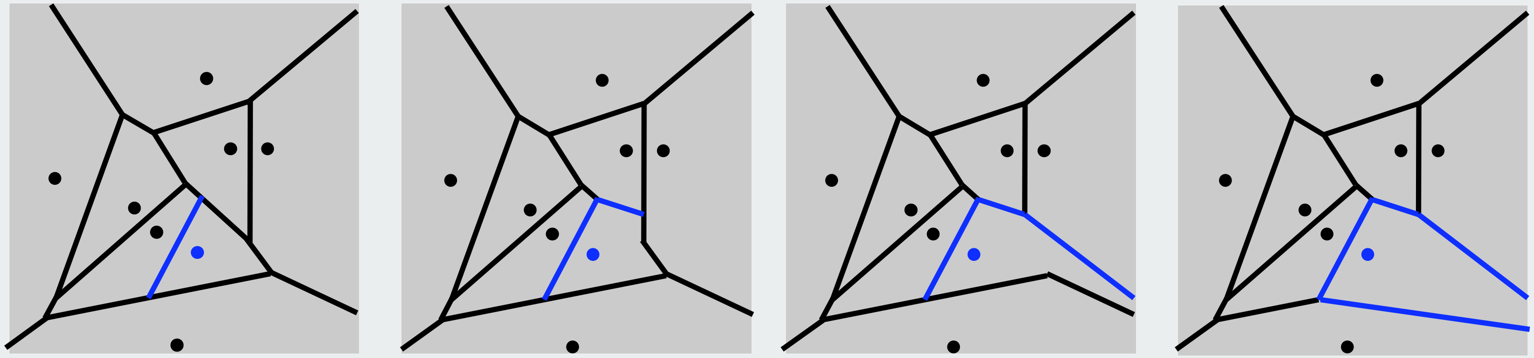
Year	Discoverer	Discipline	Name
1644	Descartes	Astronomy	"Heavens"
1850	Dirichlet	Math	Dirichlet tesselation
1908	Voronoi	Math	Voronoi diagram
1909	Boldyrev	Geology	area of influence polygons
1911	Thiessen	Meteorology	Thiessen polygons
1927	Niggli	Crystallography	domains of action
1933	Wigner-Seitz	Physics	Wigner-Seitz regions
1958	Frank-Casper	Physics	atom domains
1965	Brown	Ecology	area of potentially available
1966	Mead	Ecology	plant polygons
1985	Hoofd et al.	Anatomy	capillary domains

Reference: Kenneth E. Hoff III

Adding a Point to Voronoi Diagram

Challenge. Compute Voronoi.

Basis for incremental algorithms: region containing point gives points to check to compute new Voronoi region boundaries.



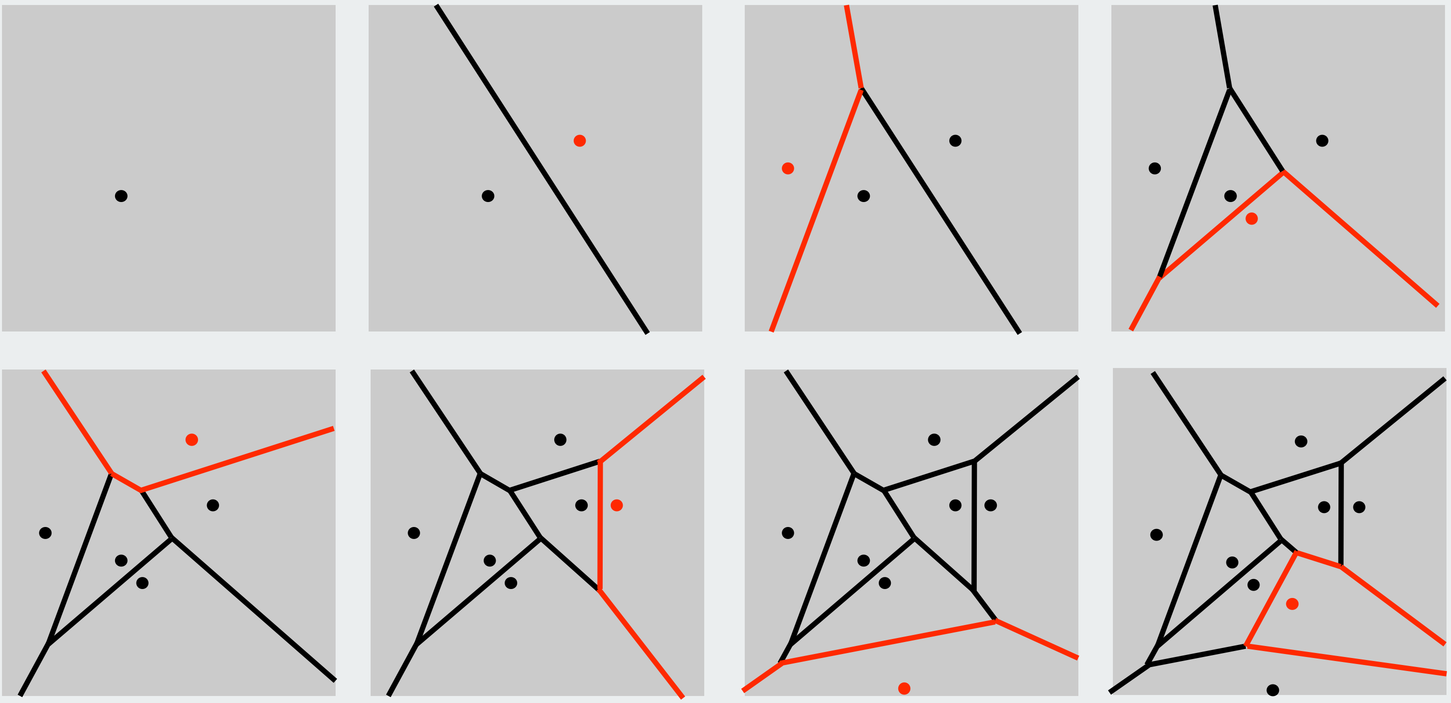
How to represent the Voronoi diagram?
Use multilist associating each point with its Voronoi neighbors

How to find region containing point?
Use Voronoi itself (possible, but not easy!)

Randomized Incremental Voronoi Algorithm

Add points (in random order).

- Find region containing point. ← using Voronoi itself
- Update neighbor regions, create region for new point.



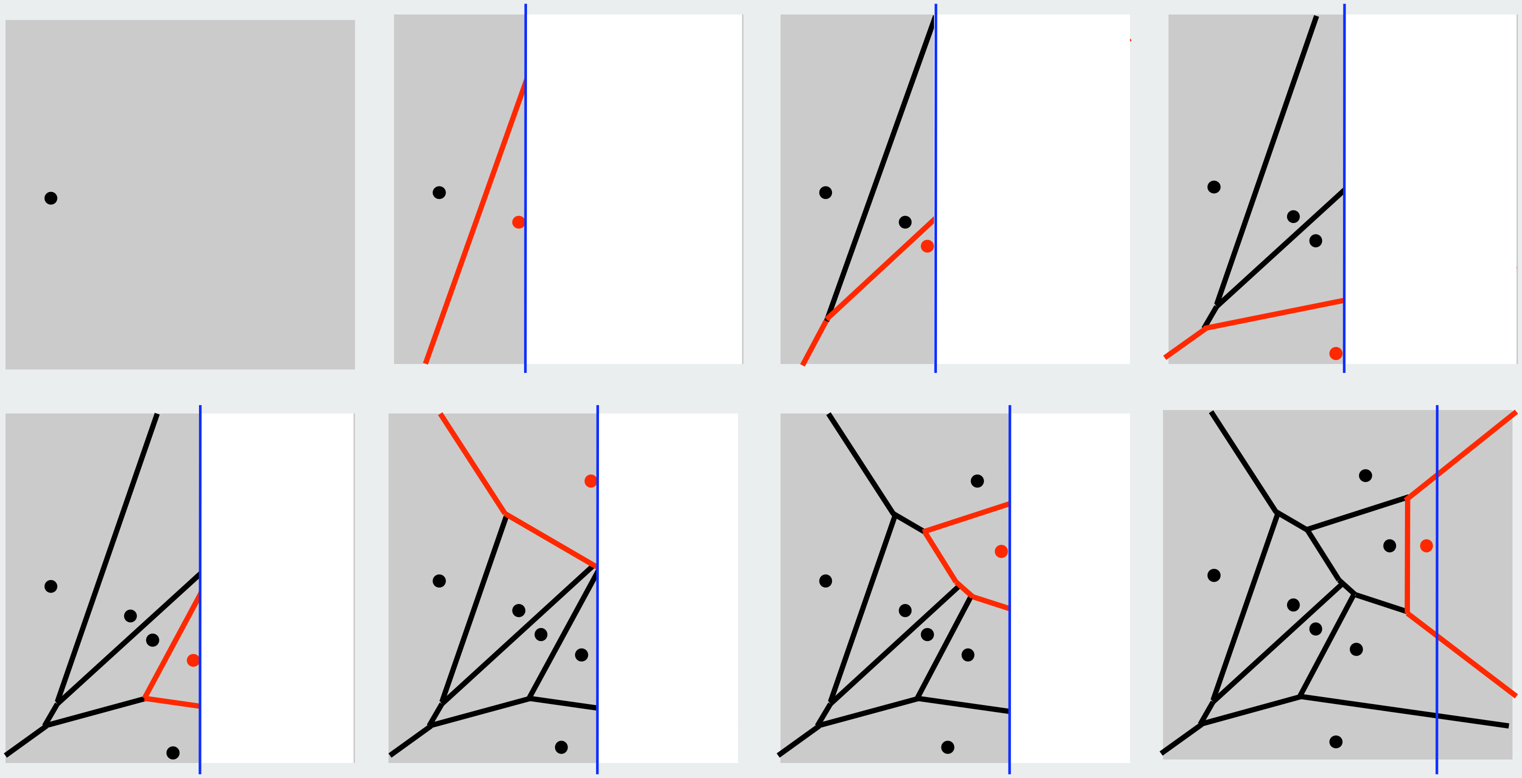
- Running time: O(N log N) on average.

Not an elementary algortihm

Sweep-line Voronoi algorithm

Presort points on x-coordinate

Eliminates point location problem



Fortune's Algorithm

Industrial-strength Voronoi implementation.

- Sweep-line algorithm
- O(N log N) time
- properly handles degeneracies
- properly handles floating-point computations

Algorithm	Preprocess	Query
Brute	1	N
Goal	N log N	log N

Try it yourself!

http://www.diku.dk/hjemmesider/studerende/duff/Fortune/

best animation on the web
student Java project
"lost" the source
decompiled source available

Interface between numeric and combinatorial computing

- exact calculations impossible (using floating point)
- exact calculations required!
- one solution: randomly jiggle the points

Fortune's algorithm in action

http://www.diku.dk/hjemmesider/studerende/duff/Fortune/

Fortune's algorithm in action

Fortune's algorithm in action

Fortune's algorithm in action

Fortune's algorithm in action

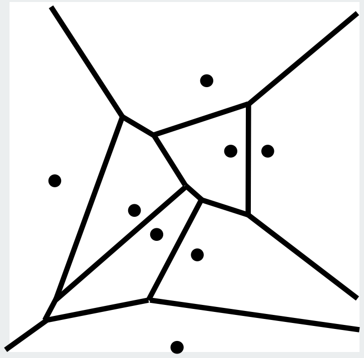
Geometric-algorithm challenge

Problem: Draw a Voronoi diagram

Goals: lecture slide, book diagram

How difficult?

1) any COS126 student could do it
2) need to be a typical diligent COS226 student
3) hire an expert
4) intractable
5) no one knows
6) impossible



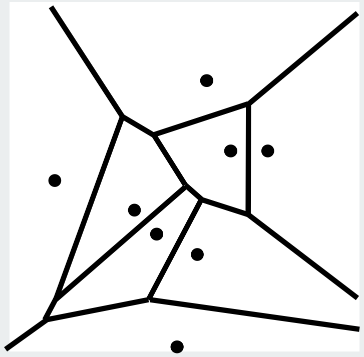
Geometric-algorithm challenge

Problem: Draw a Voronoi diagram
Goals: lecture slide, book diagram

surprise!

How difficult?

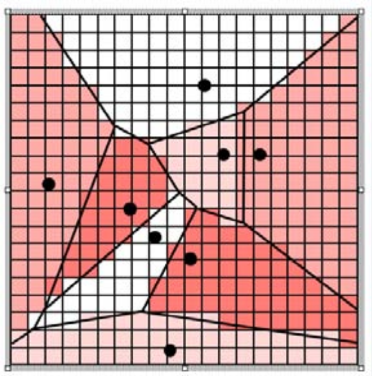
✓ 1) any COS126 student could do it
2) need to be a typical diligent COS226 student
3) hire an expert
4) intractable
5) no one knows
6) impossible



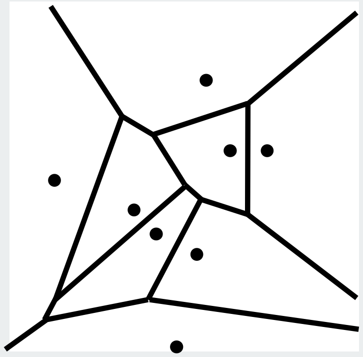
Discretized Voronoi diagram

Observation: to draw a Voronoi diagram, only need an approximation

Ex: Assign a color to each pixel corresponding to its nearest neighbor



P pixels



An effective approximate solution to the nearest neighbor problem

Algorithm	Preprocess	Query
Brute	1	N
Fortune	N log N	log N
Discretized	N P	1

complicated alg (stay tuned)

Discretized Voronoi: Java Implementation

InteractiveDraw. Version of **StdDraw** that supports user interaction.

DrawListener. Interface to support **InteractiveDraw** callbacks.

```
public class Voronoi implements DrawListener
{
   private int SIZE = 512;
   private Point[][] nearest = new Point[SIZE][SIZE];
   private InteractiveDraw draw;
   public Voronoi()
   {
      draw = new InteractiveDraw(SIZE, SIZE);
      draw.setScale(0, 0, SIZE, SIZE);
      draw.addListener(this);
      draw.show();
   }

   public void keyTyped(char c) { }
   public void mouseDragged (double x, double y) { }
   public void mouseReleased(double x, double y) { }
   public void mousePressed
   {  /* See next slide  */  }

}
```

← send callbacks to **Voronoi**

http://www.cs.princeton.edu/introcs/35inheritance/Voronoi.java

Discretized Voronoi: Java Implementation

```
public void mousePressed(double x, double y)
{
   Point p = new Point(x, y);
   draw.setColorRandom();
   for (int i = 0; i < SIZE; i++)
      for (int j = 0; j < SIZE; j++)
      {
         Point q = new Point(i, j);
         if ((nearest[i][j] == null) ||
             (q.distanceTo(p) < q.distanceTo(nearest[i][j])))
         {
            nearest[i][j] = p;
            draw.moveTo(i, j);
            draw.spot();
         }
      }
   draw.setColor(StdDraw.BLACK);
   draw.moveTo(x, y);
   draw.spot(4);
   draw.show();
}
```

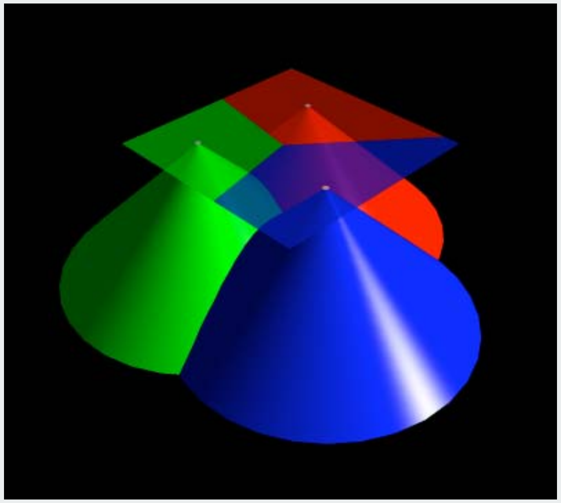
user clicks (x, y)

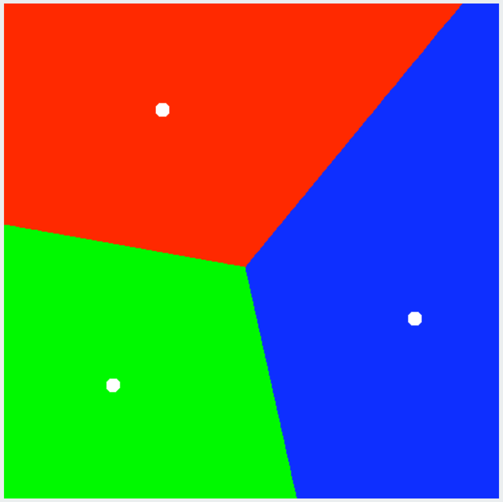
check every other point q to see if p
became its nearest neighbor

Voronoi alternative 2: Hoff's algorithm

Hoff's algorithm. Align apex of a right circular cone with sites.

- Minimum envelope of cone intersections projected onto plane is the Voronoi diagram.
- View cones in different colors ⇒ render Voronoi.



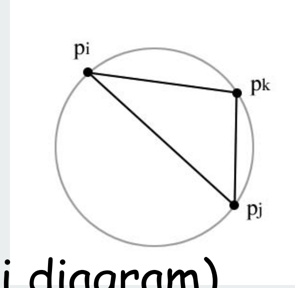


Implementation. Draw cones using standard graphics hardware!

http://www.cs.unc.edu/~geom/voronoi/siggraph_paper/voronoi.pdf

Delaunay Triangulation

Delaunay triangulation. Triangulation of N points such that no point is inside circumcircle of any other triangle.



Fact 0. It exists and is unique (assuming no degeneracy).

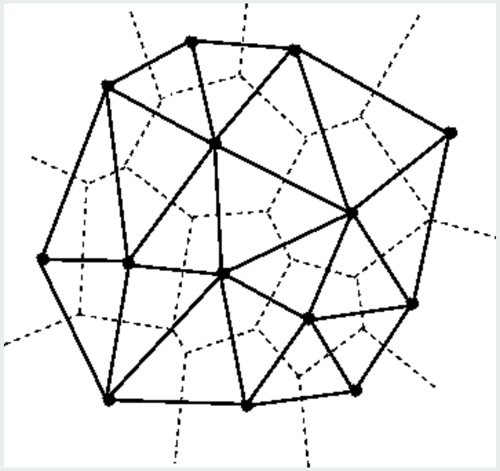
Fact 1. Dual of Voronoi (connect adjacent points in Voronoi diagram).

Fact 2. No edges cross ⇒ O(N) edges.

Fact 3. Maximizes the minimum angle for all triangular elements.

Fact 4. Boundary of Delaunay triangulation is convex hull.

Fact 5. Shortest Delaunay edge connects closest pair of points.



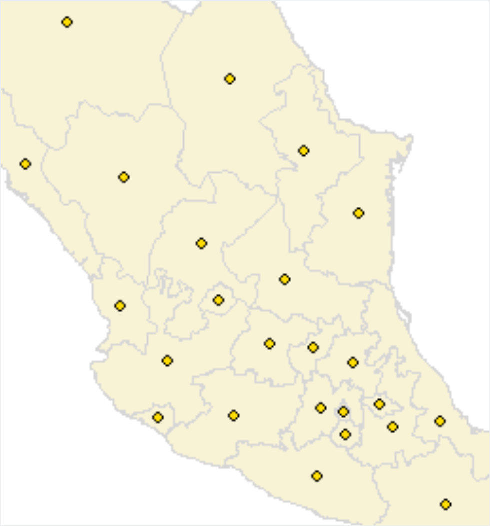
— Delaunay

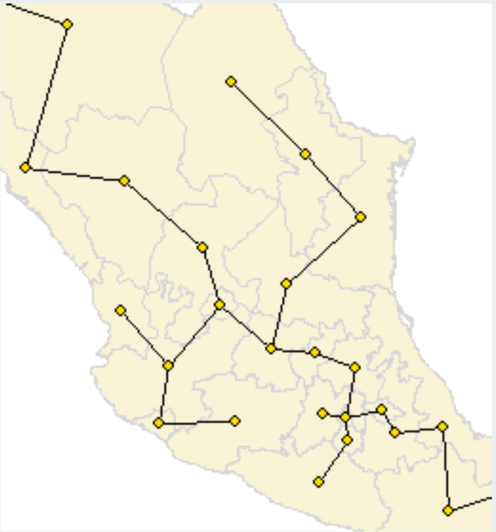
⋯ Voronoi

Euclidean MST

Euclidean MST. Given N points in the plane, find MST connecting them.

- Distances between point pairs are Euclidean distances.





Brute force. Compute $N^2 / 2$ distances and run Prim's algorithm.

Ingenuity.

- MST is subgraph of Delauney triagulation
- Delauney has O(N) edges
- Compute Delauney, then use Prim or Kruskal to get MST in O(N log N) !

Summary

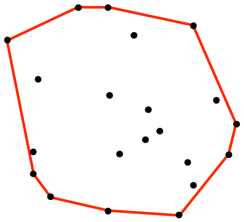
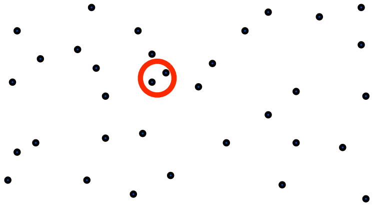
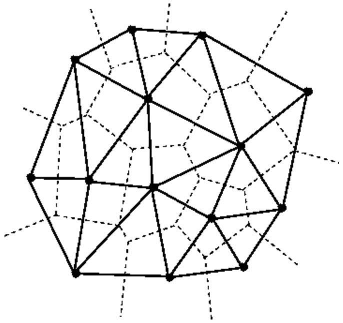
Ingenuity in algorithm design can enable solution of large instances for numerous fundamental geometric problems.

Problem	Brute	Cleverness
convex hull	N^2	N log N
closest pair	N^2	N log N
Voronoi	?	N log N
Delaunay triangulation	N^4	N log N
Euclidean MST	N^2	N log N

asymptotic time to solve a 2D problem with N points

Note: 3D and higher dimensions test limits of our ingenuity

Geometric algorithms summary: Algorithms of the day

		asymptotic time to solve a 2D problem with N points	
		brute	ingenuity
convex hull		N^2	N log N
closest pair		N^2	N log N
Voronoi/Delauney		N^4	N log N
Euclidean MST		N^2	N log N

Geometric Algorithms

- range search
- quad and kd trees
- intersection search
- VLSI rules check

References:

Algorithms in C (2nd edition), Chapters 26-27

http://www.cs.princeton.edu/introalgsds/73range

http://www.cs.princeton.edu/introalgsds/74intersection

Overview

Types of data. Points, lines, planes, polygons, circles, ...

This lecture. Sets of N objects.

Geometric problems extend to higher dimensions.

- Good algorithms also extend to higher dimensions.
- Curse of dimensionality.

Basic problems.

- Range searching.
- Nearest neighbor.
- Finding intersections of geometric objects.

- range search
- quad and kd trees
- intersection search
- VLSI rules check

1D Range Search

Extension to symbol-table ADT with comparable keys.

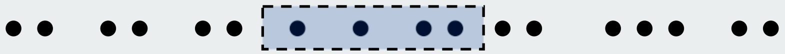
- Insert key-value pair.
- Search for key k.
- How many records have keys between k_1 and k_2?
- Iterate over all records with keys between k_1 and k_2.

Application: database queries.

insert B	B
insert D	B D
insert A	A B D
insert I	A B D I
insert H	A B D H I
insert F	A B D F H I
insert P	A B D F H I P
count G to K	2
search G to K	H I

Geometric intuition.

- Keys are point on a line.
- How many points in a given interval?



1D Range search: implementations

Range search. How many records have keys between k_1 and k_2?

Ordered array. Slow insert, binary search for k_1 and k_2 to find range.

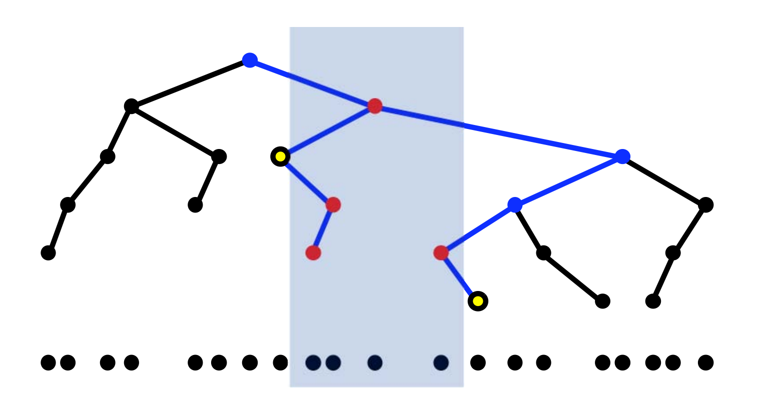
Hash table. No reasonable algorithm (key order lost in hash).

BST. In each node x, maintain number of nodes in tree rooted at x.

Search for smallest element ≥ k_1 and largest element ≤ k_2.

	insert	count	range
ordered array	N	log N	R + log N
hash table	1	N	N
BST	log N	log N	R + log N

N = # records
R = # records that match



● nodes examined
● within interval
● not touched

2D Orthogonal Range Search

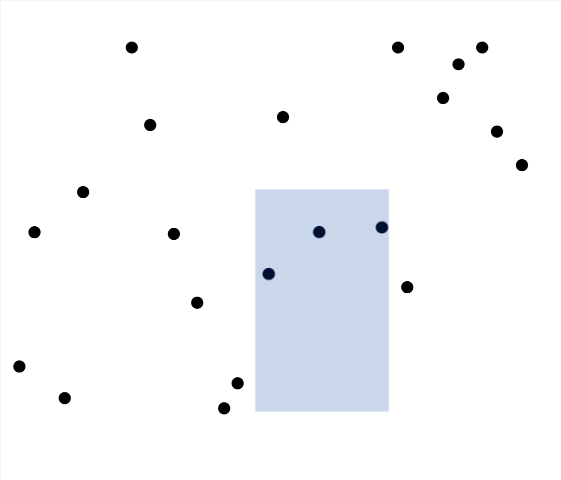
Extension to symbol-table ADT with 2D keys.

- Insert a 2D key.
- Search for a 2D key.
- Range search: find all keys that lie in a 2D range?
- Range count: how many keys lie in a 2D range?

Applications: networking, circuit design, databases.

Geometric interpretation.

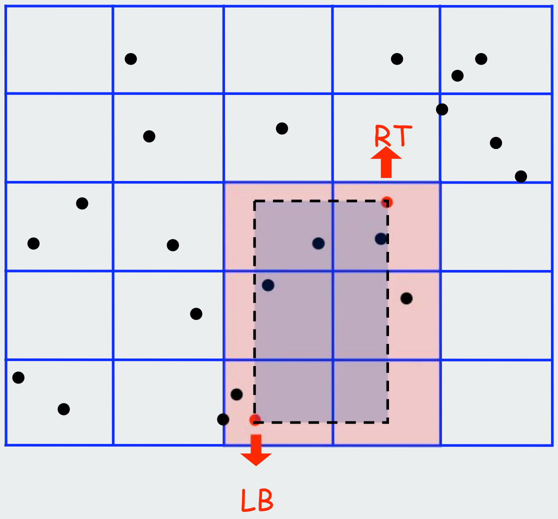
- Keys are point in the plane
- Find all points in a given h-v rectangle



2D Orthogonal range Search: Grid implementation

Grid implementation. [Sedgewick 3.18]

- Divide space into M-by-M grid of squares.
- Create linked list for each square.
- Use 2D array to directly access relevant square.
- Insert: insert (x, y) into corresponding grid square.
- Range search: examine only those grid squares that could have points in the rectangle.



2D Orthogonal Range Search: Grid Implementation Costs

Space-time tradeoff.

- Space: $M^2 + N$.
- Time: $1 + N / M^2$ per grid cell examined on average.

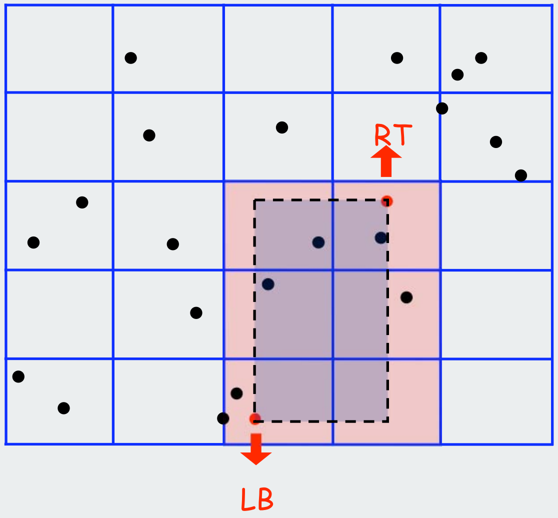
Choose grid square size to tune performance.

- Too small: wastes space.
- Too large: too many points per grid square.
- Rule of thumb: $\sqrt{N}$ by $\sqrt{N}$ grid.

Running time. [if points are evenly distributed]

- Initialize: O(N).
- Insert: O(1).
- Range: O(1) per point in range.

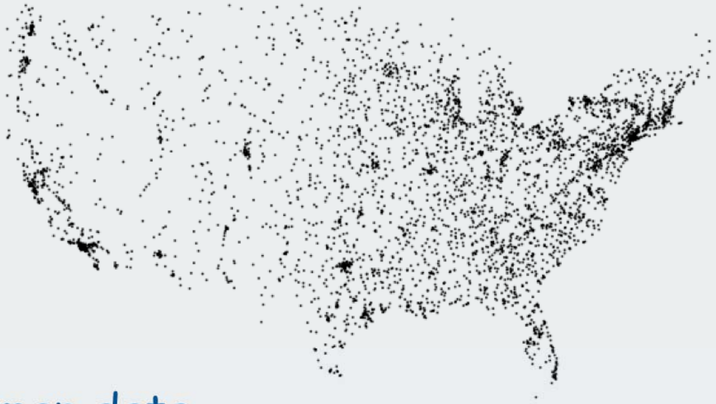
$M \approx \sqrt{N}$

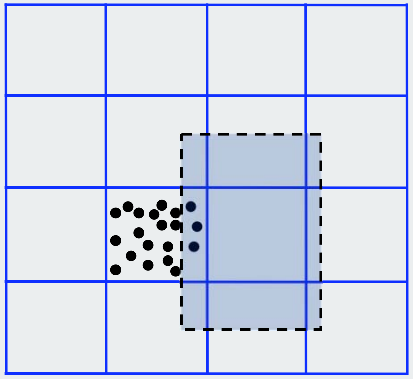


Clustering

Grid implementation. Fast, simple solution for well-distributed points.

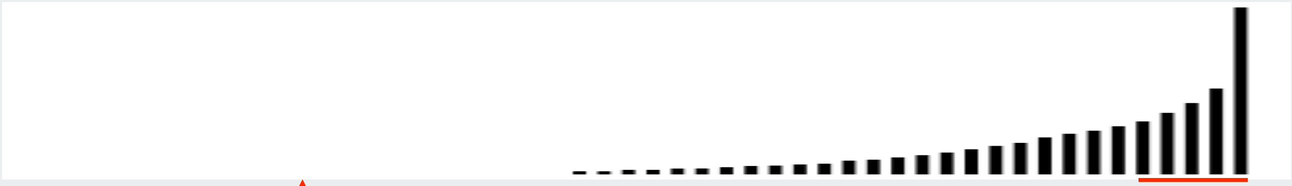
Problem. Clustering is a well-known phenomenon in geometric data.





Ex: USA map data.

13,000 points, 1000 grid squares.



half the squares are empty

half the points are in 10% of the squares

Lists are too long, even though average length is short.

Need data structure that gracefully adapts to data.

- range search
- **quad and kd trees**
- intersection search
- VLSI rules check

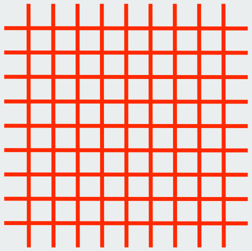
Space Partitioning Trees

Use a tree to represent a recursive subdivision of d-dimensional space.

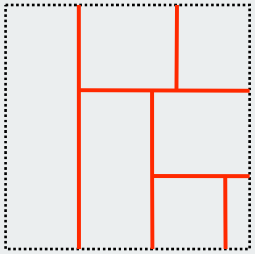
BSP tree. Recursively divide space into two regions.
Quadtree. Recursively divide plane into four quadrants.
Octree. Recursively divide 3D space into eight octants.
kD tree. Recursively divide k-dimensional space into two half-spaces.
[possible but much more complicated to define Voronoi-based structures]

Applications.

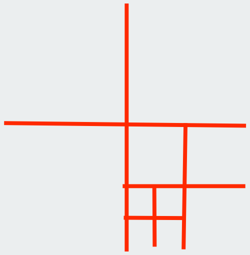
- Ray tracing.
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.



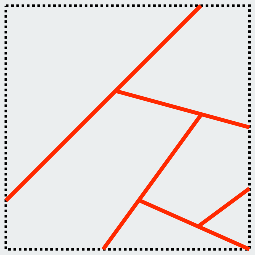
Grid



kD tree



Quadtree



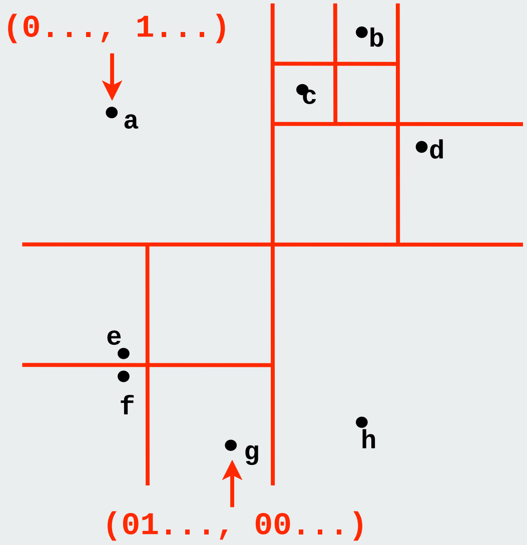
BSP tree

Quadtree

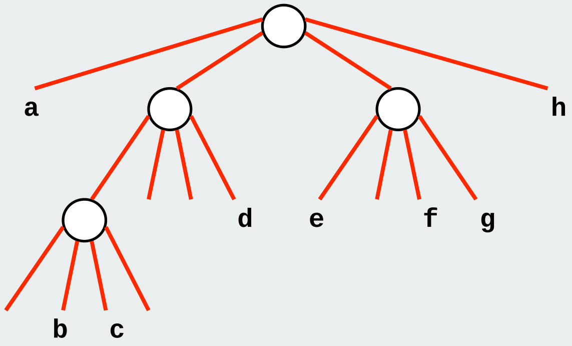
Recursively partition plane into 4 quadrants.

Implementation: 4-way tree.

actually a trie
partitioning on bits of coordinates



```
public class QuadTree
{
   private Quad quad;
   private Value value;
   private QuadTree NW, NE, SW, SE;
}
```

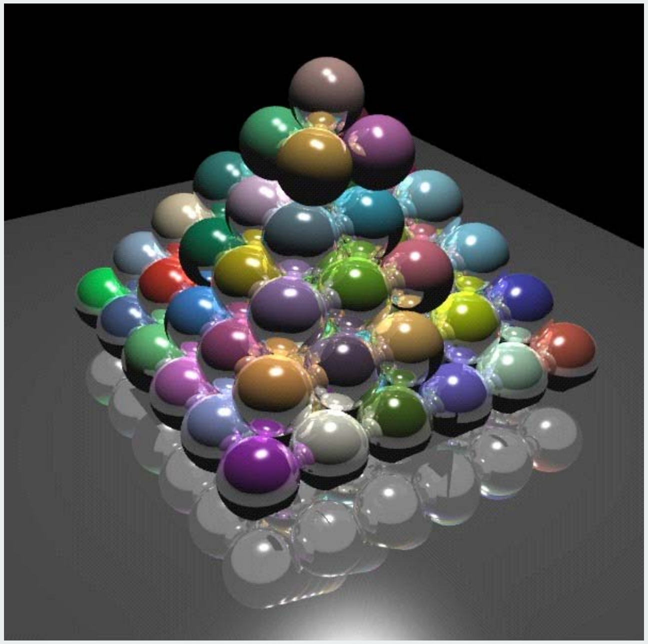


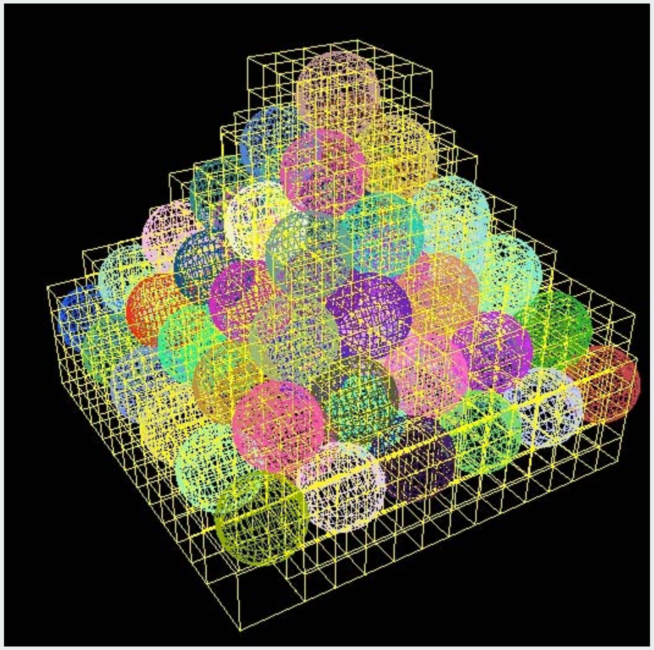
Primary reason to choose quad trees over grid methods:
good performance in the presence of clustering

Curse of Dimensionality

Range search / nearest neighbor in k dimensions?
Main application. Multi-dimensional databases.

3D space. Octrees: recursively divide 3D space into 8 octants.
100D space. Centrees: recursively divide into 2^{100} centrants???





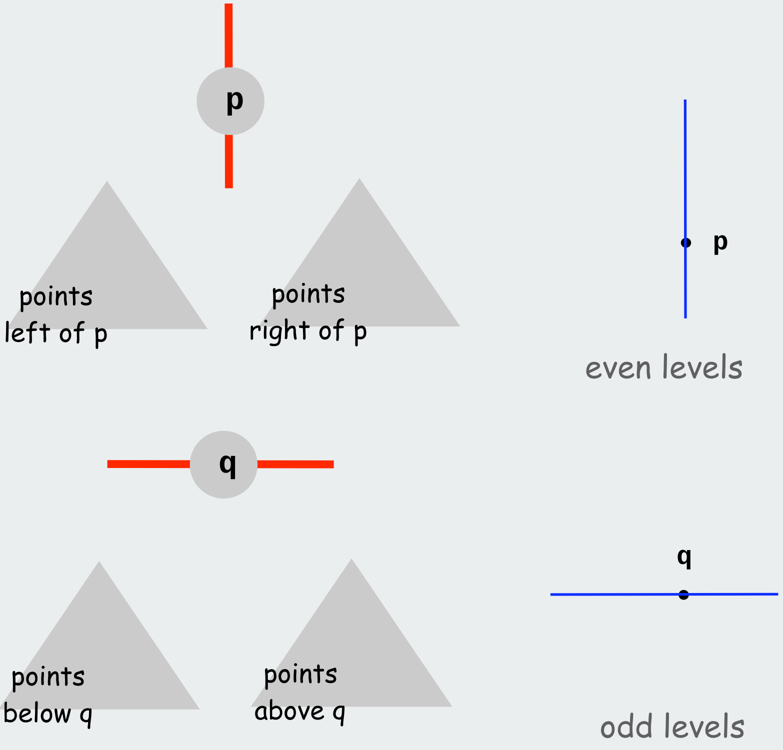
Raytracing with octrees
http://graphics.cs.ucdavis.edu/~gregorsk/graphics/275.html

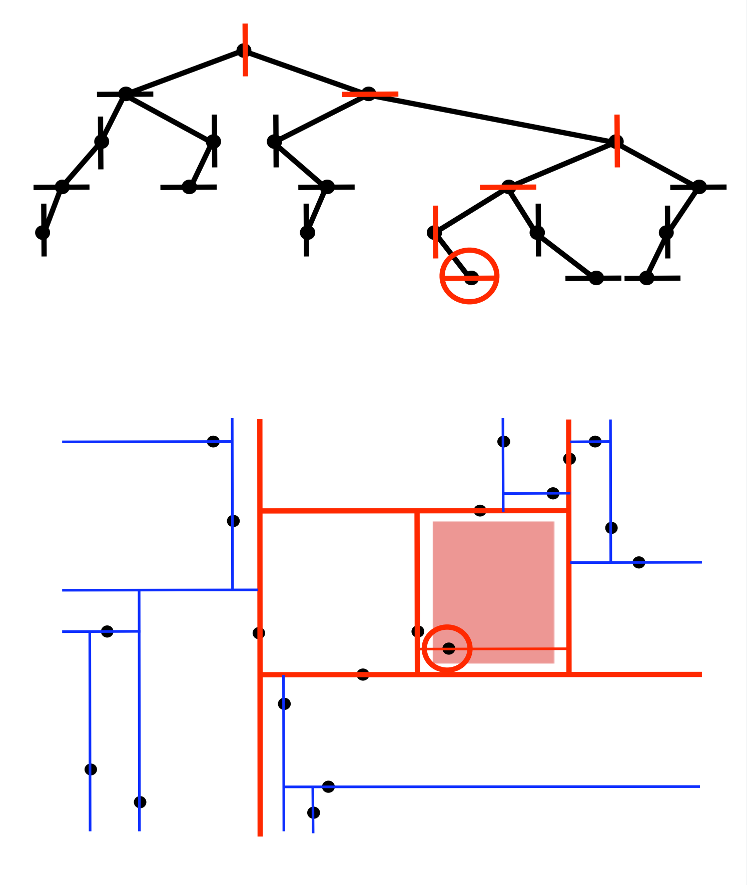
2D Trees

Recursively partition plane into 2 halfplanes.

Implementation: BST, but alternate using x and y coordinates as key.

- Search gives rectangle containing point.
- Insert further subdivides the plane.





Near Neighbor Search

Useful extension to symbol-table ADT for records with metric keys.

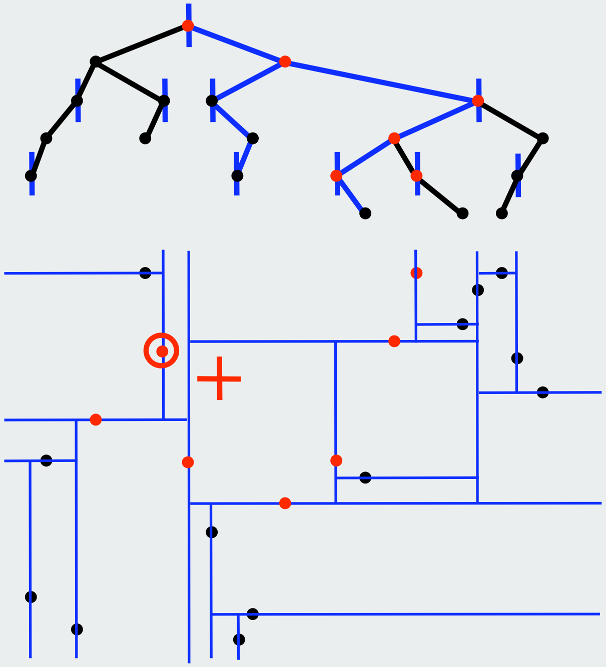
- Insert a k dimensional point.
- Near neighbor search: given a point p, which point in data structure is nearest to p?

Need concept of distance, not just ordering.

kD trees provide fast, elegant solution.

- Recursively search subtrees that could have near neighbor (may search both).
- O(log N) ?

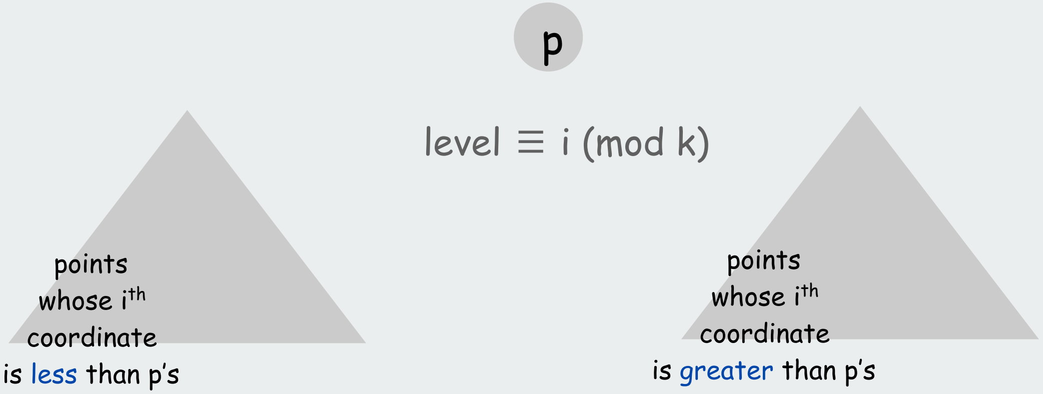
Yes, in practice (but not proven)



kD Trees

kD tree. Recursively partition k-dimensional space into 2 halfspaces.

Implementation: BST, but cycle through dimensions ala 2D trees.

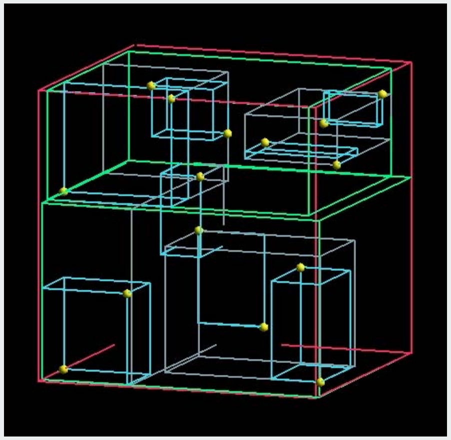


p

level ≡ i (mod k)

points whose i^{th} coordinate is less than p's

points whose i^{th} coordinate is greater than p's



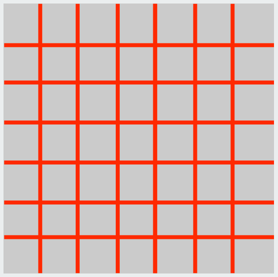
Efficient, simple data structure for processing k-dimensional data.

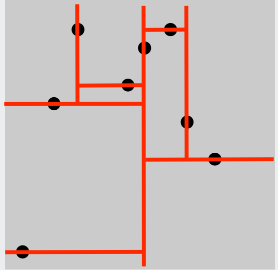
- adapts well to clustered data.
- adapts well to high dimensional data.
- widely used.
- discovered by an undergrad in an algorithms class!

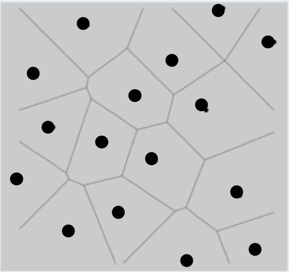
Summary

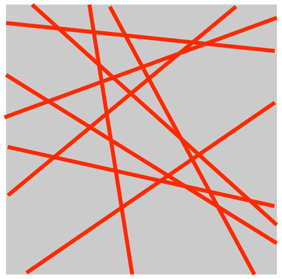
Basis of many geometric algorithms: search in a planar subdivision.

	grid	2D tree	Voronoi diagram	intersecting lines
basis	√N h-v lines	N points	N points	√N lines
representation	2D array of N lists	N-node BST	N-node multilist	~N-node BST
cells	~N squares	N rectangles	N polygons	~N triangles
search cost	1	log N	log N	log N
extend to kD?	too many cells	easy	cells too complicated	use (k-1)D hyperplane







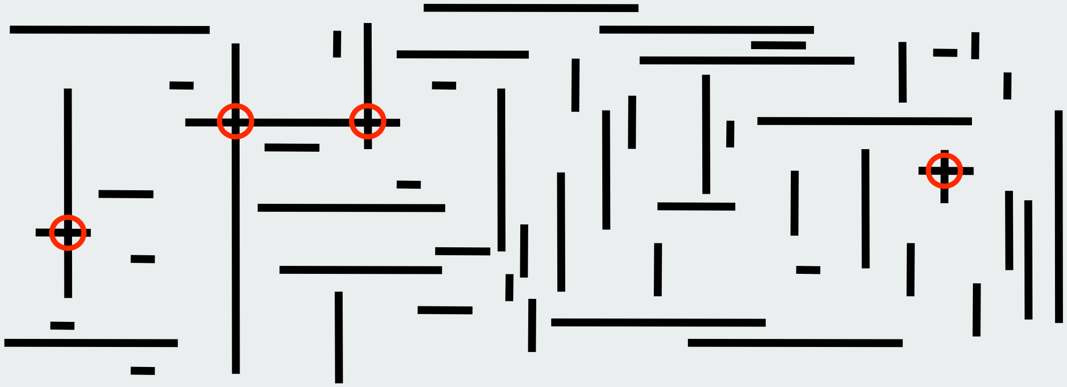


- range search
- quad and kd trees
- **intersection search**
- VLSI rules check

Search for intersections

Problem. Find all intersecting pairs among set of N geometric objects.
Applications. CAD, games, movies, virtual reality.

Simple version: 2D, all objects are horizontal or vertical line segments.

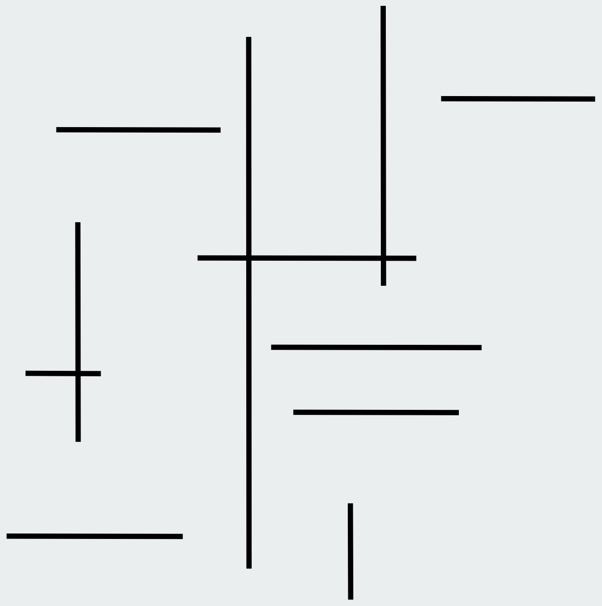


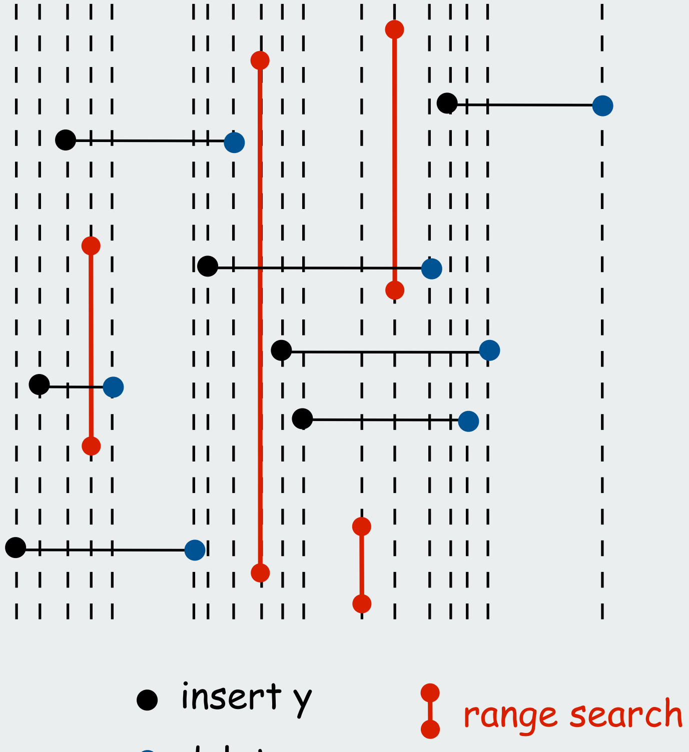
Brute force. Test all $\Theta(N^2)$ pairs of line segments for intersection.
Sweep line. Efficient solution extends to 3D and general objects.

Orthogonal segment intersection search: Sweep-line algorithm

Sweep vertical line from left to right.

- x-coordinates define events.
- left endpoint of h-segment: insert y coordinate into ST.
- right endpoint of h-segment: remove y coordinate from ST.
- v-segment: range search for interval of y endpoints.





Orthogonal segment intersection: Sweep-line algorithm

Reduces 2D orthogonal segment intersection search to 1D range search!

Running time of sweep line algorithm.

• Put x-coordinates on a PQ (or sort).	$O(N \log N)$
• Insert y-coordinate into SET.	$O(N \log N)$
• Delete y-coordinate from SET.	$O(N \log N)$
• Range search.	$O(R + N \log N)$

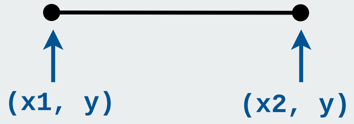
N = # line segments
R = # intersections

Efficiency relies on judicious use of data structures.

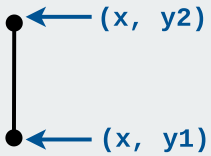
Immutable H-V segment ADT

```
public final class SegmentHV implements Comparable<SegmentHV>
{
   public final int x1, y1;
   public final int x2, y2;

   public SegmentHV(int x1, int y1, int x2, int y2)
   {  ...  }
   public boolean isHorizontal()
   {  ...  }
   public boolean isVertical()
   {  ...  }
   public int compareTo(SegmentHV b)
   {  ...  }
   public String toString()
   {  ...  }
}
```

compare by x-coordinate;
break ties by y-coordinate



horizontal segment



vertical segment

Sweep-line event

```
public class Event implements Comparable<Event>
{
   private int time;
   private SegmentHV segment;

   public Event(int time, SegmentHV segment)
   {
      this.time    = time;
      this.segment = segment;
   }

   public int compareTo(Event b)
   {
      return a.time - b.time;
   }
}
```

Sweep-line algorithm: Initialize events

```
MinPQ<Event> pq = new MinPQ<Event>();

for (int i = 0; i < N; i++)
{
   if (segments[i].isVertical())
   {
      Event e = new Event(segments[i].x1, segments[i]);
      pq.insert(e);
   }
   else if (segments[i].isHorizontal())
   {
      Event e1 = new Event(segments[i].x1, segments[i]);
      Event e2 = new Event(segments[i].x2, segments[i]);
      pq.insert(e1);
      pq.insert(e2);
   }
}
```

initialize PQ

vertical segment

horizontal segment

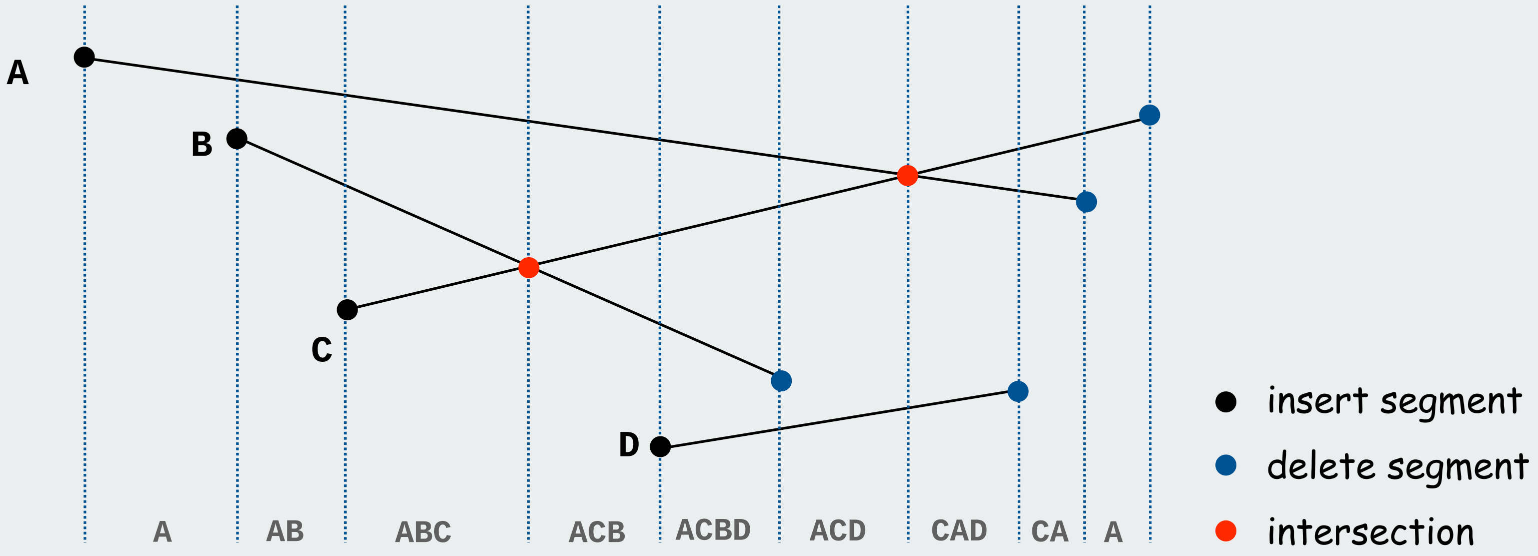
Sweep-line algorithm: Simulate the sweep line

```
int INF = Integer.MAX_VALUE;

SET<SegmentHV> set = new SET<SegmentHV>();

while (!pq.isEmpty())
{
   Event e = pq.delMin();
   int sweep = e.time;
   SegmentHV segment = e.segment;

   if (segment.isVertical())
   {
      SegmentHV seg1, seg2;
      seg1 = new SegmentHV(-INF, segment.y1, -INF, segment.y1);
      seg2 = new SegmentHV(+INF, segment.y2, +INF, segment.y2);
      for (SegmentHV seg : set.range(seg1, seg2))
          System.out.println(segment + " intersects " + seg);
   }


   else if (sweep == segment.x1) set.add(segment);
   else if (sweep == segment.x2) set.remove(segment);
}
```

General line segment intersection search

Extend sweep-line algorithm

- Maintain order of segments that intersect sweep line by y-coordinate.
- Intersections can only occur between adjacent segments.
- Add/delete line segment ⇒ one new pair of adjacent segments.
- Intersection ⇒ swap adjacent segments.



order of segments

Line Segment Intersection: Implementation

Efficient implementation of sweep line algorithm.

- Maintain PQ of important x-coordinates: endpoints and intersections.
- Maintain SET of segments intersecting sweep line, sorted by y.
- $O(R \log N + N \log N)$.

to support "next largest"
and "next smallest" queries

Implementation issues.

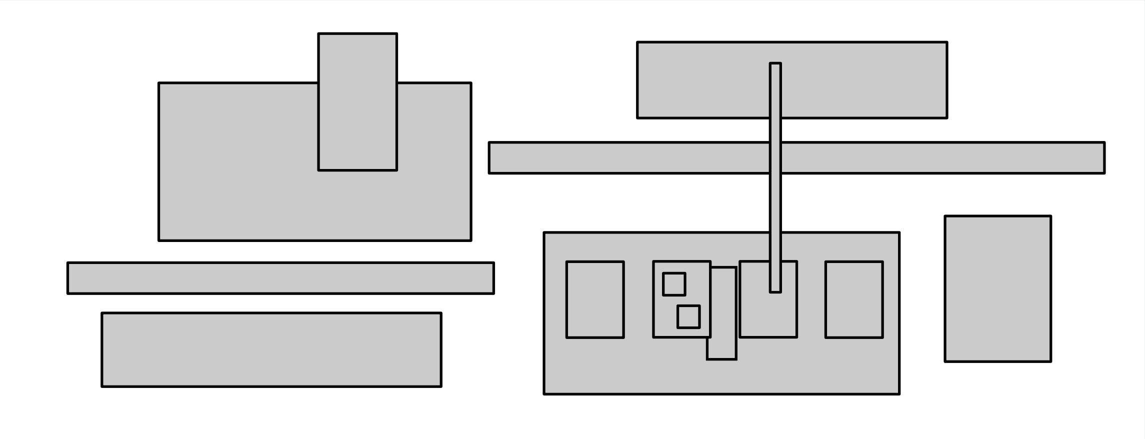
- Degeneracy.
- Floating point precision.
- Use PQ, not presort (intersection events are unknown ahead of time).

- range search
- quad and kd trees
- intersection search
- **VLSI rules check**

Algorithms and Moore's Law

Rectangle intersection search. Find all intersections among h-v rectangles.

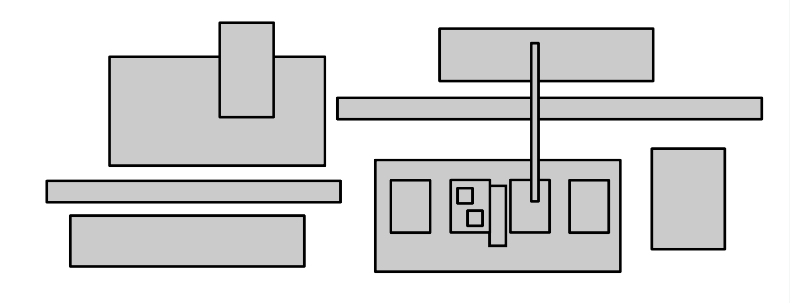
Application. Design-rule checking in VLSI circuits.



Algorithms and Moore's Law

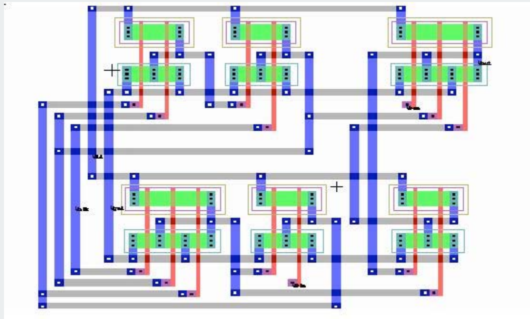
Early 1970s: microprocessor design became a geometric problem.

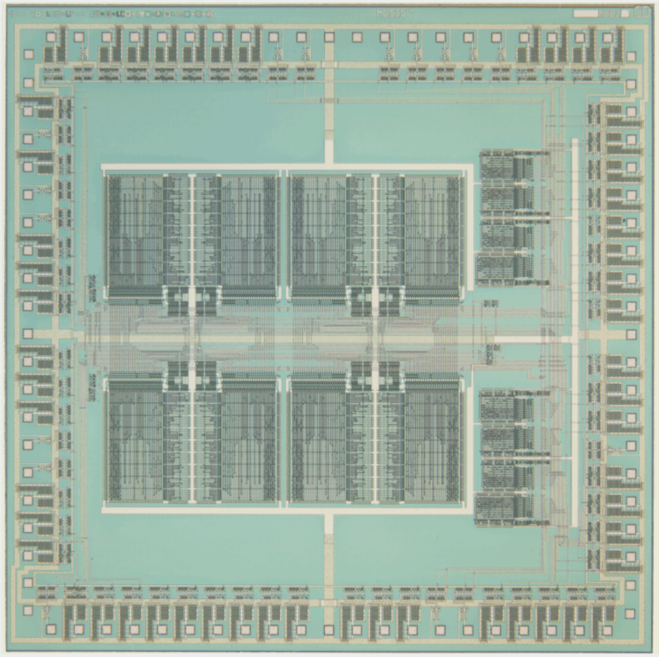
- Very Large Scale Integration (VLSI).
- Computer-Aided Design (CAD).



Design-rule checking:

- certain wires cannot intersect
- certain spacing needed between different types of wires
- debugging = rectangle intersection search





Algorithms and Moore's Law

"Moore's Law." Processing power doubles every 18 months.

- 197x: need to check N rectangles.
- 197(x+1.5): need to check 2N rectangles on a 2x-faster computer.

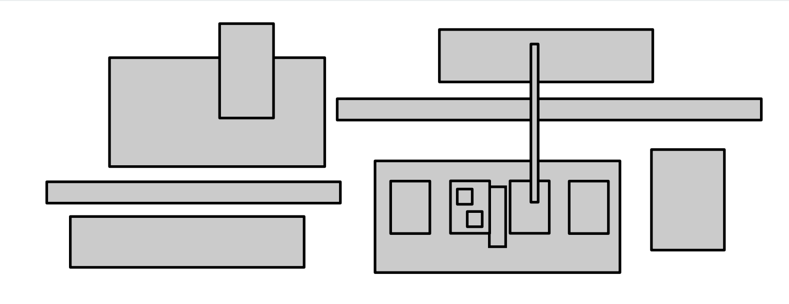
Bootstrapping: we get to use the faster computer for bigger circuits

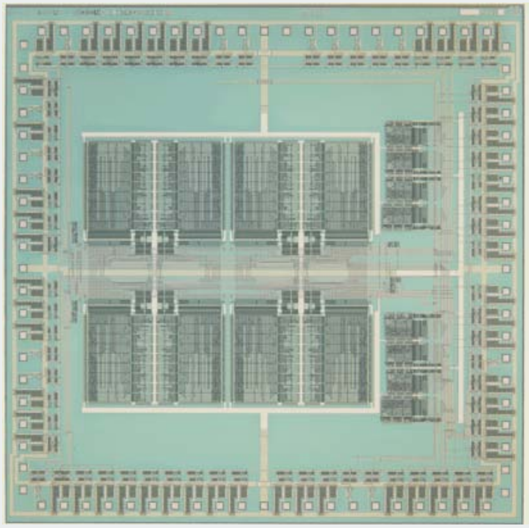
But bootstrapping is not enough if using a quadratic algorithm

- 197x: takes M days.
- 197(x+1.5): takes (4M)/2 = 2M days. (!)

quadratic algorithm

2x-faster computer



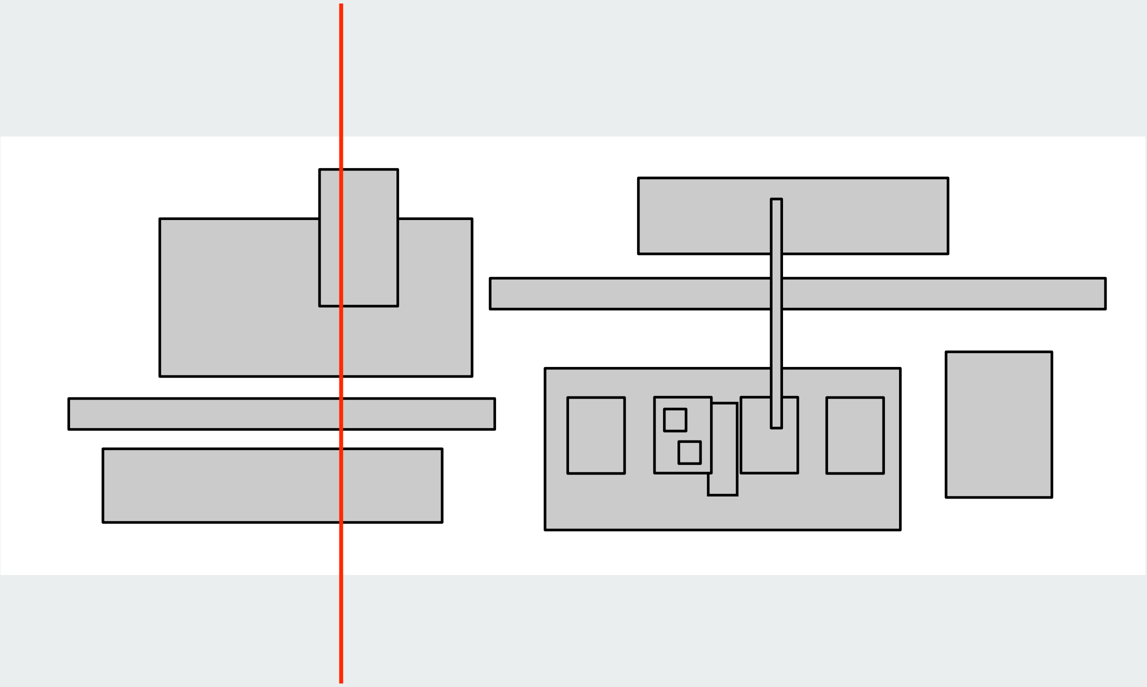


O(N log N) CAD algorithms are necessary to sustain Moore's Law.

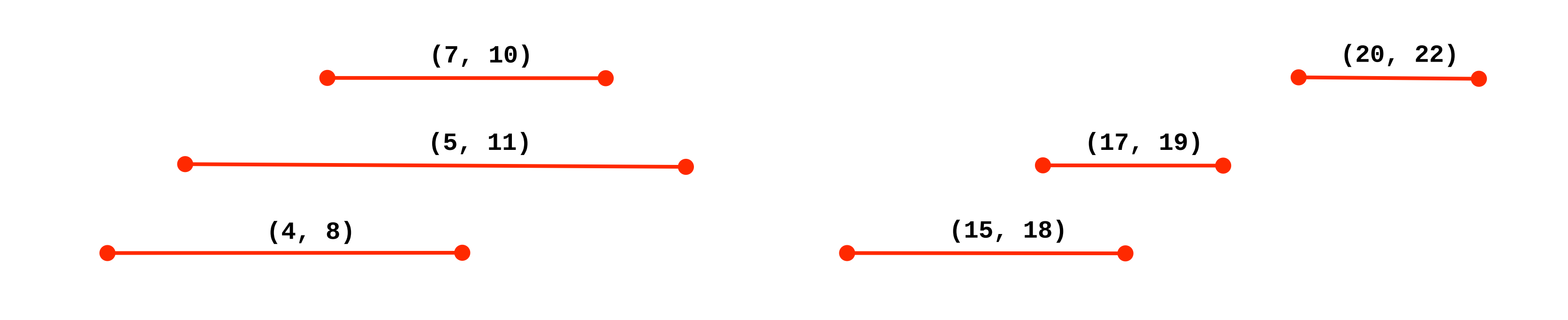
Rectangle intersection search

Move a vertical "sweep line" from left to right.

- Sweep line: sort rectangles by x-coordinate and process in this order, stopping on left and right endpoints.
- Maintain set of intervals intersecting sweep line.
- Key operation: given a new interval, does it intersect one in the set?



Interval Search Trees

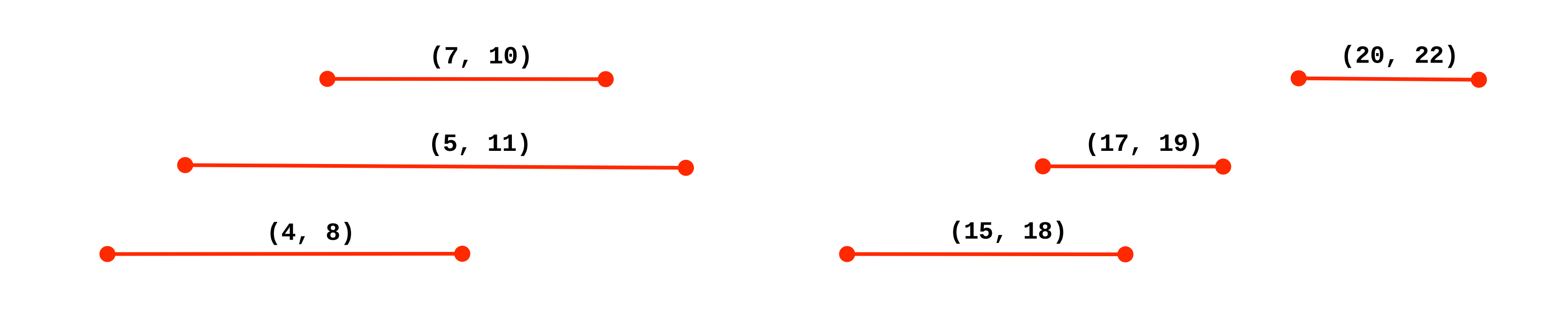


Support following operations.

- Insert an interval `(lo, hi)`.
- Delete the interval `(lo, hi)`.
- Search for an interval that intersects `(lo, hi)`.

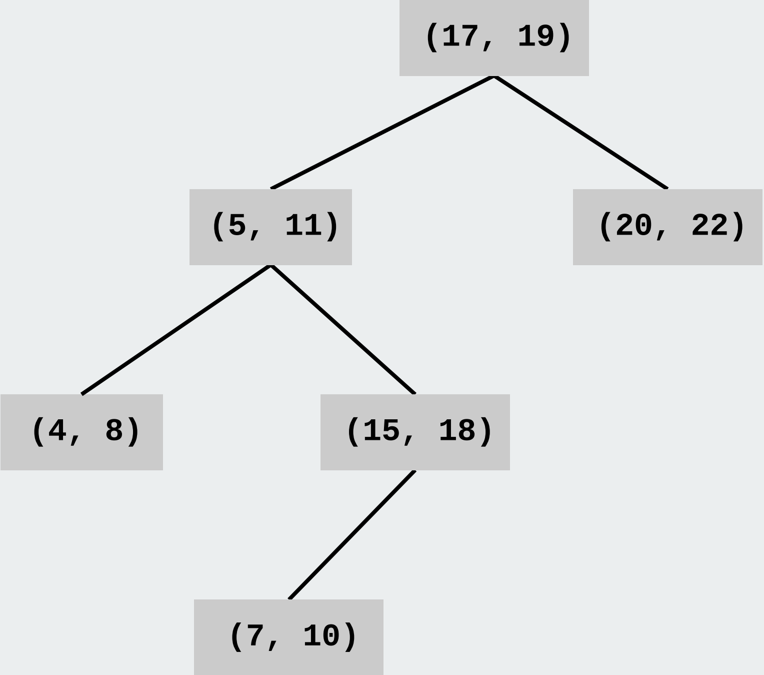
Non-degeneracy assumption. No intervals have the same x-coordinate.

Interval Search Trees

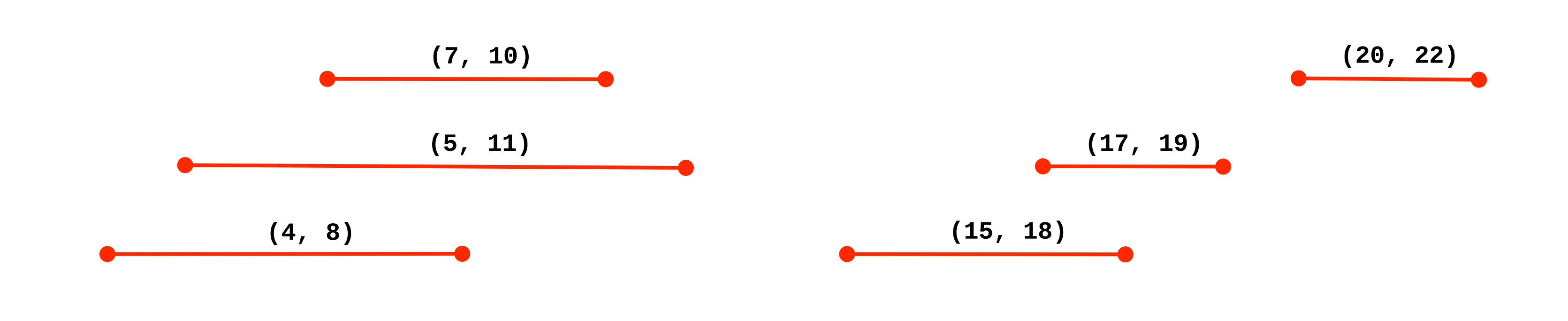


Interval tree implementation with BST.

- Each BST node stores one interval.
- use `lo` endpoint as BST key.

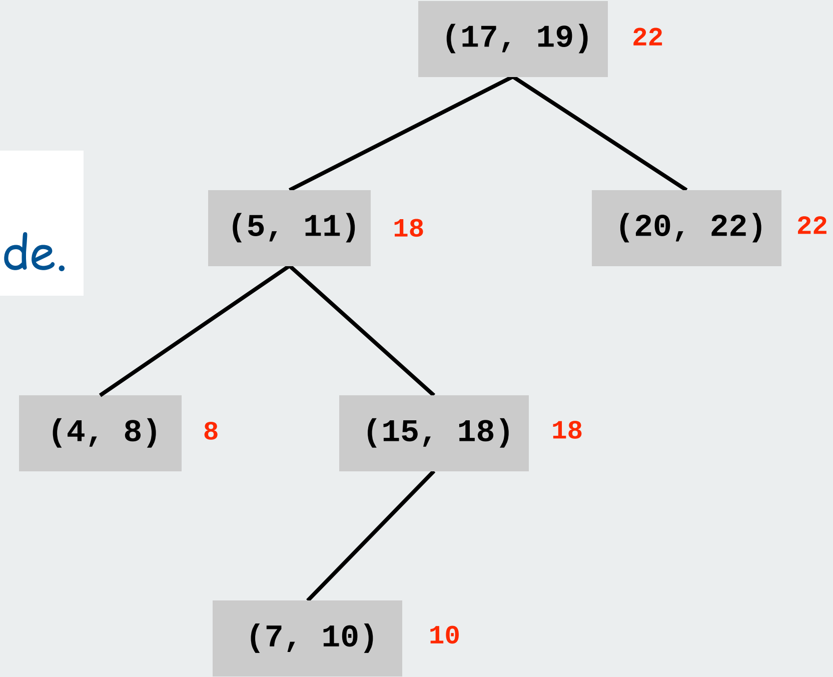


Interval Search Trees



Interval tree implementation with BST.

- Each BST node stores one interval.
- BST nodes sorted on `lo` endpoint.
- Additional info: store and maintain max endpoint in subtree rooted at node.



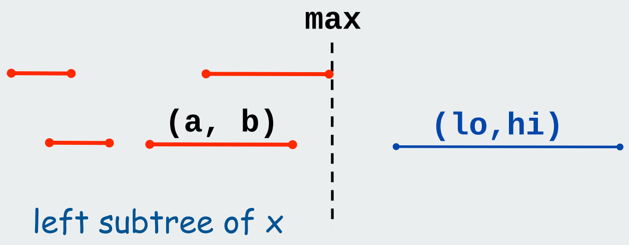
Finding an intersecting interval

Search for an interval that intersects **(lo, hi)**.

```
Node x = root;
while (x != null)
{
   if (x.interval.intersects(lo, hi)) return x.interval;
   else if (x.left == null)  x = x.right;
   else if (x.left.max < lo) x = x.right;
   else                      x = x.left;
}
return null;
```

Case 1. If search goes right, then either

- there is an intersection in right subtree
- there are no intersections in either subtree.



Pf. Suppose no intersection in right.

- **(x.left == null)** ⇒ trivial.
- **(x.left.max < lo)** ⇒ for any interval **(a, b)** in left subtree of x, we have **b** ≤ **max** < **lo**.

defn of max

reason for going right

Finding an intersecting interval

Search for an interval that intersects **(lo, hi)**.

```
Node x = root;
while (x != null)
{
   if (x.interval.intersects(lo, hi)) return x.interval;
   else if (x.left == null)  x = x.right;
   else if (x.left.max < lo) x = x.right;
   else                      x = x.left;
}
return null;
```

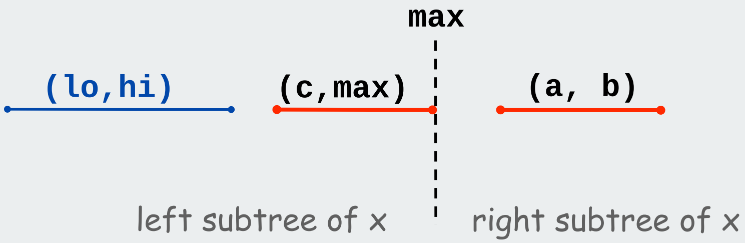
Case 2. If search goes left, then either

- there is an intersection in left subtree
- there are no intersections in either subtree.

Pf. Suppose no intersection in left. Then for any interval **(a, b)** in right subtree, $\mathtt{a} \geq \mathtt{c} > \mathtt{hi} \Rightarrow$ no intersection in right.

intervals sorted by left endpoint

no intersection in left subtree



Interval Search Tree: *Analysis*

Implementation. Use a red-black tree to guarantee performance.

can maintain auxiliary information
using log N extra work per op

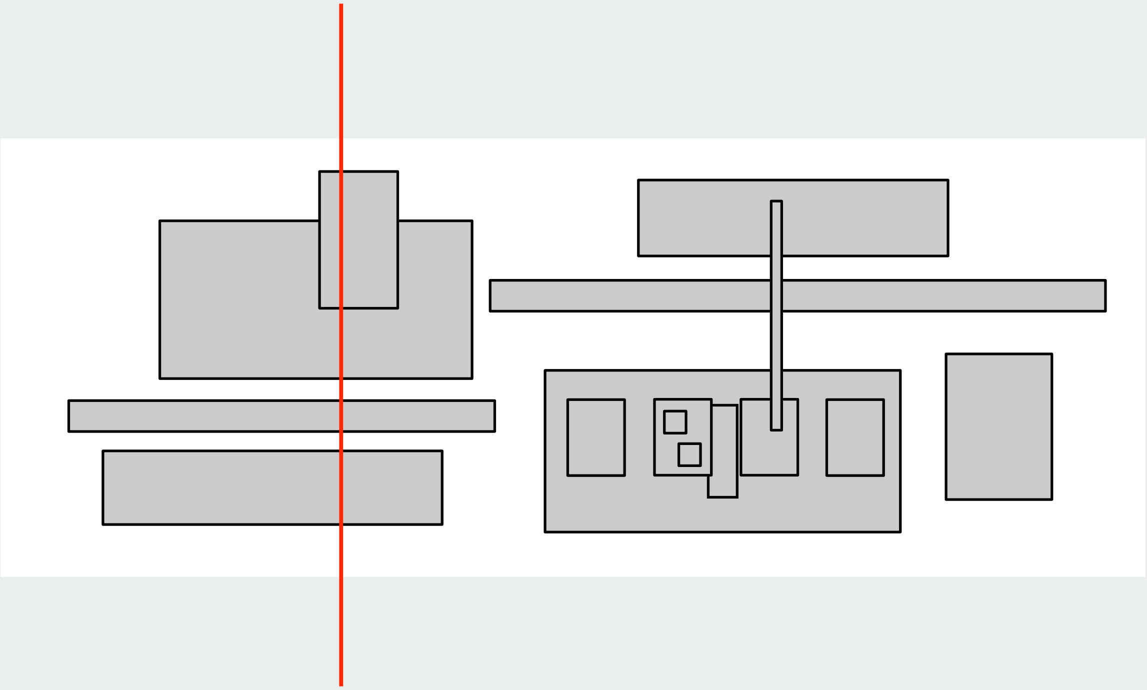
Operation	Worst case
insert interval	log N
delete interval	log N
find an interval that intersects (lo, hi)	log N
find all intervals that intersect (lo, hi)	R log N

N = # intervals
R = # intersections

Rectangle intersection sweep-line algorithm: Review

Move a vertical "sweep line" from left to right.

- Sweep line: sort rectangles by x-coordinates and process in this order.
- Store set of rectangles that intersect the sweep line in an interval search tree (using y-interval of rectangle).
- Left side: interval search for y-interval of rectangle, insert y-interval.
- Right side: delete y-interval.



VLSI Rules checking: Sweep-line algorithm (summary)

Reduces 2D orthogonal rectangle intersection search to 1D interval search!

Running time of sweep line algorithm.

• Sort by x-coordinate.	O(N log N)
• Insert y-interval into ST.	O(N log N)
• Delete y-interval from ST.	O(N log N)
• Interval search.	O(R log N)

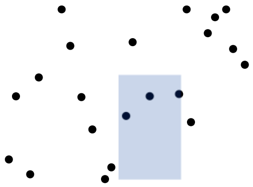
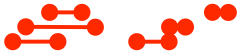
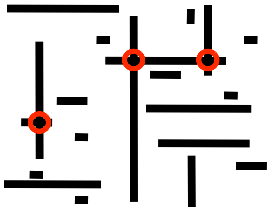
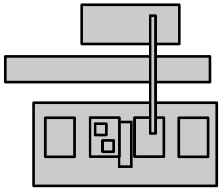
N = # line segments
R = # intersections

Efficiency relies on judicious extension of BST.

Bottom line.

Linearithmic algorithm enables design-rules checking for huge problems

Geometric search summary: Algorithms of the day

1D range search		BST
kD range search		kD tree
1D interval intersection search		interval tree
2D orthogonal line intersection search		sweep line reduces to 1D range search
2D orthogonal rectangle intersection search		sweep line reduces to 1D interval intersection search

Radix Sorts

- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- application: LRS

References:

Algorithms in Java, Chapter 10
http://www.cs.princeton.edu/introalgsds/61sort

Review: summary of the performance of sorting algorithms

Frequency of execution of instructions in the inner loop:

algorithm	guarantee	average	extra space	operations on keys
insertion sort	$N^2 /2$	$N^2 /4$	no	`compareTo()`
selection sort	$N^2 /2$	$N^2 /2$	no	`compareTo()`
mergesort	N lg N	N lg N	N	`compareTo()`
quicksort	1.39 N lg N	1.39 N lg N	c lg N	`compareTo()`

lower bound: N lg N -1.44 N compares are required by any algorithm

Q: Can we do better (despite the lower bound)?

Digital keys

Many commonly-use key types are inherently digital (sequences of fixed-length characters)

example interface

```
interface Digital
{
  public int charAt(int k);
  public int length(int);
}
```

Examples

- Strings
- 64-bit integers

This lecture:

- refer to fixed-length vs. variable-length strings
- **R** different characters for some fixed value **R**.
- assume key type implements **`charAt()`** and **`length()`** methods
- code works for **`String`**

Widely used in practice

- low-level bit-based sorts
- string sorts

- **key-indexed counting**
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- application: LRS

Key-indexed counting: assumptions about keys

Assume that keys are integers between **0** and **R-1**
Implication: Can use key as an array index

Examples:

- **char** (**R = 256**)
- **short** with fixed **R**, enforced by client
- **int** with fixed **R**, enforced by client

Reminder: equal keys are not uncommon in sort applications

Applications:

- sort phone numbers by area code
- sort classlist by precept
- Requirement: sort must be stable
- Ex: Full sort on primary key, then stable radix sort on secondary key

Key-indexed counting

Task: sort an array `a[]` of N integers between 0 and R-1

Plan: produce sorted result in array `temp[]`

1. Count frequencies of each letter using key as index
2. Compute frequency cumulates
3. Access cumulates using key as index to find record positions.
4. Copy back into original array

```
int N = a.length;
int[] count = new int[R];
```

count frequencies →

```
for (int i = 0; i < N; i++)
   count[a[i]+1]++;
```

compute cumulates →

```
for (int k = 1; k < 256; k++)
   count[k] += count[k-1];
```

move records →

```
for (int i = 0; i < N; i++)
   temp[count[a[i]++]] = a[i]
```

copy back →

```
for (int i = 0; i < N; i++)
   a[i] = temp[i];
```

	a[]	
0	a	
1	a	
2	b	
3	b	
4	b	
5	c	
6	d	
7	d	
8	e	
9	f	
10	f	
11	f	

count[]	
a	2
b	5
c	6
d	8
e	9
f	12

	temp[]	
0	a	
1	a	
2	b	
3	b	
4	b	
5	c	
6	d	
7	d	
8	e	
9	f	
10	f	
11	f	

Review: summary of the performance of sorting algorithms

Frequency of execution of instructions in the inner loop:

algorithm	guarantee	average	extra space	operations on keys
insertion sort	$N^2 /2$	$N^2 /4$	no	`compareTo()`
selection sort	$N^2 /2$	$N^2 /2$	no	`compareTo()`
mergesort	N lg N	N lg N	N	`compareTo()`
quicksort	1.39 N lg N	1.39 N lg N	c lg N	`compareTo()`
key-indexed counting	N + R	N + R	N + R	use as array index

inplace version is possible and practical

Q: Can we do better (despite the lower bound)?

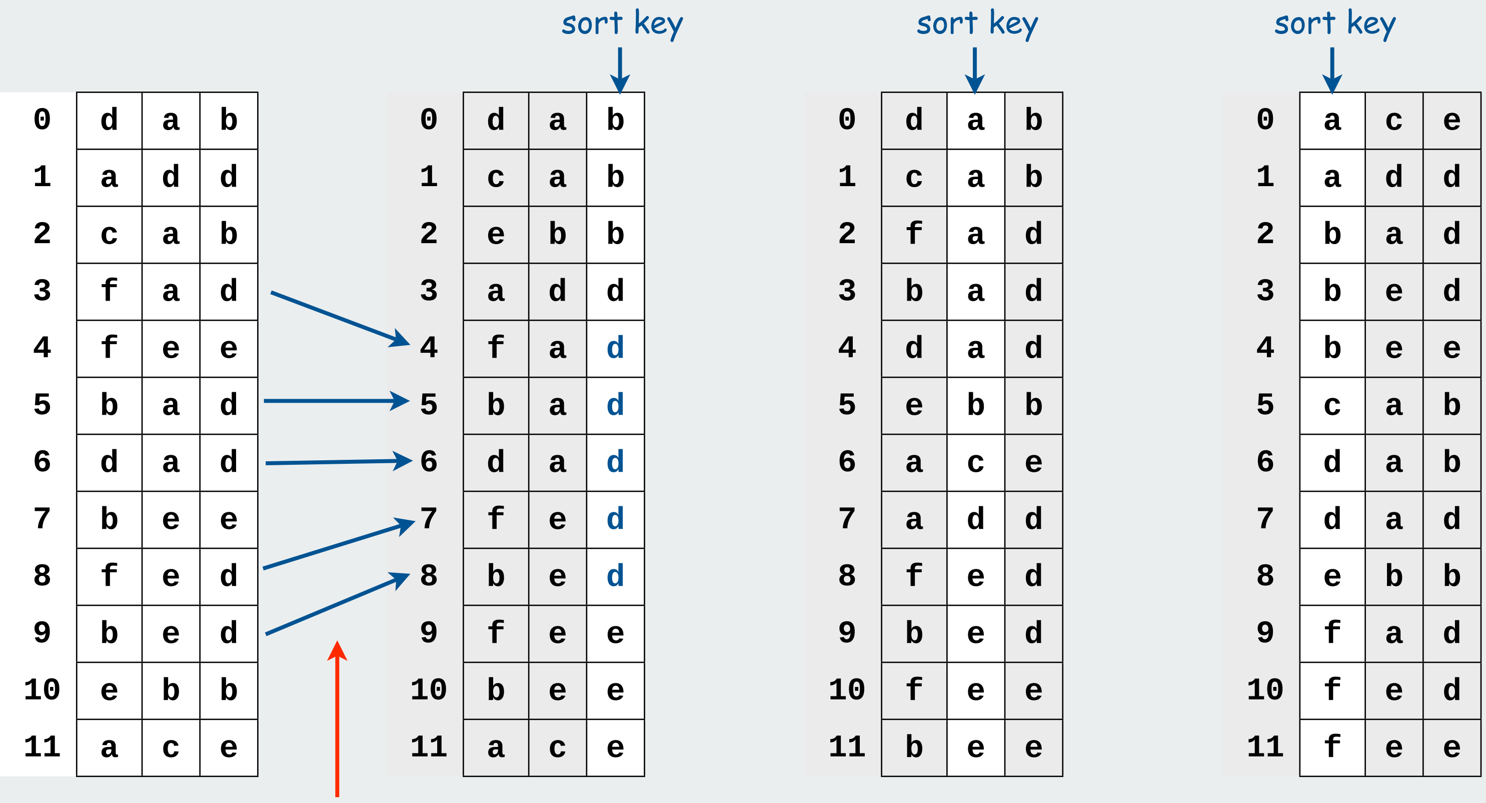
A: Yes, if we do not depend on comparisons

- key-indexed counting
- **LSD radix sort**
- MSD radix sort
- 3-way radix quicksort
- application: LRS

Least-significant-digit-first radix sort

LSD radix sort.

- Consider characters d from right to left
- Stably sort using dth character as the key via key-indexed counting.



0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

0	d	a	b
1	c	a	b
2	e	b	b
3	a	d	d
4	f	a	d
5	b	a	d
6	d	a	d
7	f	e	d
8	b	e	d
9	f	e	e
10	b	e	e
11	a	c	e

0	d	a	b
1	c	a	b
2	f	a	d
3	b	a	d
4	d	a	d
5	e	b	b
6	a	c	e
7	a	d	d
8	f	e	d
9	b	e	d
10	f	e	e
11	b	e	e

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	d
11	f	e	e

sort must be stable
arrows do not cross

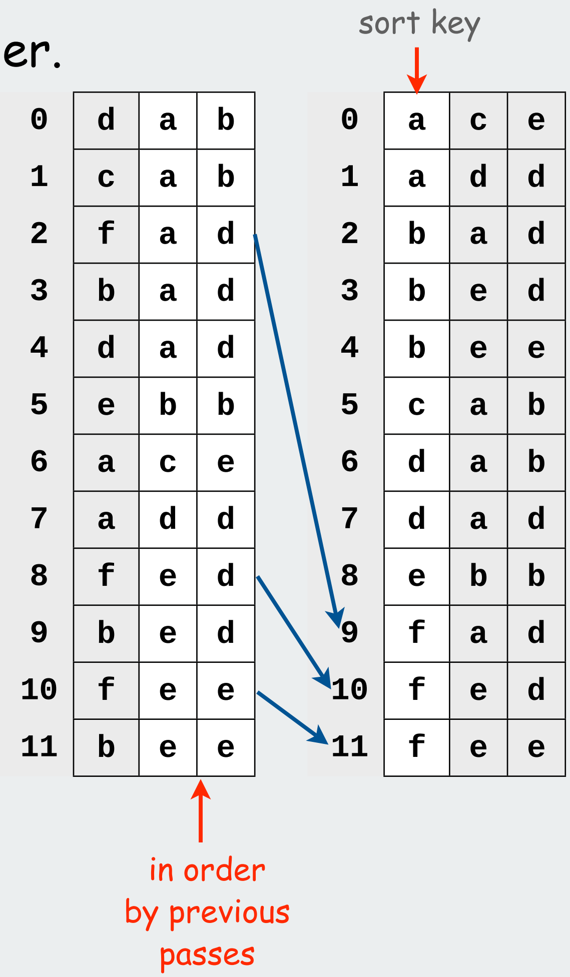
LSD radix sort: Why does it work?

Pf 1. [thinking about the past]

- If two strings differ on first character, key-indexed sort puts them in proper relative order.
- If two strings agree on first character, stability keeps them in proper relative order.

Pf 2. [thinking about the future]

- If the characters not yet examined differ, it doesn't matter what we do now.
- If the characters not yet examined agree, stability ensures later pass won't affect order.



sort key

0	d	a	b
1	c	a	b
2	f	a	d
3	b	a	d
4	d	a	d
5	e	b	b
6	a	c	e
7	a	d	d
8	f	e	d
9	b	e	d
10	f	e	e
11	b	e	e

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	d
11	f	e	e

in order by previous passes

LSD radix sort implementation

Use k-indexed counting on characters, moving right to left

```
public static void lsd(String[] a)
{
   int N = a.length;
   int W = a[0].length;
   for (int d = W-1; d >= 0; d--)
   {
      int[] count = new int[R];
      for (int i = 0; i < N; i++)
         count[a[i].charAt(d) + 1]++;
      for (int k = 1; k < 256; k++)
         count[k] += count[k-1];
      for (int i = 0; i < N; i++)
         temp[count[a[i].charAt(d)]++] = a[i];
      for (int i = 0; i < N; i++)
         a[i] = temp[i];
   }
}
```

key-indexed counting

count frequencies

compute cumulates

move records

copy back

Assumes fixed-length keys (length = W)

Review: summary of the performance of sorting algorithms

Frequency of execution of instructions in the inner loop:

algorithm	guarantee	average	extra space	assumptions on keys
insertion sort	$N^2/2$	$N^2/4$	no	Comparable
selection sort	$N^2/2$	$N^2/2$	no	Comparable
mergesort	N lg N	N lg N	N	Comparable
quicksort	1.39 N lg N	1.39 N lg N	c lg N	Comparable
LSD radix sort	WN	WN	N + R	digital

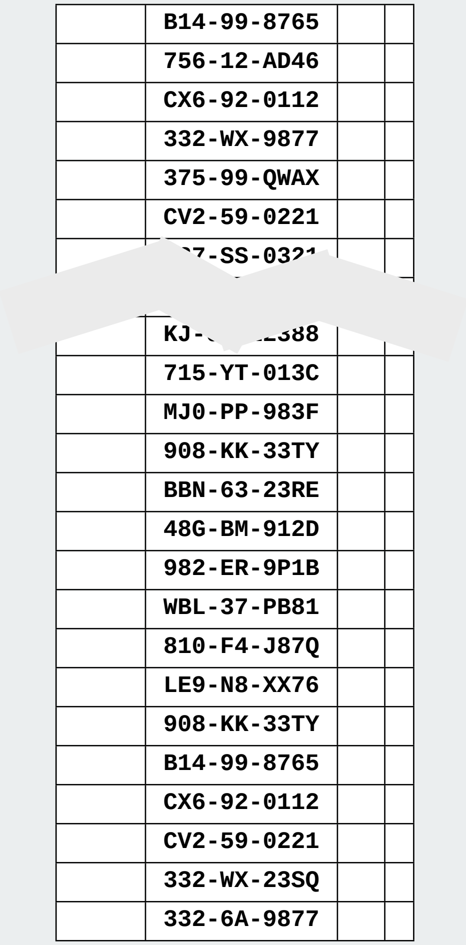
Sorting Challenge

Problem: sort a huge commercial database on a fixed-length key field

Ex: account number, date, SS number

Which sorting method to use?

1. insertion sort
2. mergesort
3. quicksort
4. LSD radix sort

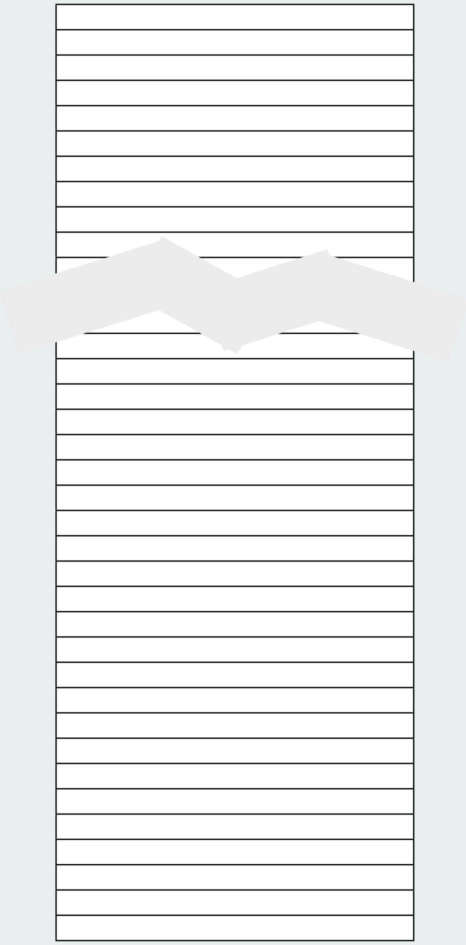


	B14-99-8765		
	756-12-AD46		
	CX6-92-0112		
	332-WX-9877		
	375-99-QWAX		
	CV2-59-0221		
	[illegible]7-SS-0321		
	KJ-[illegible]2388		
	715-YT-013C		
	MJ0-PP-983F		
	908-KK-33TY		
	BBN-63-23RE		
	48G-BM-912D		
	982-ER-9P1B		
	WBL-37-PB81		
	810-F4-J87Q		
	LE9-N8-XX76		
	908-KK-33TY		
	B14-99-8765		
	CX6-92-0112		
	CV2-59-0221		
	332-WX-23SQ		
	332-6A-9877		

Sorting Challenge

Problem: sort huge files of random 128-bit numbers

Ex: supercomputer sort, internet router



Which sorting method to use?

1. insertion sort
2. mergesort
3. quicksort
4. LSD radix sort

LSD radix sort: a moment in history (1960s)

card punch

punched cards

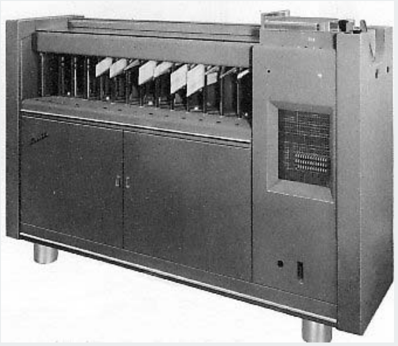
card reader

mainframe

line printer

To sort a card deck

1. start on right column
2. put cards into hopper
3. machine distributes into bins
4. pick up cards (stable)
5. move left one column
6. continue until sorted

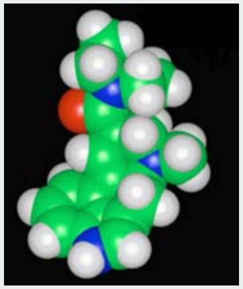


card sorter

LSD not related to sorting

↓

"Lucy in the Sky with Diamonds"



Lysergic Acid Diethylamide

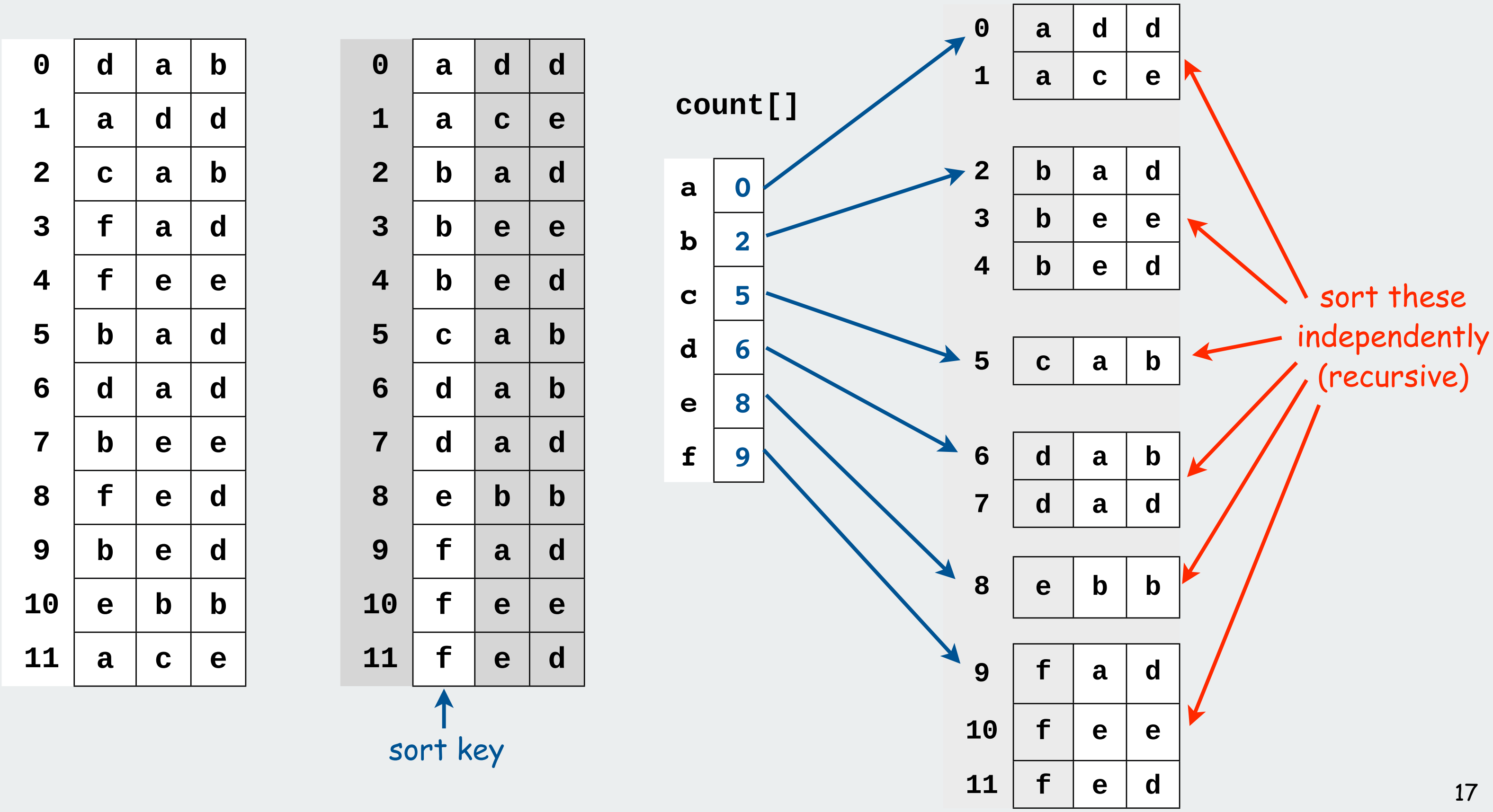
LSD radix sort actually predates computers

- key-indexed counting
- LSD radix sort
- **MSD radix sort**
- 3-way radix quicksort
- application: LRS

MSD Radix Sort

Most-significant-digit-first radix sort.

- Partition file into R pieces according to first character (use key-indexed counting)
- Recursively sort all strings that start with each character (key-indexed counts delineate files to sort)



0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

sort key

count[]

a	0
b	2
c	5
d	6
e	8
f	9

0	a	d	d
1	a	c	e

2	b	a	d
3	b	e	e
4	b	e	d

5	c	a	b

6	d	a	b
7	d	a	d

8	e	b	b

9	f	a	d
10	f	e	e
11	f	e	d

sort these independently (recursive)

MSD radix sort implementation

Use key-indexed counting on first character, recursively sort subfiles

```
public static void msd(String[] a)
{  msd(a, 0, a.length, 0);  }

private static void msd(String[] a, int lo, int hi, int d)
{
   if (hi <= lo + 1) return;
   int[] count = new int[256+1];
   for (int i = 0; i < N; i++)
      count[a[i].charAt(d) + 1]++;
   for (int k = 1; k < 256; k++)
      count[k] += count[k-1];
   for (int i = 0; i < N; i++)
      temp[count[a[i].charAt(d)]++] = a[i];
   for (int i = 0; i < N; i++)
      a[i] = temp[i];
   for (int i = 0; i < 255; i++)
      msd(a, l + count[i], l + count[i+1], d+1);
}
```

key-indexed counting →

← count frequencies

← compute cumulates

← move records

← copy back

MSD radix sort: potential for disastrous performance

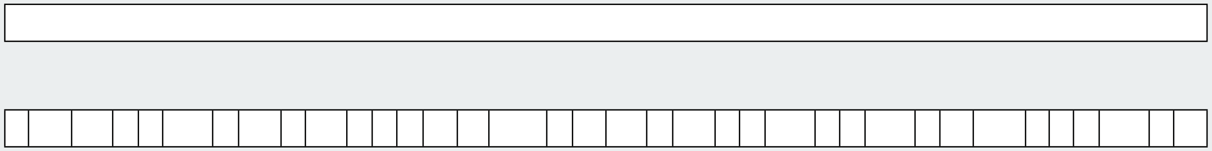
Observation 1: Much too slow for small files

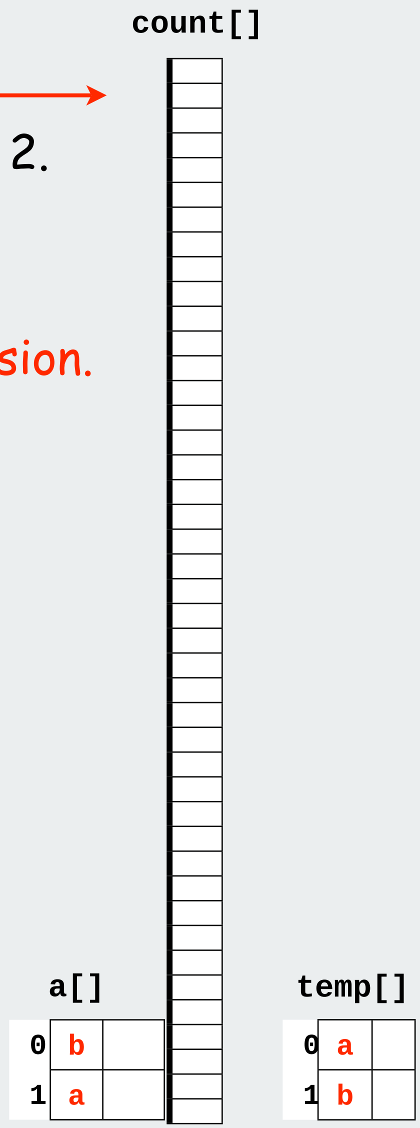
- all counts must be initialized to zero
- ASCII (256 counts): 100x slower than copy pass for N = 2.
- Unicode (65536 counts): 30,000x slower for N = 2

Observation 2: Huge number of small files because of recursion.

- keys all different: up to N/2 files of size 2
- ASCII: 100x slower than copy pass for all N.
- Unicode: 30,000x slower for all N

switch to Unicode might be a big surprise!





a[]	
0	b
1	a

temp[]	
0	a
1	b

Solution. Switch to insertion sort for small N.

MSD radix sort bonuses

Bonus 1: May not have to examine all of the keys.

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d

← 19/24 ≈ 80% of the characters examined

Bonus 2: Works for variable-length keys (**`String`** values)

0	a	c	e	t	o	n	e	\0		
1	a	d	d	i	t	i	o	n	\0	
2	b	a	d	g	e	\0				
3	b	e	d	a	z	z	l	e	d	\0
4	b	e	e	h	i	v	e	\0		
5	c	a	b	i	n	e	t	r	y	\0
6	d	a	b	b	l	e	\0			
7	d	a	d	\0						

← 19/64 ≈ 30% of the characters examined

Implication: sublinear sorts (!)

MSD string sort implementation

Use key-indexed counting on first character, recursively sort subfiles

```
public static void msd(String[] a)
{  msd(a. 0. a.length, 0);

private static void msd(String[] a, int l, int r, int d)
{
   if (r <= l + 1) return;
   int[] count = new int[256];
   for (int i = 0; i < N; i++)
      count[a[i].charAt(d) + 1]++;
   for (int k = 1; k < 256; k++)
      count[k] += count[k-1];
   for (int i = 0; i < N; i++)
      temp[count[a[i].charAt(d)]++] = a[i];
   for (int i = 0; i < N; i++)
      a[i] = temp[i];
   for (int i = 1; i < 255; i++)
      msd(a, l + count[i], l + count[i+1], d+1);
}
```

key-indexed counting

don't sort strings that start with '\0' (end of string char)

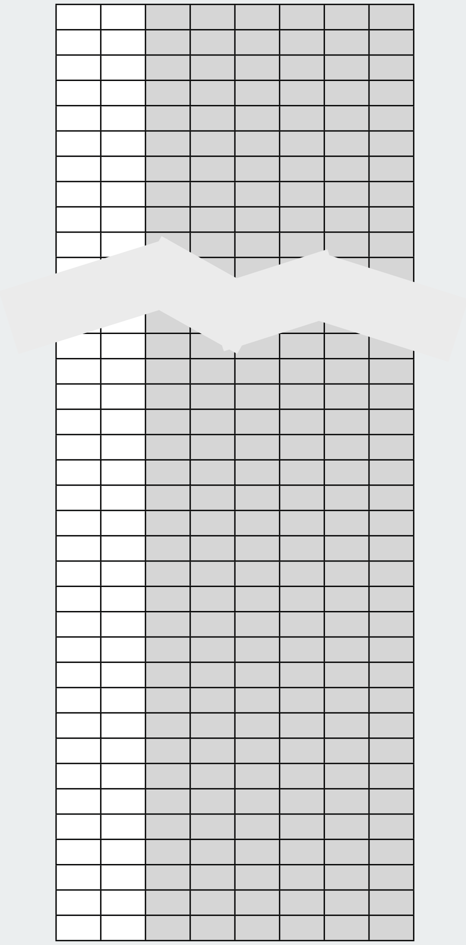
Sorting Challenge (revisited)

Problem: sort huge files of random 128-bit numbers

Ex: supercomputer sort, internet router

Which sorting method to use?

1. insertion sort
2. mergesort
3. quicksort
4. ✓ LSD radix sort on MSDs



2^{16} = 65536 counters

divide each word into 16-bit "chars"

sort on leading 32 bits in 2 passes

finish with insertion sort

examines only ~25% of the data

MSD radix sort versus quicksort for strings

Disadvantages of MSD radix sort.

- Accesses memory "randomly" (cache inefficient)
- Inner loop has a lot of instructions.
- Extra space for counters.
- Extra space for temp (or complicated inplace key-indexed counting).

Disadvantage of quicksort.

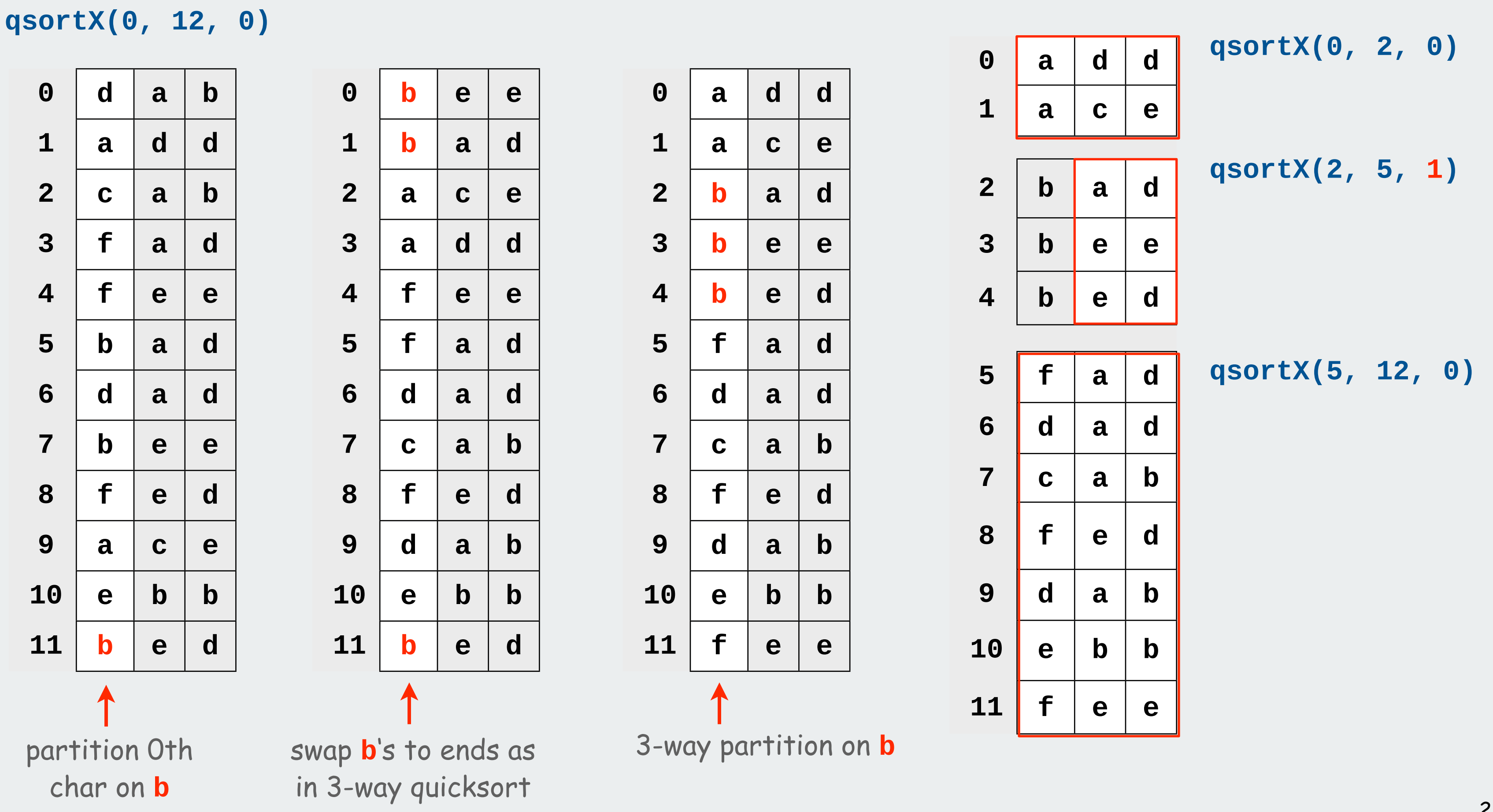
- N lg N, not linear.
- Has to rescan long keys for compares
- [but stay tuned]

- key-indexed counting
- LSD radix sort
- MSD radix sort
- **3-way radix quicksort**
- application: LRS

3-Way radix quicksort (Bentley and Sedgewick, 1997)

Idea. Do 3-way partitioning on the dth character.

- cheaper than R-way partitioning of MSD radix sort
- need not examine again chars equal to the partitioning char



qsortX(0, 12, 0)

0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	a	c	e
10	e	b	b
11	b	e	d

partition 0th char on b

0	b	e	e
1	b	a	d
2	a	c	e
3	a	d	d
4	f	e	e
5	f	a	d
6	d	a	d
7	c	a	b
8	f	e	d
9	d	a	b
10	e	b	b
11	b	e	d

swap b's to ends as in 3-way quicksort

0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	f	a	d
6	d	a	d
7	c	a	b
8	f	e	d
9	d	a	b
10	e	b	b
11	f	e	e

3-way partition on b

0	a	d	d
1	a	c	e

qsortX(0, 2, 0)

2	b	a	d
3	b	e	e
4	b	e	d

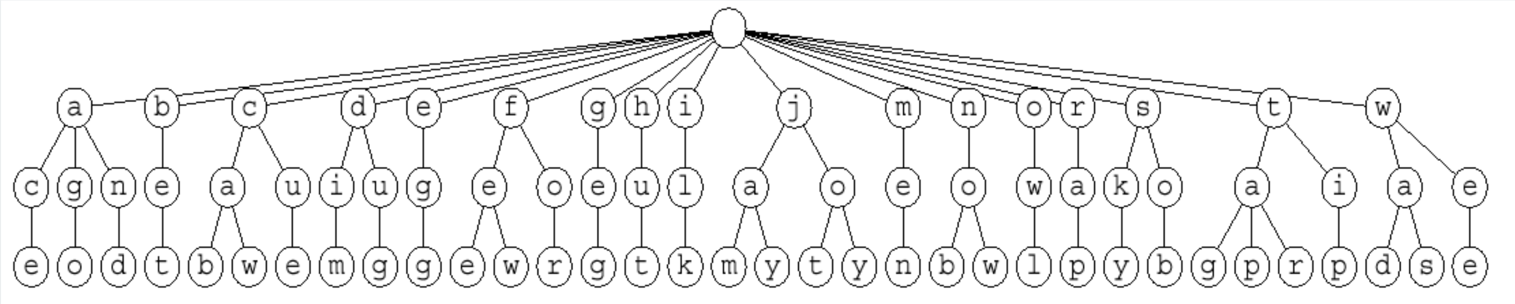
qsortX(2, 5, 1)

5	f	a	d
6	d	a	d
7	c	a	b
8	f	e	d
9	d	a	b
10	e	b	b
11	f	e	e

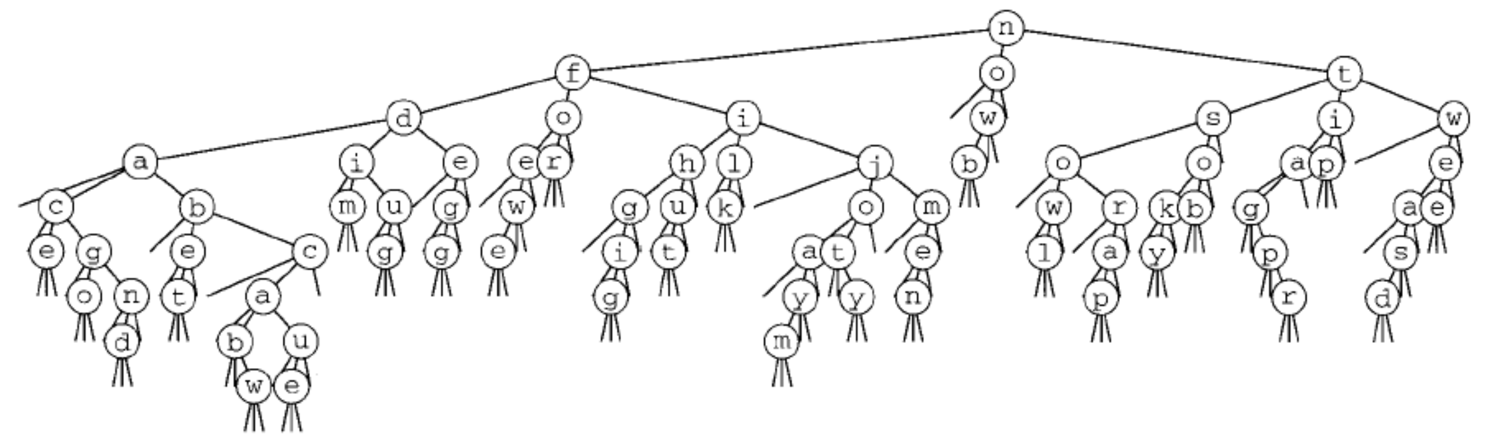
qsortX(5, 12, 0)

Recursive structure: MSD radix sort vs. 3-Way radix quicksort

3-way radix quicksort collapses empty links in MSD recursion tree.


a b c d e f g h i j m n o r s t w
c g n e a u i u g e o e u l a o e o w a k o a i a e
e o d t b w e m g g e w r g t k m y t y n b w l p y b g p r p d s e

MSD radix sort recursion tree
(1035 null links, not shown)


n
f o t
d o i w s i w
a i e e r h l j b o o a p e
c b m u g w g u k o m w r k b g a e
e g e c g g e i t a t e l a y p s
o n t a g y y n p r d
d b u m
w e

3-way radix quicksort recursion tree
(155 null links)

3-Way radix quicksort

```
private static void quicksortX(String a[], int lo, int hi, int d)
{
   if (hi - lo <= 0) return;
   int i = lo-1, j = hi;
   int p = lo-1, q = hi;
   char v = a[hi].charAt(d);
   while (i < j)
   {
      while (a[++i].charAt(d) < v) if (i == hi) break;
      while (v < a[--j].charAt(d)) if (j == lo) break;
      if (i > j) break;
      exch(a, i, j);
      if (a[i].charAt(d) == v) exch(a, ++p, i);
      if (a[j].charAt(d) == v) exch(a, j, --q);
   }

   if (p == q)
   {
      if (v != '\0') quicksortX(a, lo, hi, d+1);
      return;
   }

   if (a[i].charAt(d) < v) i++;
   for (int k = lo; k <= p; k++) exch(a, k, j--);
   for (int k = hi; k >= q; k--) exch(a, k, i++);

   quicksortX(a, lo, j, d);
   if ((i == hi) && (a[i].charAt(d) == v)) i++;
   if (v != '\0') quicksortX(a, j+1, i-1, d+1);
   quicksortX(a, i, hi, d);
}
```

← 4-way partition with equals at ends

← special case for all equals

← swap equals back to middle

← sort 3 pieces recursively

3-Way Radix quicksort vs. standard quicksort

standard quicksort.

- uses 2N ln N string comparisons on average.
- uses costly compares for long keys that differ only at the end, and this is a common case!

3-way radix quicksort.

- avoids re-comparing initial parts of the string.
- adapts to data: uses just "enough" characters to resolve order.
- uses 2 N ln N character comparisons on average for random strings.
- is sub-linear when strings are long

to within a constant factor

Theorem. Quicksort with 3-way partitioning is OPTIMAL.
No sorting algorithm can examine fewer chars on any input

asymptotically

Pf. Ties cost to entropy. Beyond scope of 226.

3-Way Radix quicksort vs. MSD radix sort

MSD radix sort

- has a long inner loop
- is cache-inefficient
- repeatedly initializes counters for long stretches of equal chars, and this is a common case!

Ex. Library call numbers

```
WUS-------10706-----7---10
WUS-------12692-----4---27
WLSOC------2542----30
LTK--6015-P-63-1988
LDS---361-H-4
 ...
```

3-way radix quicksort

- uses one compare for equal chars.
- is cache-friendly
- adapts to data: uses just "enough" characters to resolve order.

3-way radix quicksort is the method of choice for sorting strings

- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- **application: LRS**

Longest repeated substring

Given a string of N characters, find the longest repeated substring.

Ex:

```
a a c a a g t t t a c a a g c a t g a t g c t g t a c t a
g g a g a g t t a t a c t g g t c g t c a a a c c t g a a
c c t a a t c c t t g t g t g t a c a c a c a c t a c t a
c t g t c g t c g t c a t a t a t c g a g a t c a t c g a
a c c g g a a g g c c g g a c a a g g c g g g g g g t a t
a g a t a g a t a g a c c c c t a g a t a c a c a t a c a
t a g a t c t a g c t a g c t a g c t c a t c g a t a c a
c a c t c t c a c a c t c a a g a g t t a t a c t g g t c
a a c a c a c t a c t a c g a c a g a c g a c c a a c c a
g a c a g a a a a a a a a c t c t a t a t c t a t a a a a
```

Longest repeated substring

Given a string of N characters, find the longest repeated substring.

Ex:

a a c a a g t t t a c a a g c a t g a t g c t g t a c t a
g g a g a g t t a t a c t g g t c g t c a a a c c t g a a
c c t a a t c c t t g t g t g t a c a c a c a c t a c t a
c t g t c g t c g t c a t a t a t c g a g a t c a t c g a
a c c g g a a g g c c g g a c a a g g c g g g g g g t a t
a g a t a g a t a g a c c c c t a g a t a c a c a t a c a
t a g a t c t a g c t a g c t a g c t c a t c g a t a c a
c a c t c t c a c a c t c a a g a g t t a t a c t g g t c
a a c a c a c t a c t a c g a c a g a c g a c c a a c c a
g a c a g a a a a a a a a c t c t a t a t c t a t a a a a

String processing

String. Sequence of characters.

Important fundamental abstraction

Natural languages, Java programs, genomic sequences, …

> The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology. -M. V. Olson

Using Strings in Java

String concatenation: append one string to end of another string.

Substring: extract a contiguous list of characters from a string.

s	t	r	i	n	g	s
0	1	2	3	4	5	6

```
String s = "strings";          // s = "strings"
char   c = s.charAt(2);        // c = 'r'
String t = s.substring(2, 6);  // t = "ring"
String u = s + t;              // u = "stringsring"
```

Implementing Strings In Java

Memory. 40 + 2N bytes for a virgin string!

could use byte array instead of String to save space

```
public final class String implements Comparable<String>
{
   private char[] value;  // characters
   private int offset;    // index of first char into array
   private int count;     // length of string
   private int hash;      // cache of hashCode()

   private String(int offset, int count, char[] value)
   {
       this.offset = offset;
       this.count  = count;
       this.value  = value;
   }
   public String substring(int from, int to)
   {
      return new String(offset + from, to - from, value);  }
   …
}
```

java.lang.String

String vs. StringBuilder

`String`. [immutable] Fast substring, slow concatenation.

`StringBuilder`. [mutable] Slow substring, fast (amortized) append.

Ex. Reverse a string

```
public static String reverse(String s)
{
   String rev = "";
   for (int i = s.length() - 1; i >= 0; i--)
         rev += s.charAt(i);
   return rev;
}
```

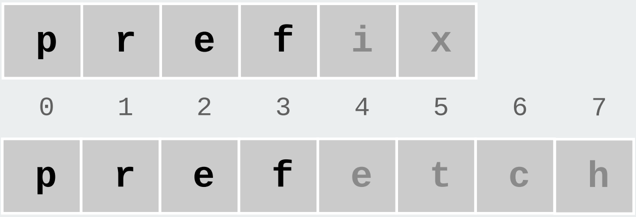
quadratic time

```
public static String reverse(String s)
{
   StringBuilder rev = new StringBuilder();
   for (int i = s.length() - 1; i >= 0; i--)
      rev.append(s.charAt(i));
   return rev.toString();
}
```

linear time

Warmup: longest common prefix

Given two strings, find the longest substring that is a prefix of both



```
public static String lcp(String s, String t)
{
   int n = Math.min(s.length(), t.length());
   for (int i = 0; i < n; i++)
   {
      if (s.charAt(i) != t.charAt(i))
         return s.substring(0, i);
   }
   return s.substring(0, n);
}
```

linear time

Would be quadratic with **StringBuilder**

Lesson: cost depends on implementation

This lecture: need constant-time **substring()**, use **String**

Longest repeated substring

Given a string of N characters, find the longest repeated substring.

Classic string-processing problem.

Ex: `a a c a a g t t t a c a a g c`

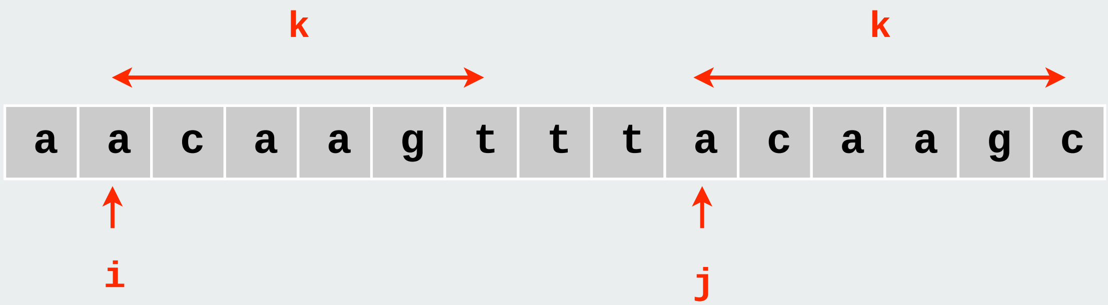
`1` `9`

Applications

- bioinformatics.
- cryptanalysis.

Brute force.

- Try all indices `i` and `j` for start of possible match, and check.
- Time proportional to M N^2 , where M is length of longest match.



Longest repeated substring

Suffix sort solution.

- form N suffixes of original string.
- sort to bring longest repeated substrings together.
- check LCP of adjacent substrings to find longest match

suffixes

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
3	a	a	g	t	t	t	a	c	a	a	g	c			
4	a	g	t	t	t	a	c	a	a	g	c				
5	g	t	t	t	a	c	a	a	g	c					
6	t	t	t	a	c	a	a	g	c						
7	t	t	a	c	a	a	g	c							
8	t	a	c	a	a	g	c								
9	a	c	a	a	g	c									
10	c	a	a	g	c										
11	a	a	g	c											
12	a	g	c												
13	g	c													
14	c														

sorted suffixes

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
11	a	a	g	c											
3	a	a	g	t	t	t	a	c	a	a	g	c			
▶ 9	a	c	a	a	g	c									
▶ 1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
12	a	g	c												
4	a	g	t	t	t	a	c	a	a	g	c				
14	c														
10	c	a	a	g	c										
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
13	g	c													
5	g	t	t	t	a	c	a	a	g	c					
8	t	a	c	a	a	g	c								
7	t	t	a	c	a	a	g	c							
6	t	t	t	a	c	a	a	g	c						

Suffix Sorting: Java Implementation

```
public class LRS {
   public static void main(String[] args) {
      String s = StdIn.readAll();
      int N = s.length();

      String[] suffixes = new String[N];
      for (int i = 0; i < N; i++)
         suffixes[i] = s.substring(i, N);

      Arrays.sort(suffixes);

      String lrs = "";
      for (int i = 0; i < N - 1; i++) {
         String x = lcp(suffixes[i], suffixes[i+1]);
         if (x.length() > lrs.length()) lrs = x;
      }
      System.out.println(lrs);
   }
}
```

read input

create suffixes
(linear time)

sort suffixes

find LCP

```
% java LRS < mobydick.txt
,- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th
```

Sorting Challenge

Problem: suffix sort a long string

Ex. Moby Dick ~1.2 million chars

Which sorting method to use?

1. insertion sort
2. mergesort
3. quicksort
4. LSD radix sort
5. MSD radix sort
6. ✓ 3-way radix quicksort

only if LRS is not long (!)

Suffix sort experimental results

algorithm	time to suffix-sort Moby Dick (seconds)
brute-force	36.000 (est.)
quicksort	9.5
LSD	not fixed-length
MSD	395
MSD with cutoff	6.8
3-way radix quicksort	2.8

Suffix Sorting: Worst-case input

Longest match not long:

- hard to beat 3-way radix quicksort.

Longest match very long:

- radix sorts are quadratic in the length of the longest match
- Ex: two copies of Moby Dick.

Can we do better? linearithmic? linear?

Observation. Must find longest repeated substring while suffix sorting to beat N^2.

```
abcdefghi
abcdefghiabcdefghi
bcdefghi
bcdefghiabcdefghi
cdefghi
cdefghiabcdefgh
defghi
efghiabcdefghi
efghi
fghiabcdefghi
fghi
ghiabcdefghi
fhi
hiabcdefghi
hi
iabcdefghi
i
```

Input: "abcdeghiabcdefghi"

Fast suffix sorting

Manber's MSD algorithm

- phase 0: sort on first character using key-indexed sort.
- phase i: given list of suffixes sorted on first 2^{i-1} characters, create list of suffixes sorted on first 2^{i} characters

Running time

- finishes after lg N phases
- obvious upper bound on growth of total time: O(N (lg N)^2)
- actual growth of total time (proof omitted): ~N lg N.

not many subfiles if not much repetition
3-way quicksort handles equal keys if repetition

Best algorithm in theory is linear (but more complicated to implement).

Linearithmic suffix sort example: phase 0

		index sort		inverse	
0	babaaaabcbabaaaaa0	17	0	0	12
1	abaaaabcbabaaaaa0	1	abaaaabcbabaaaaa0	1	1
2	baaaabcbabaaaaa0	16	a0	2	16
3	aaaabcbabaaaaa0	3	aaaabcbabaaaaa0	3	3
4	aaabcbabaaaaa0	4	aaabcbabaaaaa0	4	4
5	aabcbabaaaaa0	5	aabcbabaaaaa0	5	5
6	abcbabaaaaa0	6	abcbabaaaaa0	6	6
7	bcbabaaaaa0	15	aa0	7	15
8	cbabaaaaa0	14	aaa0	8	17
9	babaaaaa0	13	aaaa0	9	13
10	abaaaaa0	12	aaaaa0	10	11
11	baaaaa0	10	abaaaaa0	11	14
12	aaaaa0	0	babaaaabcbabaaaaa0	12	10
13	aaaa0	9	babaaaaa0	13	9
14	aaa0	11	baaaaa0	14	8
15	aa0	7	bcbabaaaaa0	15	7
16	a0	2	baaaabcbabaaaaa0	16	2
17	0	8	cbabaaaaa0	17	0

↑
sorted

Linearithmic suffix sort example: phase 1

0	babaaaabcbabaaaaa0
1	abaaaabcbabaaaaa0
2	baaaabcbabaaaaa0
3	aaaabcbabaaaaa0
4	aaabcbabaaaaa0
5	aabcbabaaaaa0
6	abcbabaaaaa0
7	bcbabaaaaa0
8	cbabaaaaa0
9	babaaaaa0
10	abaaaaa0
11	baaaaa0
12	aaaaa0
13	aaaa0
14	aaa0
15	aa0
16	a0
17	0

index sort

17	0
16	a0
12	aaaaa0
3	aaaabcbabaaaaa0
4	aaabcbabaaaaa0
5	aabcbabaaaaa0
13	aaaa0
15	aa0
14	aaa0
6	abcbabaaaaa0
1	abaaaabcbabaaaaa0
10	abaaaaa0
0	babaaaabcbabaaaaa0
9	babaaaaa0
11	baaaaa0
2	baaaabcbabaaaaa0
7	bcbabaaaaa0
8	cbabaaaaa0

↑
sorted

inverse

0	12
1	10
2	15
3	3
4	4
5	5
6	9
7	16
8	17
9	13
10	11
11	14
12	2
13	6
14	8
15	7
16	1
17	0

Linearithmic suffix sort example: phase 2

0	babaaaabcbabaaaaa0
1	abaaaabcbabaaaaa0
2	baaaabcbabaaaaa0
3	aaaabcbabaaaaa0
4	aaabcbabaaaaa0
5	aabcbabaaaaa0
6	abcbabaaaaa0
7	bcbabaaaaa0
8	cbabaaaaa0
9	babaaaaa0
10	abaaaaa0
11	baaaaa0
12	aaaaa0
13	aaaa0
14	aaa0
15	aa0
16	a0
17	0

index sort

17	0
16	a0
15	aa0
14	aaa0
3	aaaabcbabaaaaa0
12	aaaaa0
13	aaaa0
4	aaabcbabaaaaa0
5	aabcbabaaaaa0
1	abaaaabcbabaaaaa0
10	abaaaaa0
6	abcbabaaaaa0
2	baaaabcbabaaaaa0
11	baaaaa0
0	babaaaabcbabaaaaa0
9	babaaaaa0
7	bcbabaaaaa0
8	cbabaaaaa0

sorted

inverse

0	14
1	9
2	12
3	4
4	7
5	8
6	11
7	16
8	17
9	15
10	10
11	13
12	5
13	6
14	3
15	2
16	1
17	0

Linearithmic suffix sort example: phase 3

0	babaaaabcbabaaaaa0
1	abaaaabcbabaaaaa0
2	baaaabcbabaaaaa0
3	aaaabcbabaaaaa0
4	aaabcbabaaaaa0
5	aabcbabaaaaa0
6	abcbabaaaaa0
7	bcbabaaaaa0
8	cbabaaaaa0
9	babaaaaa0
10	abaaaaa0
11	baaaaa0
12	aaaaa0
13	aaaa0
14	aaa0
15	aa0
16	a0
17	0

index sort

17	0
16	a0
15	aa0
14	aaa0
3	aaaabcbabaaaaa0
13	aaaa0
12	aaaaa0
4	aaabcbabaaaaa0
5	aabcbabaaaaa0
10	abaaaaa0
1	abaaaabcbabaaaaa0
6	abcbabaaaaa0
11	baaaaa0
2	baaaabcbabaaaaa0
9	babaaaaa0
0	babaaaabcbabaaaaa0
7	bcbabaaaaa0
8	cbabaaaaa0

sorted

inverse

0	15
1	10
2	13
3	4
4	7
5	8
6	11
7	16
8	17
9	14
10	9
11	12
12	6
13	5
14	3
15	2
16	1
17	0

FINISHED! (no equal keys)

Linearithmic suffix sort: key idea

Achieve constant-time string compare by indexing into inverse

		index sort		inverse	
0	babaaaabcbabaaaaa0	17	0	0	14
1	abaaaabcbabaaaaa0	16	a0	1	9
2	baaaabcbabaaaaa0	15	aa0	2	12
3	aaaabcbabaaaaa0	14	aaa0	3	4
4	aaabcbabaaaaa0	3	aaaabcbabaaaaa0	4	7
5	aabcbabaaaaa0	12	aaaaa0	5	8
6	abcbabaaaaa0	13	aaaa0	6	11
7	bcbabaaaaa0	4	aaabcbabaaaaa0	7	16
8	cbabaaaaa0	5	aabcbabaaaaa0	8	17
9	babaaaaa0	1	abaaaabcbabaaaaa0	9	15
10	abaaaaa0	10	abaaaaa0	10	10
11	baaaaa0	6	abcbabaaaaa0	11	13
12	aaaaa0	2	baaaabcbabaaaaa0	12	5
13	aaaa0	11	baaaaa0	13	6
14	aaa0	0	babaaaabcbabaaaaa0	14	3
15	aa0	9	babaaaaa0	15	2
16	a0	7	bcbabaaaaa0	16	1
17	0	8	cbabaaaaa0	17	0

0 + 4 = 4

9 + 4 = 13

13 < 4 (because 6 < 7) so 9 < 0

Suffix sort experimental results

algorithm	time to suffix-sort Moby Dick (seconds)	time to suffix-sort AesopAesop (seconds)
brute-force	36.000 (est.)	4000 (est.)
quicksort	9.5	167
MSD	395	out of memory
MSD with cutoff	6.8	162
3-way radix quicksort	2.8	400
Manber MSD	17	8.5

counters in deep recursion

only 2 keys in subfiles with long matches

Radix sort summary

We can develop linear-time sorts.

- comparisons not necessary for some types of keys
- use keys to index an array

We can develop sub-linear-time sorts.

- should measure amount of data in keys, not number of keys
- not all of the data has to be examined

No algorithm can examine fewer bits than 3-way radix quicksort

- 1.39 N lg N bits for random data

Long strings are rarely random in practice.

- goal is often to learn the structure!
- may need specialized algorithms

lecture acronym cheatsheet	
LSD	least significant digit
MSD	most significant digit
LCP	longest common prefix
LRS	longest repeated substring

Tries

- review
- tries
- TSTs
- applications

References:

Algorithms in Java, Chapter 15

http://www.cs.princeton.edu/introalgsds/62search

- rules of the game
- tries
- TSTs
- applications

Review: summary of the performance of searching (symbol-table) algorithms

Frequency of execution of instructions in the inner loop:

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search	insert	delete		
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
randomized BST	7 lg N	7 lg N	7 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	lg N	lg N	lg N	yes	`compareTo()`

hashing	1*	1*	1*	1*	1*	1*	no	`equals()`
								`hashcode()`

* assumes random hash code

Q: Can we do better?

Review

Symbol tables.

- Associate a value with a key.
- Search for value given key.

Balanced trees

- use between lg N and 2 lg N key comparisons
- support ordered iteration and other operations

Hash tables

- typically use 1-2 probes
- require good hash function for each key type

Radix sorting

- some keys are inherently digital
- digital keys give linear and sublinear sorts

This lecture. Symbol tables for digital keys.

Digital keys (review)

Many commonly-use key types are inherently digital (sequences of fixed-length characters)

interface

```
interface Digital
{
   public int charAt(int k);
   public int length(int);
}
```

Examples

- Strings
- 64-bit integers

This lecture:

- refer to fixed-length vs. variable-length strings
- **R** different characters for some fixed value **R**.
- key type implements `charAt()` and `length()` methods
- code works for `String` and for key types that implement `Digital`.

Widely used in practice

- low-level bit-based keys
- string keys

Digital keys in applications

Key = sequence of "digits."

- DNA: sequence of a, c, g, t.
- IPv6 address: sequence of 128 bits.
- English words: sequence of lowercase letters.
- Protein: sequence of amino acids A, C, ..., Y.
- Credit card number: sequence of 16 decimal digits.
- International words: sequence of Unicode characters.
- Library call numbers: sequence of letters, numbers, periods.

This lecture. Key = string over ASCII alphabet.

String Set API

String set. Unordered collection of distinct strings.

public class StringSET

	`StringSET()`	create a set of strings
`void`	`add(String key)`	add string to set
`boolean`	`contains(String key)`	is **key** in the set?

Typical client: **Dedup** (remove duplicate strings from input)

```
StringSET set = new StringSET();
while (!StdIn.isEmpty())
{
   String key = StdIn.readString();
   if (!set.contains(key))
   {
      set.add(key);
      System.out.println(key);
   }
}
```

This lecture: focus on **StringSET** implementation

Same ideas improve STs with wider API

StringSET implementation cost summary

	typical case			dedup	
implementation	Search hit	Insert	Space	moby	actors
input *	L	L	L	0.26	15.1
red-black	L + log N	log N	C	1.40	97.4
hashing	L	L	C	0.76	40.6

* only reads in data

N = number of strings
L = length of string
C = number of characters in input
R = radix

file	megabytes	words	distinct
moby	1.2	210 K	32 K
actors	82	11.4 M	900 K

Challenge. Efficient performance for long keys (large L).

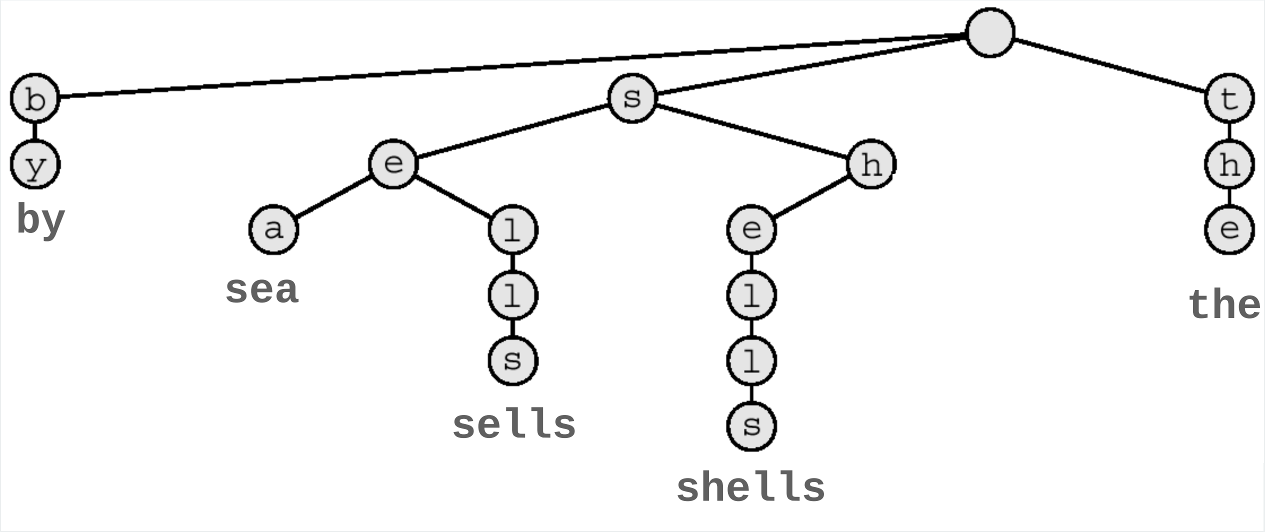
- rules of the game
- tries
- TSTs
- applications

Tries

Tries. [from retrieval, but pronounced "try"]

- Store characters in internal nodes, not keys.
- Store records in external nodes.
- Use the characters of the key to guide the search.

Ex. **`sells sea shells by the sea`**

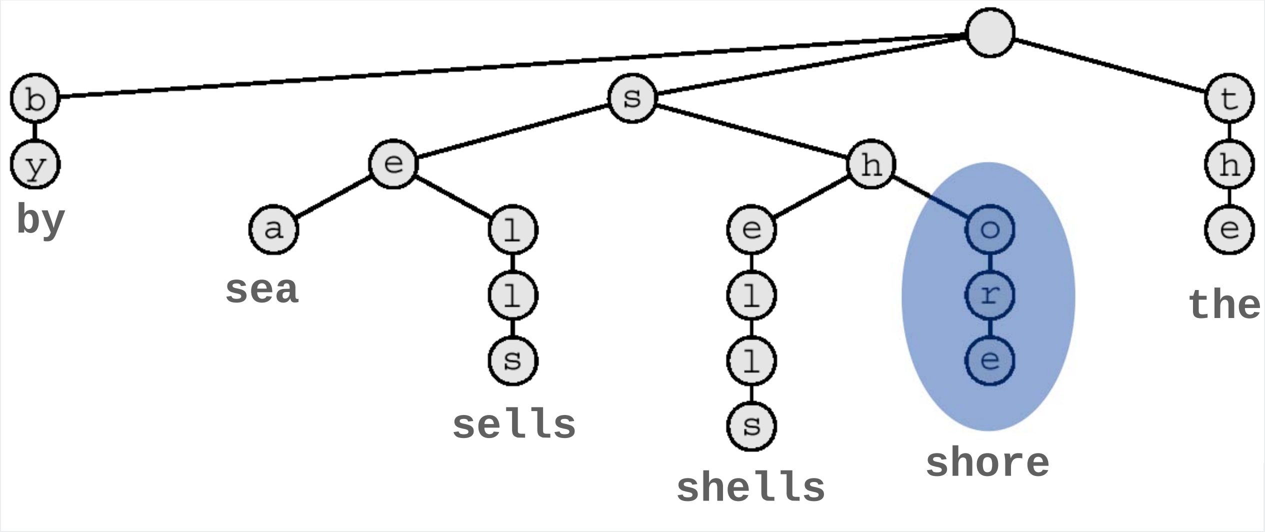


Tries

Tries. [from retrieval, but pronounced "try"]

- Store characters in internal nodes, not keys.
- Store records in external nodes.
- Use the characters of the key to guide the search.

Ex. **sells sea shells by the sea shore**



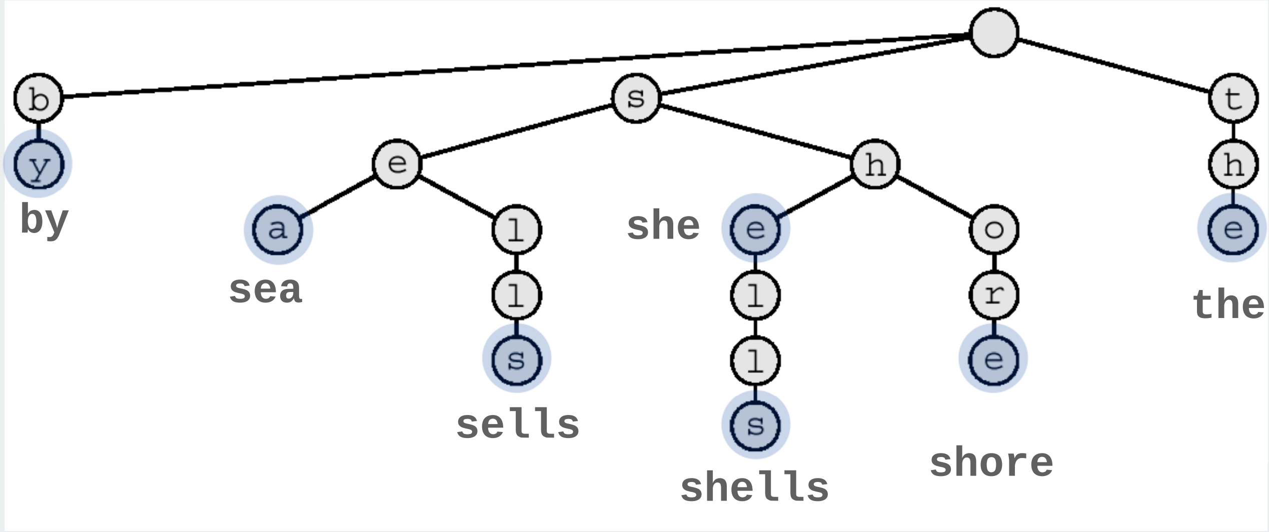
Tries

Q. How to handle case when one key is a prefix of another?

A1. Append sentinel character `'\0'` to every key so it never happens.

A2. Store extra bit to denote which nodes correspond to keys.

Ex. **she sells sea shells by the sea shore**

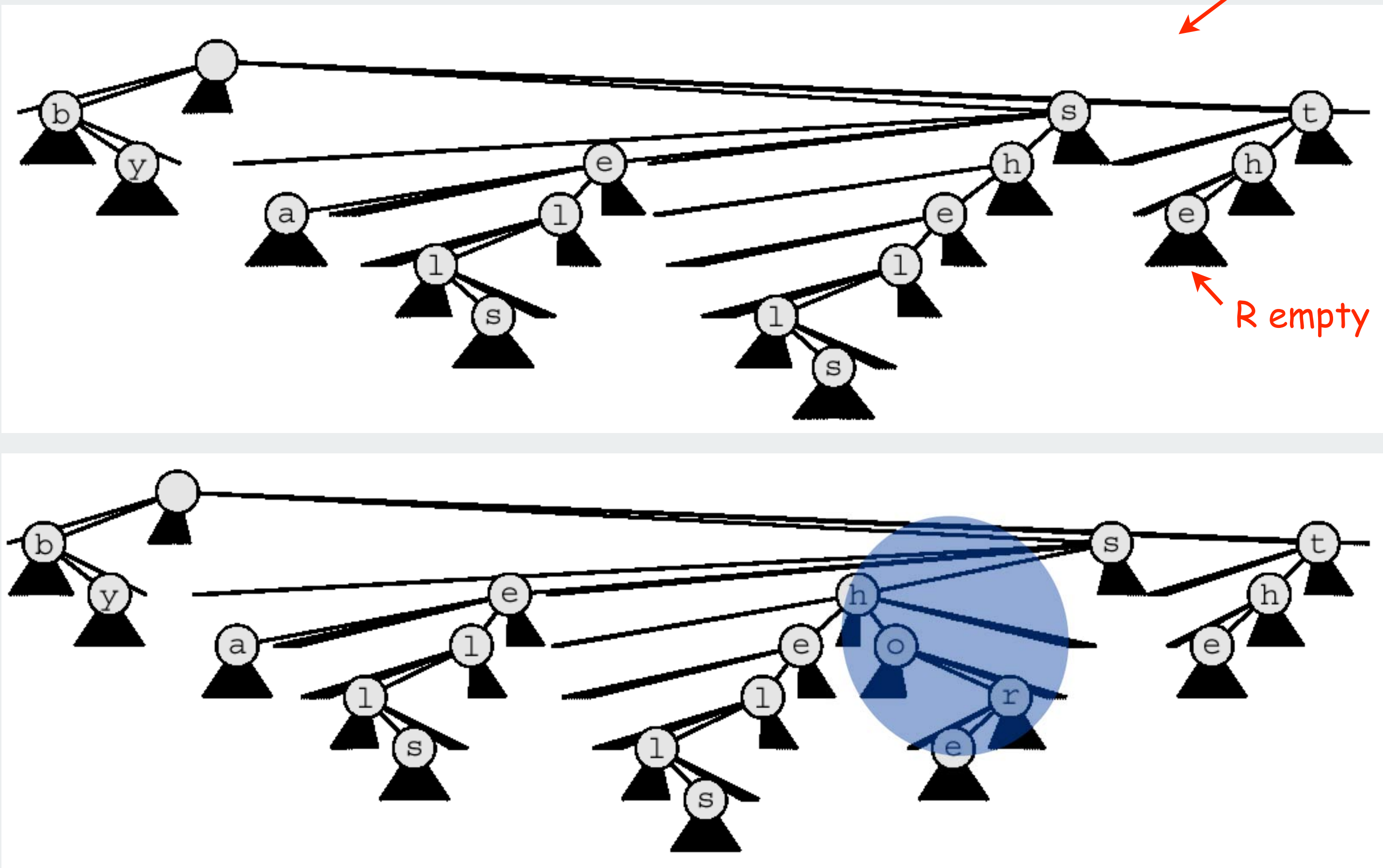


Branching in tries

Q. How to branch to next level?

A. One link for each possible character

Ex. `sells sea shells by the sea` **`shore`**



R-way trie

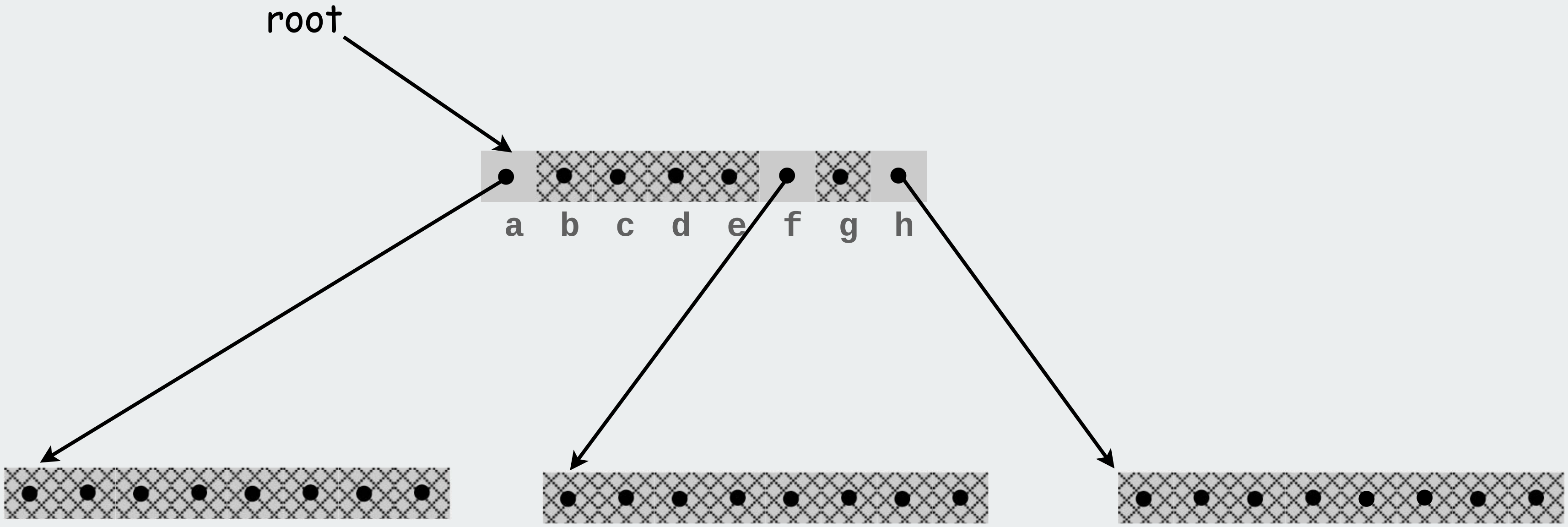
R empty links on leaves

R-Way Trie: Java implementation

R-way existence trie: a node.

Node: references to R nodes.

```
private class Node
{
   Node[] next = new Node[R];
   boolean end;
}
```



8-way trie that represents {`a, f, h`}

R-way trie implementation of **StringSET**

empty trie

current digit

```
public class StringSET
{
   private static final int R = 128;
   private Node root = new Node();

   private class Node
   {
      Node[] next = new Node[R];
      boolean end;
   }

   public boolean contains(String s)
   {  return contains(root, s, 0);  }

   private boolean contains(Node x, String s, int i)
   {
      if (x == null) return false;
      if (i == s.length()) return x.end;
      char c = s.charAt(i);
      return contains(x.next[c], s, i+1);
   }

   public void add(String s)
   // see next slide
}
```

R-way trie implementation of `StringSET` (continued)

```
public void add(String s)
{
   root = add(root, s, 0);
}

private Node add(Node x, String s, int i)
{
   if (x == null) x = new Node();
   if (i == s.length()) x.end = true;
   else
   {
      char c = s.charAt(i);
      x.next[c] = add(x.next[c], s, i+1);
   }
   return x;
}
```

R-way trie performance characteristics

Time

- examine one character to move down one level in the trie
- trie has ~$\log_R N$ levels (not many!)
- need to check whole string for search hit (equality)
- search miss only involves examining a few characters

Space

- R empty links at each leaf
- 65536-way branching for Unicode impractical

Bottom line.

- method of choice for small R
- you use tries every day
- stay tuned for ways to address space waste

Sublinear search with tries

Tries enable user to present string keys one char at a time

Search hit

- can present possible matches after a few digits
- need to examine all L digits for equality

Search miss

- could have mismatch on first character
- typical case: mismatch on first few characters

Bottom line: sublinear search cost (only a few characters)

Further help for Java `String` keys

- object equality test
- cached hash values

StringSET implementation cost summary

	typical case			dedup	
implementation	Search hit	Insert	Space	moby	actors
input *	L	L	L	0.26	15.1
red-black	L + log N	log N	C	1.40	97.4
hashing	L	L	C	0.76	40.6
R-way trie	L	<< L	RN + C	1.12	out of memory

N = number of strings
L = size of string
C = number of characters in input
R = radix

file	megabytes	words	distinct
moby	1.2	210 K	32 K
actors	82	11.4 M	900 K

R-way trie

- faster than hashing for small R
- too much memory if R not small

65536-way trie for Unicode??

Challenge. Use less memory!

Digression: Out of memory?

"640 K ought to be enough for anybody."

- attributed to Bill Gates, 1981

(commenting on the amount of RAM in personal computers)

"64 MB of RAM may limit performance of some Windows XP features; therefore, 128 MB or higher is recommended for best performance." - Windows XP manual, 2002

"64 bit is coming to desktops, there is no doubt about that. But apart from Photoshop, I can't think of desktop applications where you would need more than 4GB of physical memory, which is what you have to have in order to benefit from this technology. Right now, it is costly." - Bill Gates, 2003

Digression: Out of memory?

A short (approximate) history

		address bits	addressable memory	typical actual memory	cost
PDP-8	1960s	12	6K	6K	$16K
PDP-10	1970s	18	256K	256K	$1M
IBM S/360	1970s	24	4M	512K	$1M
VAX	1980s	32	4G	1M	$1M
Pentium	1990s	32	4G	1 GB	$1K
Xeon	2000s	64	enough	4 GB	$100
??	future	128+	enough	enough	$1

A modest proposal

Number of atoms in the universe: < 2^{266} (estimated)

Age of universe (estimated): 20 billion years ~ 2^{50} secs < 2^{80} nanoseconds

How many bits address every atom that ever existed ?

A modest proposal: use a unique 512-bit address for every object

512 bits is enough:

266 bits	80 bits	174 bits
place	time	cushion for whatever

current plan:

128 bits	64 bits
place (ipv6)	place (machine)

Use trie to map to current location. 64 8-bit chars

- wastes 255/256 actual memory
- need better use of memory

maybe OK for Bill Gates
or if memory is tiny

- rules of the game
- tries
- TSTs
- applications

Ternary Search Tries (TSTs)

Ternary search tries. [Bentley-Sedgewick, 1997]

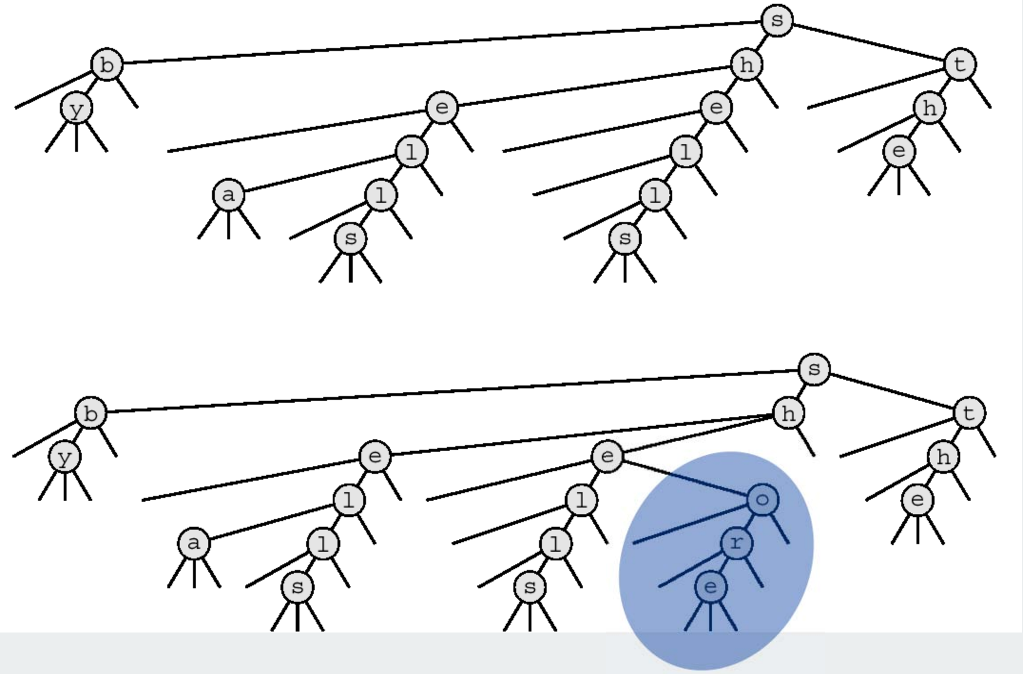
- Store characters in internal nodes, records in external nodes.
- Use the characters of the key to guide the search
- Each node has three children
- Left (smaller), middle (equal), right (larger).

Ternary Search Tries (TSTs)

Ternary search tries. [Bentley-Sedgewick, 1997]

- Store characters in internal nodes, records in external nodes.
- Use the characters of the key to guide the search
- Each node has three children:
  left (smaller), middle (equal), right (larger).

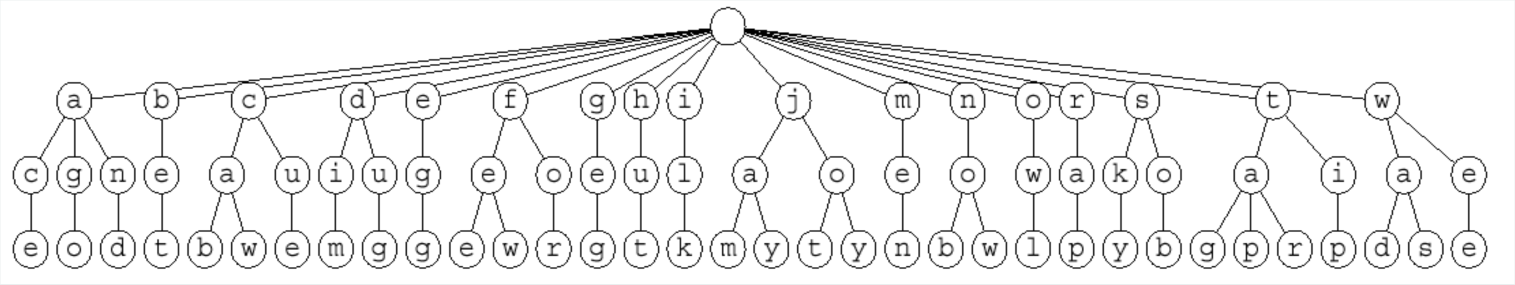
Ex. **sells sea shells by the sea shore**



Observation. Only three null links in leaves!

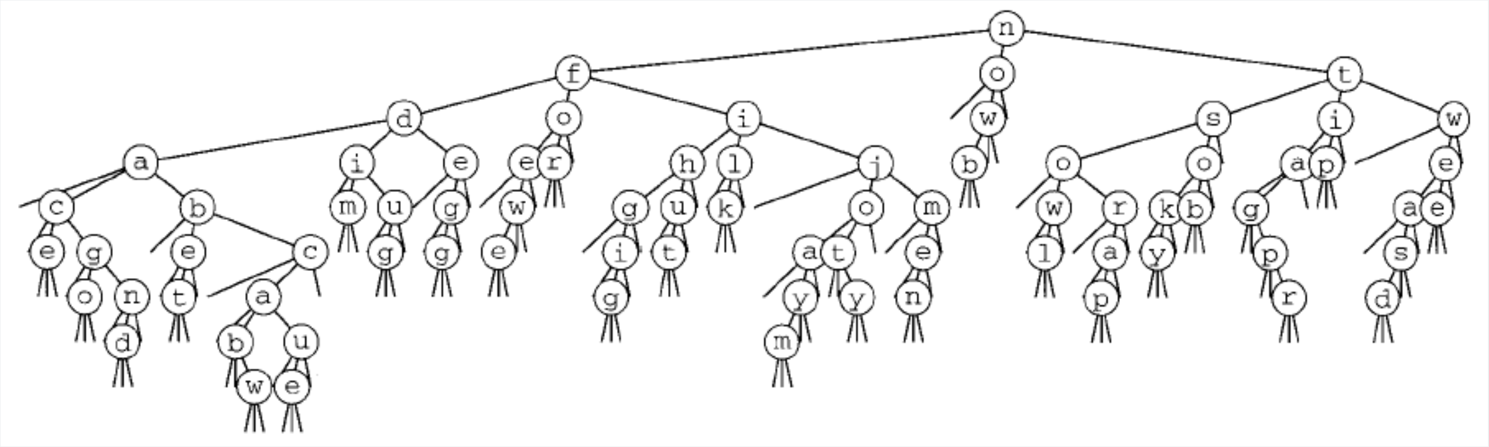
26-Way Trie vs. TST

TST. Collapses empty links in 26-way trie.



a b c d e f g h i j m n o r s t w
c g n e a u i u g e o e u l a o e o w a k o a i a e
e o d t b w e m g g e w r g t k m y t y n b w l p y b g p r p d s e

26-way trie (1035 null links, not shown)



TST (155 null links)

now
for
tip
ilk
dim
tag
jot
sob
nob
sky
hut
ace
bet
men
egg
few
jay
owl
joy
rap
gig
wee
was
cab
wad
caw
cue
fee
tap
ago
tar
jam
dug
and

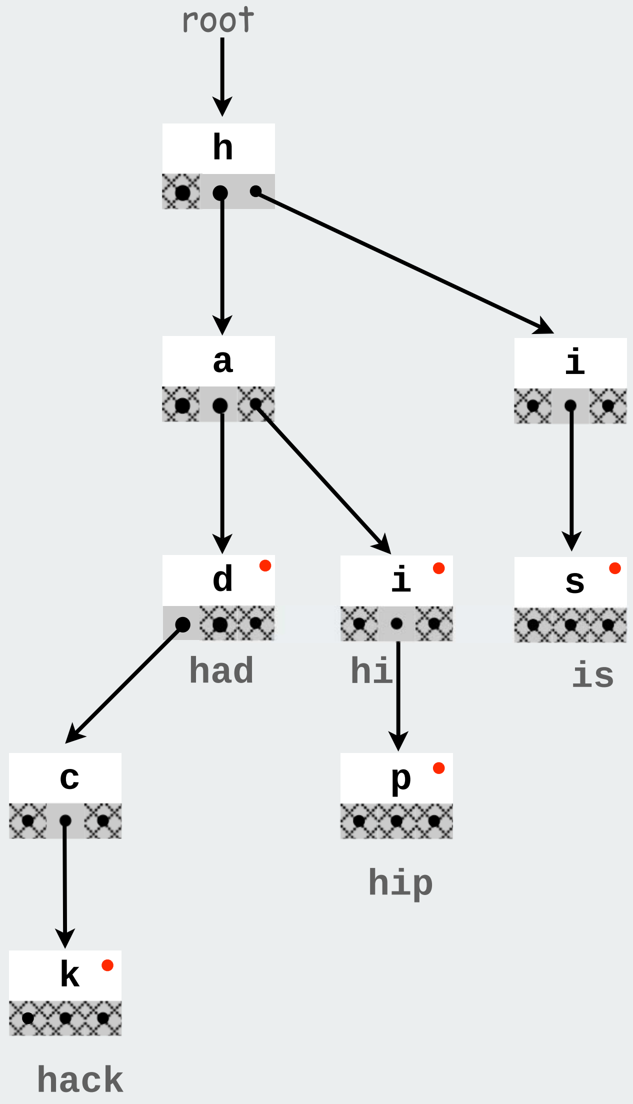
TST representation

A TST string set is a TST node.

A TST node is five fields:

- a character c.
- a reference to a left TST. [smaller]
- a reference to a middle TST. [equal]
- a reference to a right TST. [larger]
- a bit to indicate whether this node is the last character in some key.

```
private class Node
{
   char c;
   Node l, m, r;
   boolean end;
}
```



TST implementation of **contains()** for **StringSET**

Recursive code practically writes itself!

```
public boolean contains(String s)
{
   if (s.length() == 0) return false;
   return contains(root, s, 0);
}

private boolean contains(Node x, String s, int i)
{
   if (x == null) return false;
   char c = s.charAt(i);
   if      (c < x.c)            return contains(x.l, s, i);
   else if (c > x.c)            return contains(x.r, s, i);
   else if (i < s.length()-1) return contains(x.m, s, i+1);
   else                         return x.end;
}
```

TST implementation of add() for StringSET

```
public void add(String s)
{
   root = add(root, s, 0);
}


private Node add(Node x, String s, int i)
{
   char c = s.charAt(i);
   if (x == null) x = new Node(c);
   if      (c < x.c)             x.l = add(x.l, s, i);
   else if (c > x.c)             x.r = add(x.r, s, i);
   else if (i < s.length()-1)  x.m = add(x.m, s, i+1);
   else                          x.end = true;
   return x;
}
```

StringSET implementation cost summary

	typical case			dedup	
implementation	Search hit	Insert	Space	moby	actors
input *	L	L	L	0.26	15.1
red-black	L + log N	log N	C	1.40	97.4
hashing	L	L	C	0.76	40.6
R-way trie	L	L	RN + C	1.12	out of memory
TST	L	L	3C	0.72	38.7

N = number of strings
L = size of string
C = number of characters in input
R = radix

TST

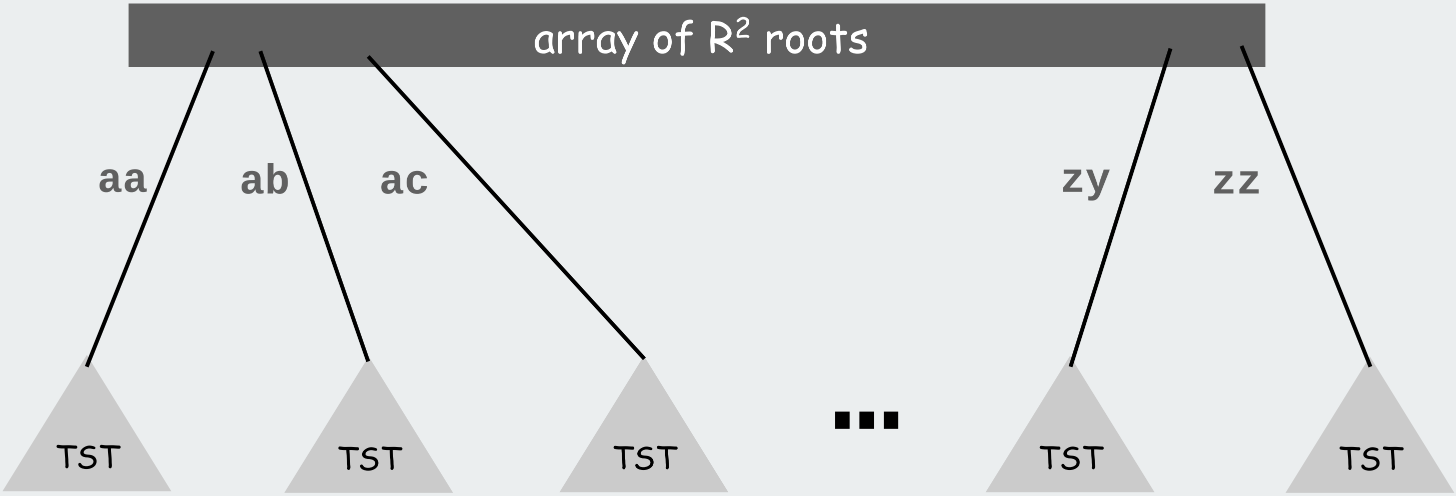
- faster than hashing
- space usage independent of R
- supports extended APIs (stay tuned)
- Unicode no problem

Space-efficient trie: challenge met.

TST With R^2 Branching At Root

Hybrid of R-way and TST.

- Do R-way or R^2-way branching at root.
- Each of R^2 root nodes points to a TST.



Note. Need special test for one-letter words.

StringSET implementation cost summary

	typical case			dedup	
implementation	Search hit	Insert	Space	moby	actors
input *	L	L	L	0.26	15.1
red-black	L + log N	log N	C	1.40	97.4
hashing	L	L	C	0.76	40.6
R-way trie	L	L	RN + C	1.12	out of memory
TST	L	L	3C	.72	38.7
TST with R^2	L	L	$3C + R^2$	.51	32.7

TST performance even better with nonuniform keys

Ex. Library call numbers

```
WUS-------10706-----7---10
WUS-------12692-----4---27
WLSOC------2542----30
LTK--6015-P-63-1988
LDS---361-H-4
 ...
```

TSTs 5 times faster than hashing

TST summary

Hashing.

- need to examine entire key
- hits and misses cost about the same.
- need good hash function for every key type
- no help for ordered-key APIs

TSTs.

- need to examine just enough key characters
- search miss may only involve a few characters
- works only for keys types that implement **`charAt()`**
- can handle ordered-key APIs

Bottom line:
TSTs are faster than hashing and more flexible than LL RB trees

- rules of the game
- tries
- TSTs
- applications

Extending the `StringSET` API

Add. Insert a key.
Contains. Check if given key in the set.
Delete. Delete key from the set.

`equals()`

Sort. Iterate over keys in ascending order.
Select. Find the k^{th} largest key.
Range search. Find all elements between k_1 and k_2.

`compareTo()`

Longest prefix match. Find longest prefix match.
Wildcard match. Allow wildcard characters.
Near neighbor search. Find strings that differ in ≤ P chars.

`charAt()`

Longest Prefix Match

Find string in set with longest prefix matching given key.

Ex. Search IP database for longest prefix matching destination IP, and route packets accordingly.

```
"128"
"128.112"
"128.112.136"
"128.112.055"
"128.112.055.15"
"128.112.155.11"
"128.112.155.13"
"128.222"
"128.222.136"

prefix("128.112.136.11") = "128.112.136"
prefix("128.166.123.45") = "128"
```

R-way trie implementation of longest prefix match operation

Find string in set with longest prefix matching a given key.

```
public String prefix(String s)
{
   int length = prefix(root, s, 0);
   return s.substring(0, length);
}

private int prefix(Node x, String s, int i)
{
   if (x == null) return 0;
   int length = 0;
   if (x.end) length = i;
   if (i == s.length()) return length;
   char c = s.charAt(i);
   return Math.max(length, prefix(x.next[c], s, i+1));
}
```

Wildcard Match

Wildcard match. Use wildcard . to match any character.

```
coalizer
coberger
codifier
cofaster
cofather
cognizer
cohelper
colander
coleader
...
compiler
...
composer
computer
cowkeper
```

```
co....er
```

```
acresce
acroach
acuracy
octarch
science
scranch
scratch
scrauch
screich
scrinch
scritch
scrunch
scudick
scutock
```

```
.c...c.
```

TST implementation of wildcard match operation

Wildcard match. Use wildcard . to match any character.

- Search as usual if query character is not a period.
- Go down all three branches if query character is a period.

for printing out matches
(use StringBuilder for long keys)

```
public void wildcard(String s)
{  wildcard(root, s, 0, "");  }

private void wildcard(Node x, String s, int i, String prefix)
{
   if (x == null) return;
   char c = s.charAt(i);
   if (c == '.' || c <  x.c) wildcard(x.left,  s, i, prefix);
   if (c == '.' || c == x.c)
   {
      if (i < s.length() - 1)
         wildcard(x.mid, s, i+1, prefix + x.c);
      else if (x.end)
         System.out.println(prefix + x.c);
   }
   if (c == '.' || c >  x.c)  wildcard(x.right, s, i, prefix);
}
```

T9 Texting

Goal. Type text messages on a phone keypad.

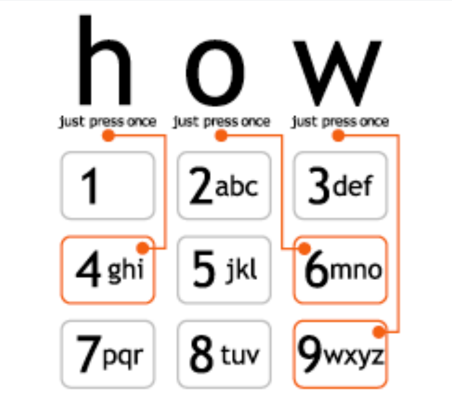
Multi-tap input. Enter a letter by repeatedly pressing a key until the desired letter appears.

T9 text input. ["A much faster and more fun way to enter text."]

- Find all words that correspond to given sequence of numbers.
- Press 0 to see all completion options.

Ex: `hello`

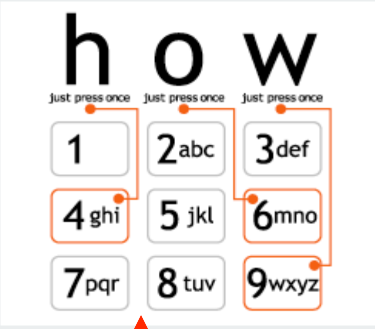
- Multi-tap: `4 4 3 3 5 5 5 5 5 5 6 6 6`
- T9: `4 3 5 5 6`



www.t9.com

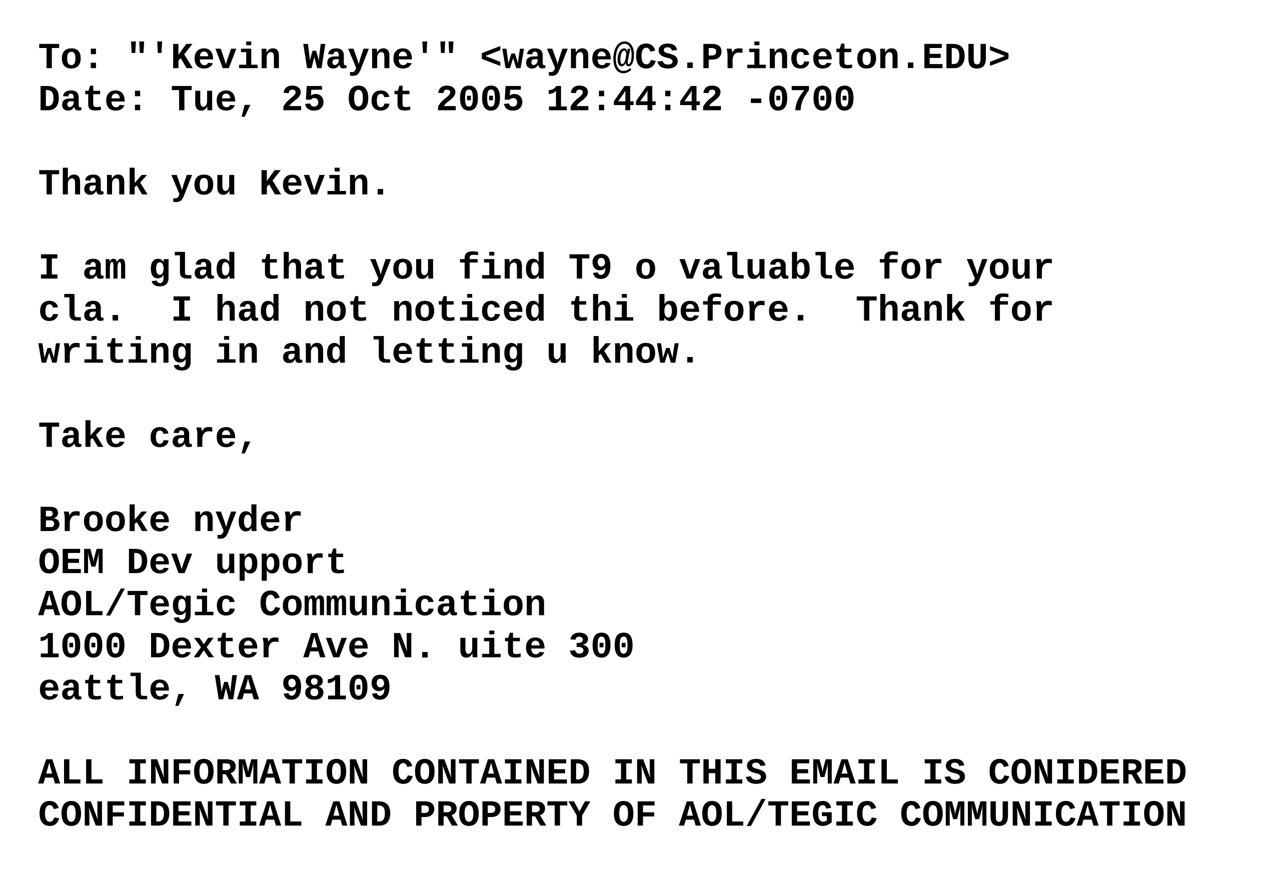
A Letter to t9.com

```
To: info@t9support.com
Date: Tue, 25 Oct 2005 14:27:21 -0400 (EDT)

Dear T9 texting folks,

I enjoyed learning about the T9 text system
from your webpage, and used it as an example
in my data structures and algorithms class.
However, one of my students noticed a bug
in your phone keypad

   http://www.t9.com/images/how.gif

Somehow, it is missing the letter s. (!)

Just wanted to bring this information to
your attention and thank you for your website.

Regards,

Kevin
```



where's the "s" ??

A world without "s" ??



To: "'Kevin Wayne'" <wayne@CS.Princeton.EDU>
Date: Tue, 25 Oct 2005 12:44:42 -0700

Thank you Kevin.

I am glad that you find T9 o valuable for your cla. I had not noticed thi before. Thank for writing in and letting u know.

Take care,

Brooke nyder
OEM Dev upport
AOL/Tegic Communication
1000 Dexter Ave N. uite 300
eattle, WA 98109

ALL INFORMATION CONTAINED IN THIS EMAIL IS CONIDERED CONFIDENTIAL AND PROPERTY OF AOL/TEGIC COMMUNICATION

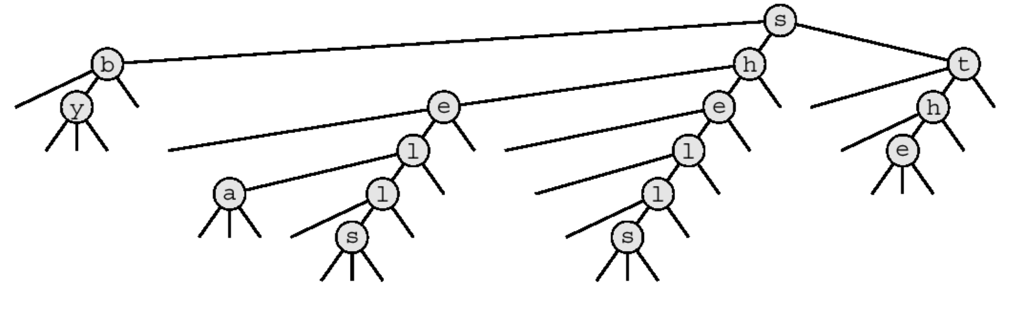
TST: Collapsing 1-Way Branches

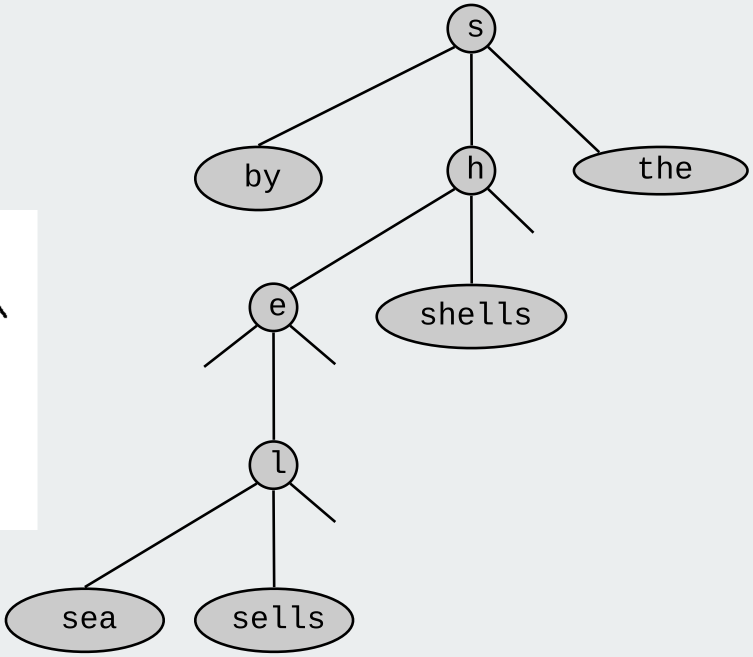
Collapsing 1-way branches at bottom.

- internal node stores **char**; external node stores full key.
- append sentinel character **'\0'** to every key
- search hit ends at leaf with given key.
- search miss ends at **null** link or leaf with different key.

Collapsing interior 1-way branches

- keep char position in nodes
- need full compare at leaf





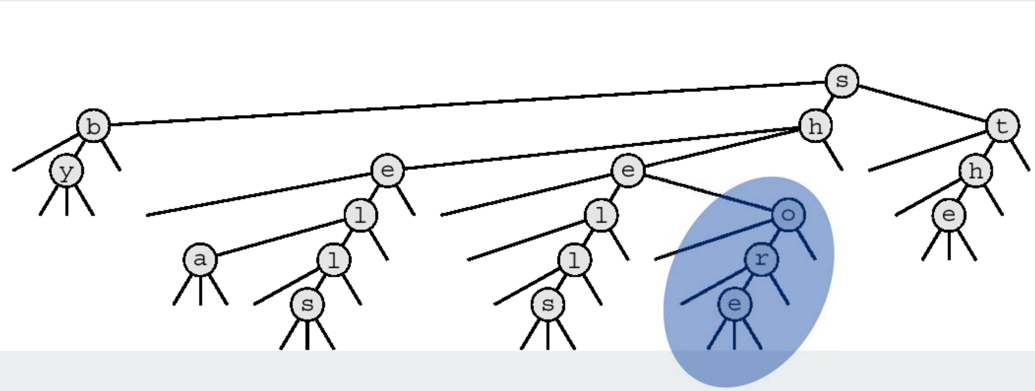
TST: Collapsing 1-Way Branches

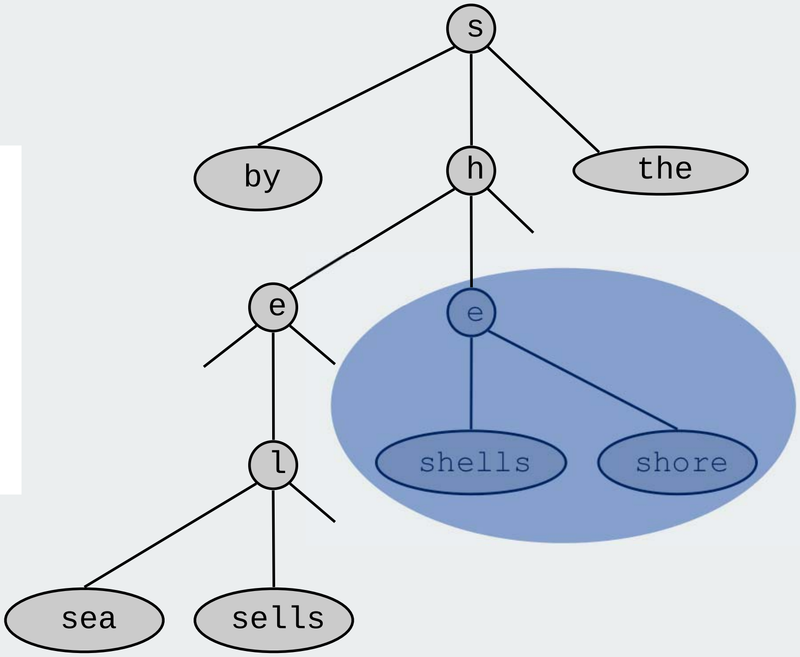
Collapsing 1-way branches at bottom.

- internal node stores **char**; external node stores full key.
- append sentinel character **'\0'** to every key
- search hit ends at leaf with given key.
- search miss ends at **null** link or leaf with different key.

Collapsing interior 1-way branches

- keep char position in nodes
- need full compare at leaf





StringSET implementation cost summary

implementation	Search hit	Insert	Space
input *	L	L	L
red-black	L + log N	log N	C
hashing	L	L	C
R-way trie	L	L	RN + C
TST	L	L	3C
TST with R^2	L	L	3C + R^2
R-way with no 1-way	$\log_R N$	$\log_R N$	RN + C
TST with no 1-way	log N	log N	C

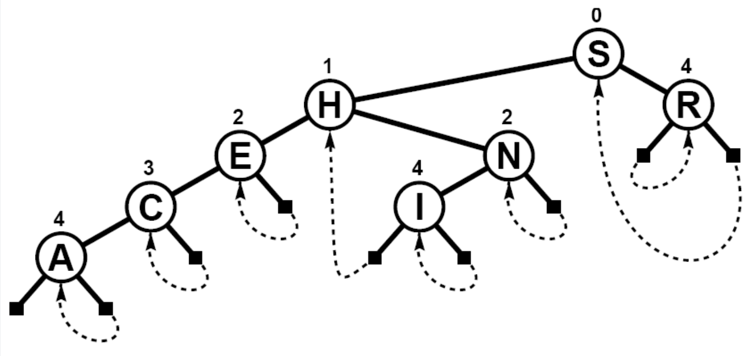
Challenge met.

- Efficient performance for arbitrarily long keys.
- Search time is independent of key length!

A classic algorithm

Patricia tries. [Practical Algorithm to Retrieve Information Coded in Alphanumeric]

- Collapse one-way branches in binary trie.
- Thread trie to eliminate multiple node types.



Applications.

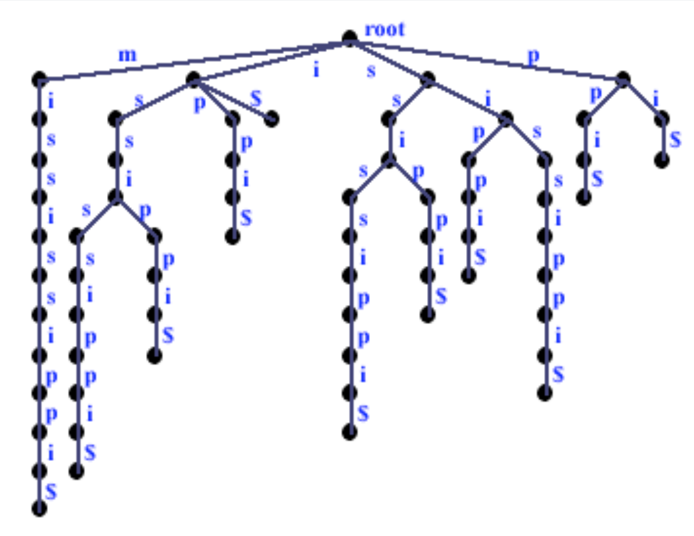
- Database search.
- P2P network search.
- IP routing tables: find longest prefix match.
- Compressed quad-tree for N-body simulation.
- Efficiently storing and querying XML documents.

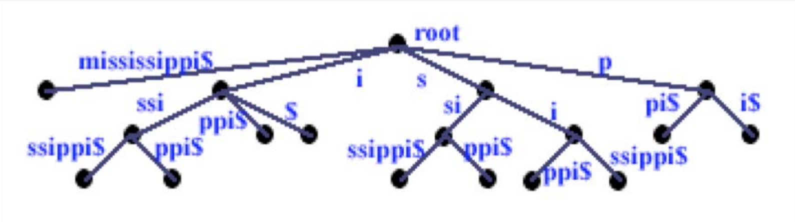
(Just slightly) beyond the scope of COS 226 (see Program 15.7)

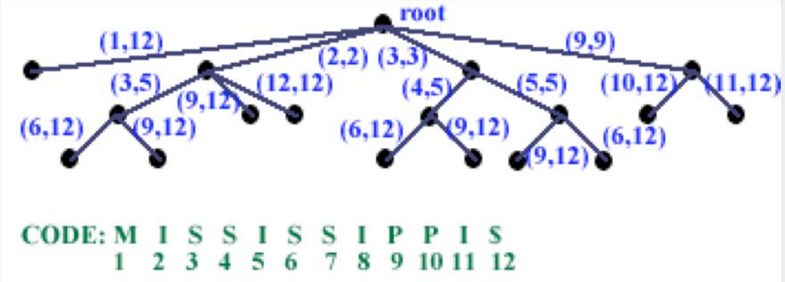
Suffix Tree

Suffix tree.

Threaded trie with collapsed 1-way branching for string suffixes.







Applications.

- Longest common substring, longest repeated substring.
- Computational biology databases (BLAST, FASTA).
- Search for music by melody.
- ...

(Just slightly) beyond the scope of COS 226.

Symbol tables summary

A success story in algorithm design and analysis.
Implementations are a critical part of our computational infrastructure.

Binary search trees. Randomized, red-black.

- performance guarantee: log N compares
- supports extensions to API based on key order

Hash tables. Separate chaining, linear probing.

- performance guarantee: N/M probes
- requires good hash function for key type
- no support for API extensions
- enjoys systems support (ex: cached value for String)

Tries. R-way, TST.

- performance guarantee: log N characters accessed
- supports extensions to API based on partial keys

Bottom line: you can get at anything by examining 50-100 bits (!!!)

Data Compression

- introduction
- basic coding schemes
- an application
- entropy
- LZW codes

References:

Algorithms 2nd edition, Chapter 22

http://www.cs.princeton.edu/introalgsds/65compression

- introduction
- basic coding schemes
- an application
- entropy
- LZW codes

Data Compression

Compression reduces the size of a file:

- To save space when storing it.
- To save time when transmitting it.
- Most files have lots of redundancy.

Who needs compression?

- Moore's law: # transistors on a chip doubles every 18-24 months.
- Parkinson's law: data expands to fill space available.
- Text, images, sound, video, …

> All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year. Not all bits have equal value. *-Carl Sagan*

Basic concepts ancient (1950s), best technology recently developed.

Applications

Generic file compression.

- Files: GZIP, BZIP, BOA.
- Archivers: PKZIP.
- File systems: NTFS.

Multimedia.

- Images: GIF, JPEG.
- Sound: MP3.
- Video: MPEG, DivX™, HDTV.

Communication.

- ITU-T T4 Group 3 Fax.
- V.42bis modem.

Databases. Google.

Google™

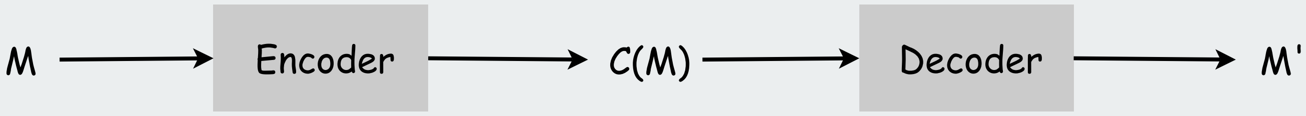
Encoding and decoding

Message. Binary data M we want to compress.

Encode. Generate a "compressed" representation C(M).

uses fewer bits (you hope)

Decode. Reconstruct original message or some approximation M'.



Compression ratio. Bits in C(M) / bits in M.

Lossless. M = M', 50-75% or lower.

Ex. Natural language, source code, executables.

← this lecture

Lossy. M ≈ M', 10% or lower.

Ex. Images, sound, video.

"Poetry is the art of lossy data compression."

Food for thought

Data compression has been omnipresent since antiquity,

- Number systems.
- Natural languages.
- Mathematical notation.

has played a central role in communications technology,

- Braille.
- Morse code.
- Telephone system.

and is part of modern life.

- zip.
- MP3.
- MPEG.

What role will it play in the future?

Ex: If memory is to be cheap and ubiquitous, why are we doing lossy compression for music and movies??

- introduction
- **basic coding schemes**
- an application
- entropy
- LZW codes

Fixed length encoding

- Use same number of bits for each symbol.
- k-bit code supports 2^k different symbols

Ex. 7-bit ASCII

char	decimal	code
NUL	0	0
...	...	
a	97	1100001
b	98	1100010
c	99	1100011
d	100	1100100
...	...	
~	126	1111110
	127	1111111

this lecture:
special code for
end-of-message

a	b	r	a	c	a	d	a	b	r	a	!
1100001	1100010	1110010	1100001	1100011	1100001	1100100	1100001	1100010	1110010	1100001	1111111

12 symbols × 7 bits per symbol = 84 bits in code

Fixed length encoding

- Use same number of bits for each symbol.
- k-bit code supports 2^k different symbols

Ex. 3-bit custom code

char	code
a	000
b	001
c	010
d	011
r	100
!	111

a	b	r	a	c	a	d	a	b	r	a	!
000	001	100	000	010	000	011	000	001	100	000	111

12 symbols × 3 bits
36 bits in code

Important detail: decoder needs to know the code!

Fixed length encoding: general scheme

- count number of different symbols.
- $\lfloor \lg M \rfloor$ bits suffice to support M different symbols

Ex. genomic sequences

- 4 different codons
- 2 bits suffice

char	code
a	00
c	01
t	10
g	11

a	c	t	a	c	a	g	a	t	g	a
00	01	10	00	01	00	11	00	10	11	00

2N bits to encode genome with N codons

- Amazing but true: initial databases in 1990s did not use such a code!

Decoder needs to know the code

- can amortize over large number of files with the same code
- in general, can encode an N-char file with $N \lfloor \lg M \rfloor + 16 \lfloor \lg M \rfloor$ bits

Variable-length encoding

Use different number of bits to encode different characters.

Ex. Morse code.

Issue: ambiguity.

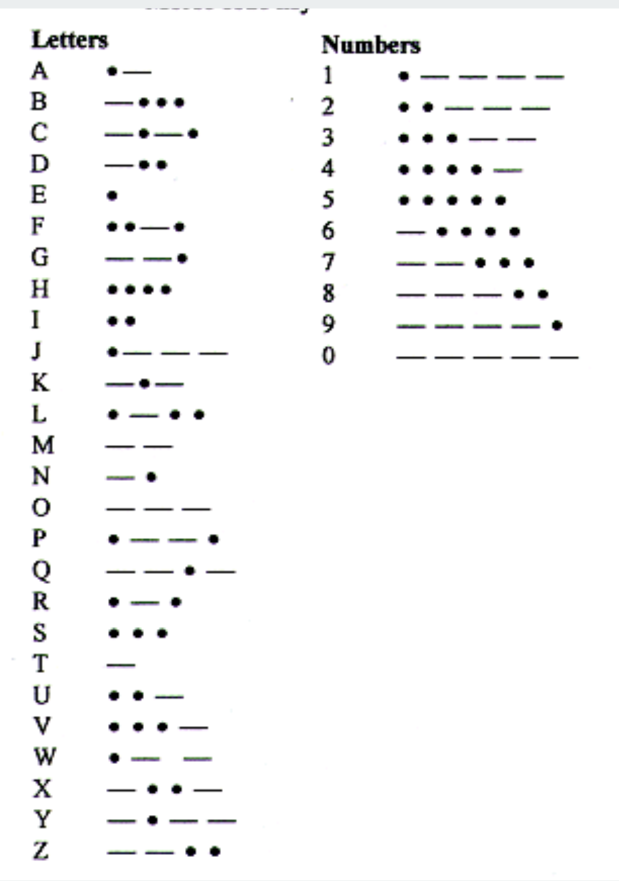
• • • — — — • • •

SOS ?

IAMIE ?

EEWNI ?

V7O ?



Letters		Numbers	
A	•—	1	• — — — —
B	—•••	2	• • — — —
C	—•—•	3	• • • — —
D	—••	4	• • • • —
E	•	5	• • • • •
F	••—•	6	— • • • •
G	— —•	7	— — • • •
H	••••	8	— — — • •
I	••	9	— — — — •
J	•— — —	0	— — — — —
K	—•—		
L	• — • •		
M	— —		
N	— •		
O	— — —		
P	• — — •		
Q	— — • —		
R	• — •		
S	• • •		
T	—		
U	• • —		
V	• • • —		
W	• — —		
X	— • • —		
Y	— • — —		
Z	— — • •		

Variable-length encoding

Use different number of bits to encode different characters.

Q. How do we avoid ambiguity?

A1. Append special stop symbol to each codeword.

A2. Ensure that no encoding is a prefix of another.

S	• • •	← prefix of V
E	•	← prefix of I, S
I	• •	← prefix of S
V	• • • -	

Ex. custom prefix-free code

char	code
a	0
b	111
c	1010
d	100
r	110
!	1011

28 bits in code

a	b	r	a	c	a	d	a	b	r	a	!
0	1 1 1	1 1 0	0	1 0 1 0	0	1 0 0	0	1 1 1	1 1 0	0	1 0 1 1

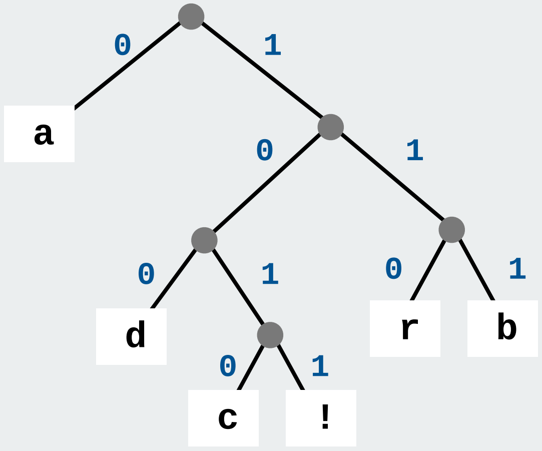
Note 1: fixed-length codes are prefix-free

Note 2: can amortize cost of including the code over similar messages

Prefix-free code: Encoding and Decoding

How to represent? Use a binary trie.

- Symbols are stored in leaves.
- Encoding is path to leaf.



Encoding.

- Method 1: start at leaf; follow path up to the root, and print bits in reverse order.
- Method 2: create ST of symbol-encoding pairs.

char	encoding
a	0
b	111
c	1010
d	100
r	110
!	1011

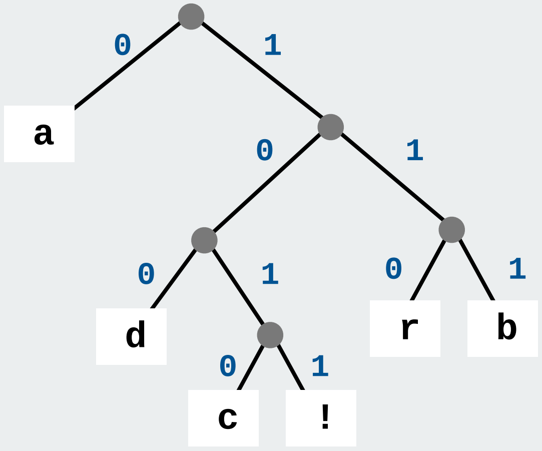
Decoding.

- Start at root of tree.
- Go left if bit is 0; go right if 1.
- If leaf node, print symbol and return to root.

Providing the code

How to transmit the trie?

- send preorder traversal of trie.
  we use * as sentinel for internal nodes
  [what if no sentinel is available?]
- send number of characters to decode.
- send bits (packed 8 to the byte).



preorder traversal	`*a**d*c!*rb`
# chars to decode	`12`
the message bits	`0111110010100100011111001011`

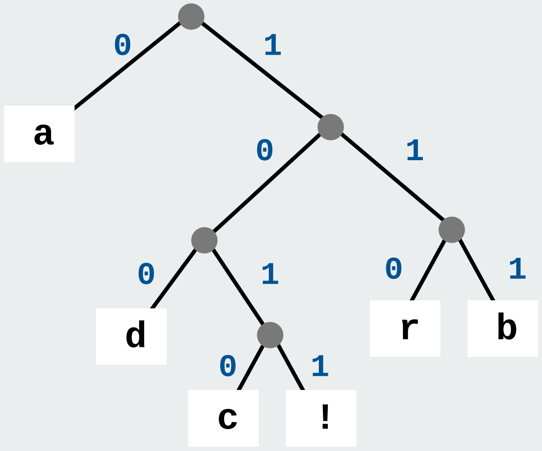
If message is long, overhead of transmitting trie is small.

```
| a |   b   |   r   | a |    c    | a |   d   | a |   b   |   r   | a |    !    |
  0   1 1 1   1 1 0   0   1 0 1 0   0   1 0 0   0   1 1 1   1 1 0   0   1 0 1 1
```

char	encoding
a	0
b	111
c	1010
d	100
r	110
!	1011

Prefix-free decoding implementation

```
public class PrefixFreeDecoder
{
   private Node root = new Node();
   private class Node
   {
      char ch;
      Node left, right;
      Node()
      {
         ch = StdIn.readChar();
         if (ch == '*')
         {
            left  = new Node();
            right = new Node();
         }
      }

     boolean isInternal() { }

   }

   public void decode()
   {  /* See next slide. */   }

}
```



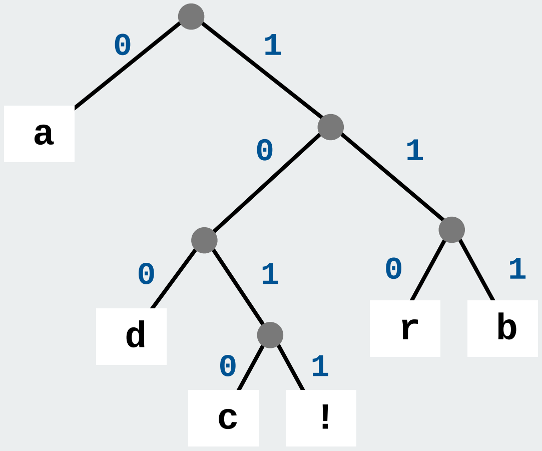
build tree from
preorder traversal

***a**d*c!*rb**

Prefix-free decoding iImplementation

```
public void decode()
{
   int N = StdIn.readInt();
   for (int i = 0; i < N; i++)
   {
      Node x = root;
      while (x.isInternal())
      {
         char bit = StdIn.readChar();
         if      (bit == '0') x = x.left;
         else if (bit == '1') x = x.right;
      }
      System.out.print(x.ch);
   }
}
```

use bits, not chars in actual applications →



```
more code.txt
12
01111100101001000111111001011

% java PrefixFreeDecoder < code.txt
abacadabra!
```

Introduction to compression: summary

Variable-length codes can provide better compression than fixed-length

a	b	r	a	c	a	d	a	b	r	a	!
1100001	1100010	1110010	1100001	1100011	1100001	1100100	1100001	1100010	1110010	1100001	1111111

a	b	r	a	c	a	d	a	b	r	a	!
000	001	100	000	010	000	011	000	001	100	000	111

a	b	r	a	c	a	d	a	b	r	a	!
0	1 1 1	1 1 0	0	1 0 1 0	0	1 0 0	0	1 1 1	1 1 0	0	1 0 1 1

Every trie defines a variable-length code

Q. What is the best variable length code for a given message?

Huffman coding

Q. What is the best variable length code for a given message?

A. Huffman code. [David Huffman, 1950]

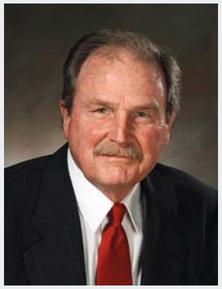
To compute Huffman code:

- count frequency p_s for each symbol s in message.
- start with one node corresponding to each symbol s (with weight p_s).
- repeat until single trie formed:

  select two tries with min weight p_1 and p_2

  merge into single trie with weight $p_1 + p_2$

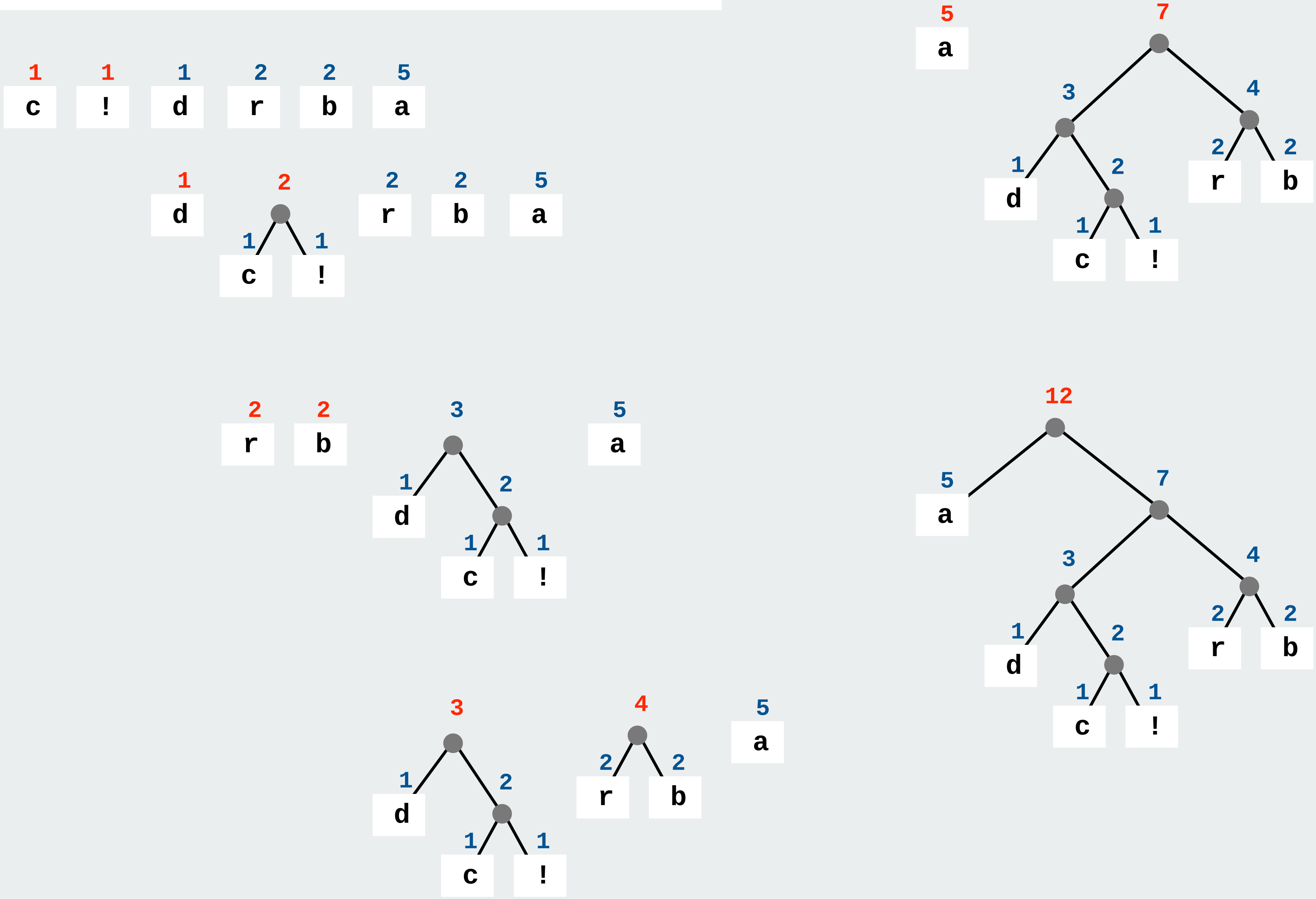
Applications. JPEG, MP3, MPEG, PKZIP, GZIP, ...



David Huffman

Huffman coding example

a b r a c a d a b r a !



1 c 1 ! 1 d 2 r 2 b 5 a
1 d 2 1 c 1 ! 2 r 2 b 5 a
2 r 2 b 3 1 d 2 1 c 1 ! 5 a
3 1 d 2 1 c 1 ! 4 2 r 2 b 5 a
5 a 7 3 1 d 2 1 c 1 ! 4 2 r 2 b
12 5 a 7 3 1 d 2 1 c 1 ! 4 2 r 2 b

Huffman trie construction code

```
int[] freq = new int[128];
for (int i = 0; i < input.length(); i++)
{  freq[input.charAt(i)]++;  }
```

tabulate frequencies

```
MinPQ<Node> pq = new MinPQ<Node>();
for (int i = 0; i < 128; i++)
   if (freq[i] > 0)
      pq.insert(new Node((char) i, freq[i], null, null));
```

initialize PQ

```
while (pq.size() > 1)
{
   Node x = pq.delMin();
   Node y = pq.delMin();
   Node parent = new Node('*', x.freq + y.freq, x, y);
   pq.insert(parent);
}
root = pq.delMin();
```

merge trees

two subtrees

internal node marker

total frequency

Huffman encoding summary

Theorem. Huffman coding is an optimal prefix-free code.

no prefix-free code uses fewer bits

Implementation.

- pass 1: tabulate symbol frequencies and build trie
- pass 2: encode file by traversing trie or lookup table.

Running time. Use binary heap ⇒ $O(M + N \log N)$.

output bits

distinct symbols

Can we do better? [stay tuned]

- introduction
- basic coding schemes
- an application
- entropy
- LZW codes

An application: compress a bitmap

Typical black-and-white-scanned image

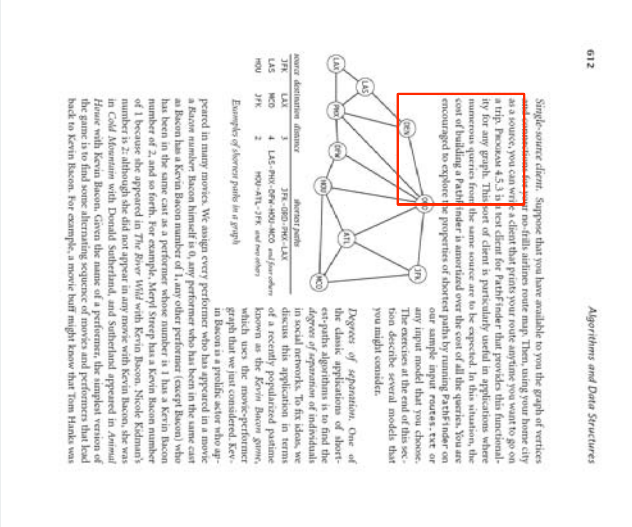
300 pixels/inch

8.5 by 11 inches

300*8.5*300*11 = 8.415 million bits

Bits are mostly white

Typical amount of text on a page:
40 lines * 75 chars per line = 3000 chars



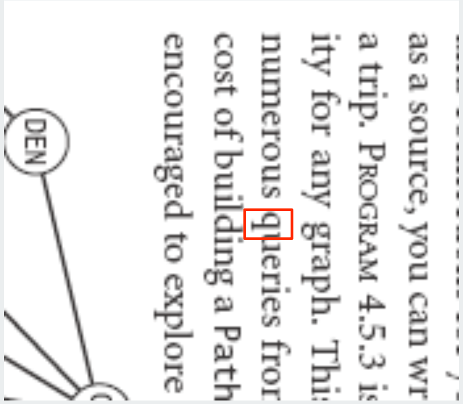
612 *Algorithms and Data Structures*

Single-source client. Suppose that you have available to you the graph of vertices and connections for your no-frills airlines route map. Then, using your home city as a source, you can write a client that prints your route anytime you want to go on a trip. PROGRAM 4.5.3 is a test client for PathFinder that provides this functionality for any graph. This sort of client is particularly useful in applications where numerous queries from the same source are to be expected. In this situation, the cost of building a PathFinder is amortized over the cost of all the queries. You are encouraged to explore the properties of shortest paths by running PathFinder on our sample input routes.txt or any input model that you choose. The exercises at the end of this section describe several models that you might consider.

source	*destination*	*distance*	*shortest paths*
JFK	LAX	3	JFK-ORD-PHX-LAX
LAS	MCO	4	LAS-PHX-DFW-HOU-MCO *and four others*
HOU	JFK	2	HOU-ATL-JFK *and two others*

Examples of shortest paths in a graph

Degrees of separation. One of the classic applications of shortest-paths algorithms is to find the *degrees of separation* of individuals in social networks. To fix ideas, we discuss this application in terms of a recently popularized pastime known as the *Kevin Bacon game*, which uses the movie-performer graph that we just considered. Kevin Bacon is a prolific actor who appeared in many movies. We assign every performer who has appeared in a movie a *Bacon number*: Bacon himself is 0, any performer who has been in the same cast as Bacon has a Kevin Bacon number of 1, any other performer (except Bacon) who has been in the same cast as a performer whose number is 1 has a Kevin Bacon number of 2, and so forth. For example, Meryl Streep has a Kevin Bacon number of 1 because she appeared in *The River Wild* with Kevin Bacon. Nicole Kidman's number is 2: although she did not appear in any movie with Kevin Bacon, she was in *Cold Mountain* with Donald Sutherland, and Sutherland appeared in *Animal House* with Kevin Bacon. Given the name of a performer, the simplest version of the game is to find some alternating sequence of movies and performers that lead back to Kevin Bacon. For example, a movie buff might know that Tom Hanks was



as a source, you can wr
a trip. PROGRAM 4.5.3 is
ity for any graph. Thi
numerous queries fror
cost of building a Path
encouraged to explore

Natural encoding of a bitmap

one bit per pixel

```
000000000000000000000000000011111111111111000000000
000000000000000000000000001111111111111111100000000
000000000000000000000001111111111111111111111110000
000000000000000000000011111111111111111111111111000
000000000000000000001111111111111111111111111111110
000000000000000000011111110000000000000000001111111
000000000000000000011111000000000000000000000011111
000000000000000000011100000000000000000000000000111
000000000000000000011100000000000000000000000000111
000000000000000000011100000000000000000000000000111
000000000000000000011100000000000000000000000000111
000000000000000000001111000000000000000000000001110
000000000000000000000011100000000000000000000111000
011111111111111111111111111111111111111111111111111
011111111111111111111111111111111111111111111111111
011111111111111111111111111111111111111111111111111
011111111111111111111111111111111111111111111111111
011111111111111111111111111111111111111111111111111
011000000000000000000000000000000000000000000000011
```

19-by-51 raster of letter 'q' lying on its side

Run-length encoding of a bitmap

to encode number of bits per line

natural encoding. (19 × 51) + 6 = 975 bits.

run-length encoding. (63 × 6) + 6 = 384 bits.

63 6-bit run lengths

```
000000000000000000000000000011111111111111000000000
000000000000000000000000001111111111111111110000000
000000000000000000000001111111111111111111111110000
000000000000000000000011111111111111111111111111000
000000000000000000001111111111111111111111111111110
000000000000000000011111110000000000000000001111111
000000000000000000011111000000000000000000000011111
000000000000000000011100000000000000000000000000111
000000000000000000011100000000000000000000000000111
000000000000000000011100000000000000000000000000111
000000000000000000011100000000000000000000000000111
000000000000000000001111000000000000000000000001110
000000000000000000000011100000000000000000000111000
011111111111111111111111111111111111111111111111111
011111111111111111111111111111111111111111111111111
011111111111111111111111111111111111111111111111111
011111111111111111111111111111111111111111111111111
011111111111111111111111111111111111111111111111111
011000000000000000000000000000000000000000000000011
```

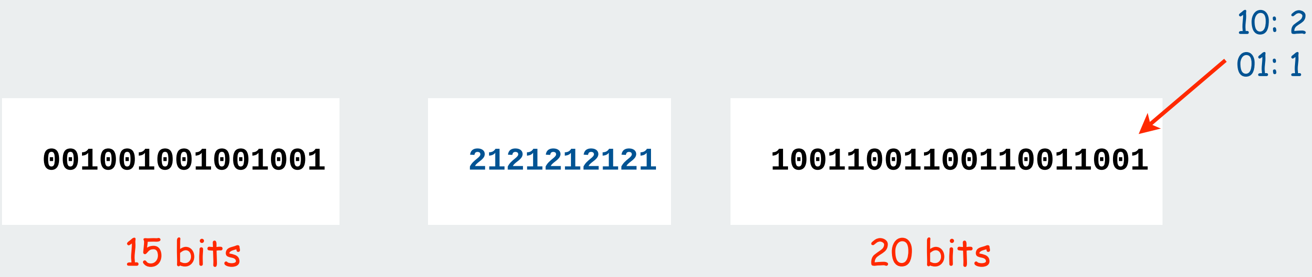
19-by-51 raster of letter 'q' lying on its side

```
51
28 14 9
26 18 7
23 24 4
22 26 3
20 30 1
19 7 18 7
19 5 22 5
19 3 26 3
19 3 26 3
19 3 26 3
19 3 26 3
20 4 23 3 1
22 3 20 3 3
1 50
1 50
1 50
1 50
1 50
1 2 46 2
```

RLE

Run-length encoding

- Exploit long runs of repeated characters.
- Bitmaps: runs alternate between 0 and 1; just output run lengths.
- Issue: how to encode run lengths (!)



001001001001001

15 bits

2121212121

10011001100110011001

20 bits

10: 2
01: 1

- Does not compress when runs are short.

Runs are long in typical applications (such as black-and-white bitmaps).

Run-length encoding and Huffman codes in the wild

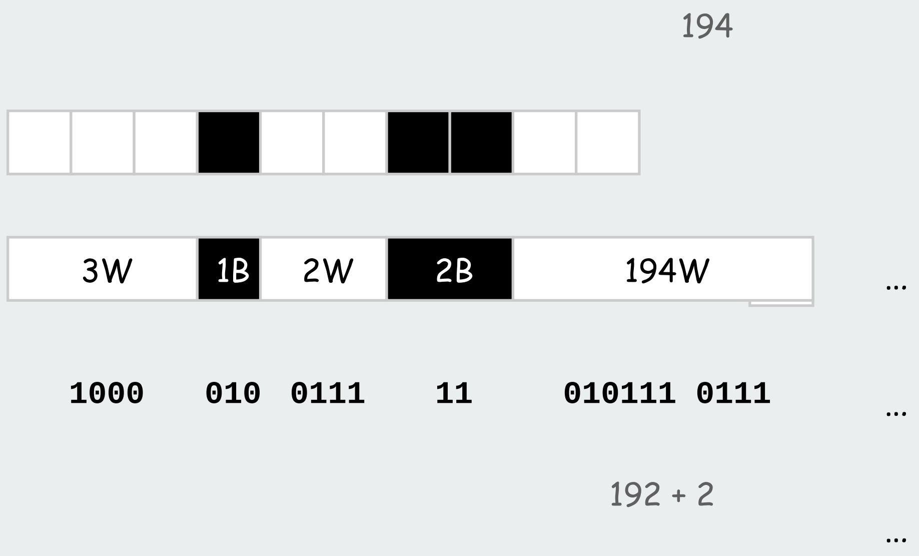
ITU-T T4 Group 3 Fax for black-and-white bitmap images (~1980)

- up to 1728 pixels per line
- typically mostly white.

Step 1. Use run-length encoding.

Step 2. Encode run lengths using two Huffman codes.

one for white and one for black



run	white	black
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
…	…	…
63	00110100	000001100111
64+	11011	0000001111
128+	10010	000011001000
…	…	…
1728+	010011011	0000001100101

Huffman codes built from frequencies in huge sample

BW bitmap compression: another approach

Fax machine (~1980)

- slow scanner produces lines in sequential order
- compress to save time (reduce number of bits to send)

Electronic documents (~2000)

- high-resolution scanners produce huge files
- compress to save space (reduce number of bits to save)

Idea:

- use OCR to get back to ASCII (!)
- use Huffman on ASCII string (!)

Ex. Typical page

- 40 lines, 75 chars/line ~ 3000 chars
- compress to ~ 2000 chars with Huffman code
- reduce file size by a factor of 500 (! ?)

Bottom line: Any extra information about file can yield dramatic gains

- introduction
- basic coding schemes
- an application
- **entropy**
- LZW codes

What data can be compressed?

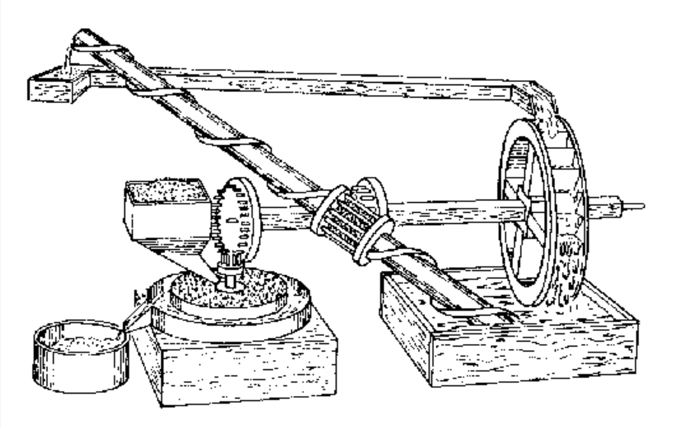
US Patent 5,533,051 on "Methods for Data Compression", which is capable of compression all files.

Slashdot reports of the Zero Space Tuner™ and BinaryAccelerator™.

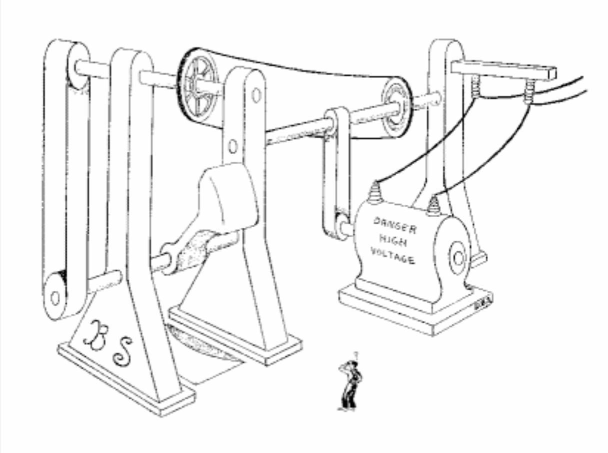
> "ZeoSync has announced a breakthrough in data compression that allows for 100:1 lossless compression of random data. If this is true, our bandwidth problems just got a lot smaller...."

Perpetual Motion Machines

Universal data compression algorithms are the analog of perpetual motion machines.



Closed-cycle mill by Robert Fludd, 1618



Gravity engine by Bob Schadewald

Reference: Museum of Unworkable Devices by Donald E. Simanek
http://www.lhup.edu/~dsimanek/museum/unwork.htm

What data can be compressed?

Theorem. Impossible to losslessly compress all files.

Pf 1.

- consider all 1,000 bit messages.
- 2^{1000} possible messages.
- only $2^{999} + 2^{998} + \ldots + 1$ can be encoded with ≤ 999 bits.
- only 1 in 2^{499} can be encoded with ≤ 500 bits!

Pf 2 (by contradiction).

- given a file M, compress it to get a smaller file M_1.
- compress that file to get a still smaller file M_2.
- continue until reaching file size 0.
- implication: all files can be compressed with 0 bits!

Practical test for any compression algorithm:

- given a file M, compress it to get a (smaller, you hope) file M_1
- compress that file to get a still smaller file M_2.
- continue until file size does not decrease

A difficult file to compress

One million pseudo-random characters (a – p)

```
fclkkacifobjofmkgdcoiicnfmcpcjfccabckjamolnihkbgobcjbngjiceeelpfgcjiihppenefllhglfemdemgahlbpi
ggmllmnefnhjelmgjncjcidlhkglhceninidmmgnobkeglpnadanfbecoonbiehglmpnhkkamdffpacjmgojmcaabpcjce
cplfbgamlidceklhfkkmioljdnoaagiheiapaimlcnlljniggpeanbmojgkccogpmkmoifioeikefjidbadgdcepnhdpfj
aeeapdjeofklpdeghidbgcaiemajllhnndigeihbebifemacfadnknhlbgincpmimdogimgeeomgeljfjgklkdgnhafoho
npjbmlkapddhmepdnckeajebmeknmeejnmenbmnnfefdbhpmigbbjknjmobimamjjaaaffhlhiggaljbaijnebidpaeigd
goghcihodnlhahllhhoojdfacnhadhgkfahmeaebccacgeojgikcoapknlomfignanedmajinlompjoaifiaejbcjcdibp
kofcbmjiobbpdhfilfajkhfmppcngdneeinpnfafaeladbhhifechinknpdnplamackphekokigpddmmjnbngklhibohdf
eaggmclllmdhafkldmimdbplggbbejkcmhlkjocjjlcngckfpfakmnpiaanffdjdlleiniilaenbnikgfnjfcophbgkhdg
mfpoehfmkbpiaignphogbkelphobonmfghpdgmkfedkfkchceeldkcofaldinljjcgafimaanelmfkokcjekefkbmegcgj
ifjcpjppnabldjoaafpbdafifgcoibbcmoffbbgigmngefpkmbhbghlbdjngenldhgnfbdlcmjdmoflhcogfjoldfjpaok
epndejmnbiealkaofifekdjkgedgdlgbioacflfjlafbcaemgpjlagbdgilhcfdcamhfmppfgohjphlmhegjechgdpkklj
pndphfcnnganmbmnggpphnckbieknjhilafkegboilajdppcodpeoddldjfcpialoalfeomjbphkmhnpdmcpgkgeaohfdm
cnegmibjkajcdcpjcpgjminhhakihfgiiachfepffnilcooiciepoapmdjniimfbolchkibkbmhbkgconimkdchahcnhap
fdkiapikencegcjapkjkfljgdlmgncpbakhjidapbldcgeekkjaoihbnbigmhboengpmedliofgioofdcphelapijcegej
gcldcfodikalehbccpbbcfakkblmoobdmdgdkafbbkjnidoikfakjclbchambcpaepfeinmenmpoodadoecbgbmfkkeabi
laoeoggghoekamaibhjibefmoppbhfbhffapjnodlofeihmjahmeipejlfhloefgmjhjnlomapjakhhjpncomippeanbik
khekpcfgbgkmklipfbiikdkdcbolofhelipbkbjmjfoempccneaebklibmcaddlmjdcajpmhhaeedbbfpjafcndianlfcj
mmbfncpdcccodeldhmnbdjmeajmboclkggojghlohlbhgjkhkmclohkgjamfmcchkchmiadjgjhjehflcbklfifackbecg
joggpbkhlcmfhipflhmnmifpjmcoldbeghpcekhgmnahijpabnomnokldjcpppbcpgcjofngmbdcpeeeiiiclmbbmfjkhl
anckidhmbeanmlabncnccpbhoafajjicnfeenppoekmlddholnbdjapbfcajblbooiaepfmmeoafedflmdcbaodgeahimc
gpcammjljoebpfmghogfckgmomecdipmodbcempidfnlcggpgbffoncajpncomalgoiikeolmigliikjkolgolfkdgiijj
iooiokdihjbbofiooibakadjnedlodeeiijkliicnioimablfdpjiafcfineecbafaamheiipegegibioocmlmhjekfikf
effmddhoakllnifdhckmbonbchfhhclecjamjildonjjdpifngbojianpljahpkindkdoanlldcbmlmhjfomifhmncikol
jjhebidjdphpdepibfgdonjljfgifimniipogockpidamnkcpipglafmlmoacjibognbplejnikdoefccdpfkomkimffgj
gielocdemnblimfmbkfbhkelkpfoheokfofochbmifleecbglmnfbnfncjmefnihdcoeiefllemnohlfdcmbdfebdmbeeb
balggfbajdamplphdgiimehglpikbipnkkecekhilchhhfaeafbbfdmcjojfhpponglkfdmhjpcieofcnjgkpibcbiblfp
njlejkcppbhopohdghljlcokhdoahfmlglbdkliajbmnkkfcoklhlelhjhoiginaimgcabcfebmjdnbfhohkjphnklcbhc
jpgbadakoecbkjcaebbanhnfhpnfkfbfpohmnkligpgfkjadomdjjnhlnfailfpcmnololdjekeolhdkebiffebajjpclg
hllmemegncknmkkeoogilijmmkomllbkkabelmodcohdhppdakbelmlejdnmbfmcjdebefnjihnejmnogeeafldabjcgfo
aehldcmkbnbafpciefhlopicifadbppgmfngecjhefnkbjmliodhelhicnfoongngemddepchkokdjafegnpgledakmbcp
cmkckhbffeihpkajginfhdolfnlgnadefamlfocdibhfkiaofeegppcjilndepleihkpkkgkphbnkggjiaolnolbjpobjd
cehglelckbhjilafccfipgebpc....
```

A difficult file to compress

```
public class Rand
{
   public static void main(String[] args)
   {
      for (int i = 0; i < 1000000; i++)
      {
         char c = 'a';
         c += (char) (Math.random() * 16);
         System.out.print(c);
      }
   }
}
```

231 bytes, but output is hard to compress
(assume random seed is fixed)

```
% javac Rand.java
% java Rand  > temp.txt
% compress -c  temp.txt > temp.Z
% gzip     -c  temp.txt > temp.gz
% bzip2    -c  temp.txt > temp.bz2
```

```
% ls -l
    231 Rand.java
1000000 temp.txt
 576861 temp.Z
 570872 temp.gz
 499329 temp.bz2
```

resulting file sizes (bytes)

Information theory

Intrinsic difficulty of compression.

- Short program generates large data file.
- Optimal compression algorithm has to discover program!
- Undecidable problem.

Q. How do we know if our algorithm is doing well?

A. Want lower bound on # bits required by any compression scheme.

Language model

Q. How do compression algorithms work?

A. They exploit statistical biases of input messages.

- ex: white patches occur in typical images.
- ex: ord **`Princeton`** occurs more frequently than **`Yale`**.

Basis of compression: probability.

- Formulate probabilistic model to predict symbols.
  simple: character counts, repeated strings
  complex: models of a human face
- Use model to encode message.
- Use same model to decode message.

Ex. Order 0 Markov model

- R symbols generated independently at random
- probability of occurrence of i th symbol: p_i (fixed).

Entropy

A measure of information. [Shannon, 1948]

$$H(M) = p_0/\lg p_0 + p_1/\lg p_1 + p_2/\lg p_2 + \ldots + p_{R-1}/\lg p_{R-1}$$

- information content of symbol s is proportional to $1/\lg_2 p(s)$.
- weighted average of information content over all symbols.
- interface between coding and model.

Claude Shannon

Ex. 4 binary models (R = 2)

	p_0	p_1	H(M)
1	1/2	1/2	1
2	0.900	0.100	0.469
3	0.990	0.010	0.0808
4	1	0	0

Ex. fair die (R = 6)

p(1)	p(2)	p(3)	p(4)	p(5)	p(6)	H(M)
1/6	1/6	1/6	1/6	1/6	1/6	2.585

Entropy and compression

Theorem. [Shannon, 1948] If data source is an order 0 Markov model, any compression scheme must use ≥ H(M) bits per symbol on average.

- Cornerstone result of information theory.
- Ex: to transmit results of fair die, need ≥ 2.58 bits per roll.

Theorem. [Huffman, 1952] If data source is an order 0 Markov model, Huffman code uses ≤ H(M) + 1 bits per symbol on average.

Q. Is there any hope of doing better than Huffman coding?

A1. Yes. Huffman wastes up to 1 bit per symbol.
if H(M) is close to 0, this difference matters
can do better with "arithmetic coding"

A2. Yes. Source may not be order 0 Markov model.

Entropy of the English Language

Q. How much redundancy is in the English language?

> "... randomising letters in the middle of words [has] little or no effect on the ability of skilled readers to understand the text. This is easy to denmtrasote. In a pubiltacion of New Scnieitst you could ramdinose all the letetrs, keipeng the first two and last two the same, and reibadailty would hadrly be aftcfeed. My ansaylis did not come to much beucase the thoery at the time was for shape and senqeuce retigcionon. Saberi's work sugsegts we may have some pofrweul palrlael prsooscers at work. The resaon for this is suerly that idnetiyfing coentnt by paarllel prseocsing speeds up regnicoiton. We only need the first and last two letetrs to spot chganes in meniang."

A. Quite a bit.

Entropy of the English Language

Q. How much information is in each character of the English language?

Q. How can we measure it?

model = English text

A. [Shannon's 1951 experiment]

- Asked subjects to predict next character given previous text.
- The number of guesses required for right answer:

# of guesses	1	2	3	4	5	≥ 6
Fraction	0.79	0.08	0.03	0.02	0.02	0.05

- Shannon's estimate: about 1 bit per char [0.6 - 1.3].

Compression less than 1 bit/char for English ? If not, keep trying!

- introduction
- basic coding schemes
- an application
- entropy
- LZW codes

Statistical Methods

Static model. Same model for all texts.

- Fast.
- Not optimal: different texts have different statistical properties.
- Ex: ASCII, Morse code.

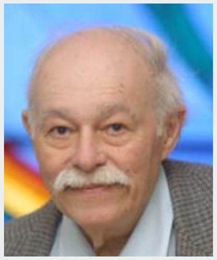
Dynamic model. Generate model based on text.

- Preliminary pass needed to generate model.
- Must transmit the model.
- Ex: Huffman code.

Adaptive model. Progressively learn and update model as you read text.

- More accurate modeling produces better compression.
- Decoding must start from beginning.
- Ex: LZW.

LZW Algorithm



Lempel-Ziv-Welch. [variant of LZ78]

- Create ST associating a fixed-length codeword with some previous substring.
- When input matches string in ST, output associated codeword.
- length of strings in ST grows, hence compression.

To send (encode) M.

- Find longest string s in ST that is a prefix of unsent part of M
- Send codeword associated with s.
- Add s · x to ST, where x is next char in M.

Ex. ST: **a, aa, ab, aba, abb, abaa, abaab, abaaa,**

- unsent part of M: **abaababbb...**
- s = **abaab**, x = **a**.
- Output integer associated with s; insert **abaaba** into ST.

LZW encoding example

input	code	add to ST
a	97	ab
b	98	br
r	114	ra
a	97	ac
c	99	ca
a	97	ad
d	100	da
a		
b	128	abr
r		
a	130	rac
c		
a	132	cad
d		
a	134	dab
b		
r	129	bra
a	97	
STOP	255	

input: 7-bit ASCII
output: 8-bit codewords

ASCII

key	value
	0
	...
a	97
b	98
c	99
d	100
	...
r	114
	...
	127

ST

key	value
ab	128
br	129
ra	130
ac	131
ca	132
ad	133
da	134
abr	135
rac	136
cad	137
dab	138
bra	139
	...
STOP	255

To send (encode) M.

- Find longest string s in ST that is a prefix of unsent part of M
- Send integer associated with s.
- Add s · x to ST, where x is next char in M.

LZW encoding example

input: 7-bit ASCII
19 chars
133 bits

input	code
a	97
b	98
r	114
a	97
c	99
a	97
d	100
a	
b	128
r	
a	130
c	
a	132
d	
a	134
b	
r	129
a	97
STOP	255

output: 8-bit codewords
14 chars
112 bits

Key point: no need to send ST (!)

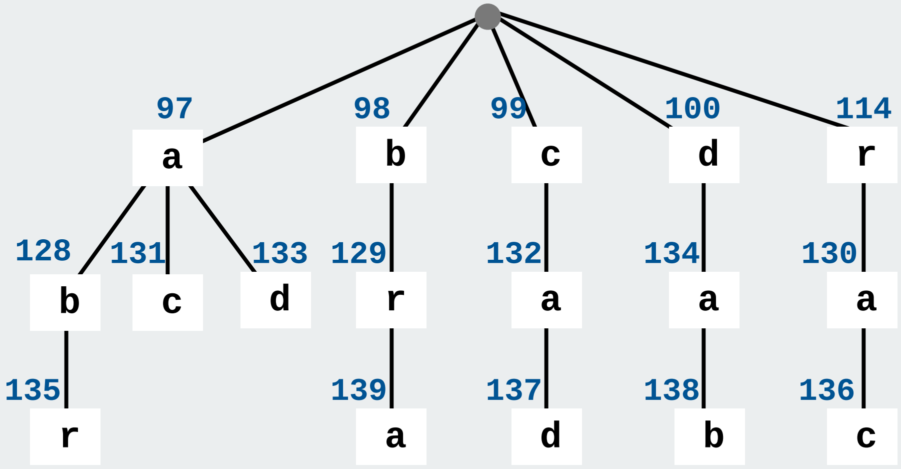
LZW encode ST implementation

Q. How to do longest prefix match?

A. Use a trie for the ST

Encode.

- lookup string suffix in trie.
- output ST index at bottom.
- add new node to bottom of trie.



Note that all substrings are in ST

ASCII

key	value
	0
	...
a	97
b	98
c	99
d	100
	...
r	114
	...
	127

ST

key	value
ab	128
br	129
ra	130
ac	131
ca	132
ad	133
da	134
abr	135
rac	136
cad	137
dab	138
bra	139
	...
STOP	255

LZW encoder: Java implementation

```java
public class LZWEncoder
{
   public static void main(String[] args)
   {
      LookAheadIn in = new LookAheadIn();

      LZWst st = new LZWst();

      while (!in.isEmpty())
      {
         int codeword = st.getput(in);
         StdOut.println(codeword);
      }
   }
}
```

input stream with lookahead

specialized TST

encode text and build TST

postprocess to encode in binary

Use specialized TST

- initialized with ASCII chars and codes
- **getput()** method returns code of longest prefix s
  and adds s + next char to symbol table

Need input stream with backup [stay tuned]

LZW encoder: Java implementation (TST scaffolding)

```
public class LZWst
{
   private int i;
   private int codeword;
   private Node[] roots;

   public LZWst()
   {
      roots = new Node[128];
      for (i = 0; i < 128; i++)
         roots[i] = new Node((char) i, i);
   }

   private class Node
   {
      Node(char c, int codeword)
      {  this.c = c; this.codeword = codeword;  }
      char c;
      Node left, mid, right;
      int codeword;
   }

   public int getput(LookAheadIn in)
   // See next slide.

}
```

next codeword to assign

codeword to return

array of TSTs

initialize
with ASCII

standard
node code

LZW encoder: Java implementation (TST search/insert)

```
public int getput(LookAheadIn in)
{
   char c  = in.readChar();
   if (c == '!') return 255;
   roots[c] = getput(c, roots[c], in);
   in.backup();
   return codeword;
}
```

longest prefix codeword

caution: tricky recursive code

```
public Node getput(char c, Node x, LookAheadIn in)
{
   if (x == null)
   {  x = new Node(c, i++); return x; }

   if      (c < x.c) x.left  = getput(c, x.left, in);
   else if (c > x.c) x.right = getput(c, x.right, in);
   else
                  {
                     char next  = in.readChar();
                     codeword = x.codeword;
                     x.mid =   getput(next, x.mid, in);
                  }
   return x;
}
```

recursive search and insert

check for codeword overflow omitted

LZW encoder: Java implementation (input stream with lookahead)

```
public class LookAheadIn
{
    In in = new In();
    char last;
    boolean backup = false;

    public void backup()
    { backup = true; }

    public char readChar()
    {
      if (!backup)
      {  last = in.readChar();  }
      backup = false;
      return last;
    }

    public boolean isEmpty()
    { return !backup && in.isEmpty();  }
}
```

Provides input stream with one-character lookahead.

`backup()` call means that last **`readChar()`** call was lookahead.

LZW Algorithm

Lempel-Ziv-Welch. [variant of LZ78]

- Create ST and associate an integer with each useful string.
- When input matches string in ST, output associated integer.
- length of strings in ST grows, hence compression.
- decode by rebuilding ST from code

To send (encode) M.

- Find longest string s in ST that is a prefix of unsent part of M
- Send integer associated with s.
- Add s · x to ST, where x is next char in M.

To decode received message to M.

- Let s be ST entry associated with received integer
- Add s to M.
- Add p · x to ST, where x is first char in s, p is previous value of s.

LZW decoding example

codeword	output	add to ST
97	a	
98	b	ab
114	r	br
97	a	ra
99	c	ac
97	a	ca
100	d	ad
128	a	
	b	da
130	r	
	a	abr
132	c	
	a	rac
134	d	
	a	cad
129	b	
	r	dab
97	a	bra
255	STOP	

role of keys and values switched

key	value
0	
...	
97	a
98	b
99	c
100	d
...	
114	r
...	
127	

key	value
128	ab
129	br
130	ra
131	ac
132	ca
133	ad
134	da
135	abr
136	rac
137	cad
138	dab
139	bra
...	
255	

Use an array
to implement ST

To decode received message to M.

- Let s be ST entry associated with received integer
- Add s to M.
- Add p · x to ST, where x is first char in s, p is previous value of s.

LZW decoder: Java implementation

```
public class LZWDecoder
{
   public static void main(String[] args)
   {
      String[] st = new String[256];
      int i;
      for (i = 0; i < 128; i++)
      {  st[i] = Character.toString((char) i);  }
      st[255] = "!";

      String prev = "";
      while (!StdIn.isEmpty())
      {
         int codeword = StdIn.readInt();
         String s;
         if (codeword == i) // Tricky situation!
              s = prev + prev.charAt(0);
         else s = st[codeword];
         StdOut.print(s);
         if (prev.length() > 0)
         {  st[i++] = prev + s.charAt(0);  }
         prev = s;
      }
      StdOut.println();
   }
}
```

initialize ST with ASCII

preprocess to decode from binary

decode text and build ST

Ex: **ababababab**

LZW decoding example (tricky situation)

input	code	add to ST
a	97	ab
b	98	ba
a		
b	128	aba
a		
b		
a	130	abab
b		
STOP	255	

key	value
128	ab
129	ba
130	aba
131	abab
...	
255	

codeword	output	add to ST
97	a	
98	b	ab
128	a	
	b	ba
130	a	
	b	
	a	aba
98	b	
255	STOP	

needed before added to ST!

To send (encode) M.

- Find longest prefix
- Send integer associated with s.
- Add s · x to ST, where
  x is next char in M.

To decode received message to M.

- Let s be ST entry for integer
- Add s to M.
- Add p · x to ST where
  x is first char in s
  p is previous value of s.

LZW implementation details

How big to make ST?

- how long is message?
- whole message similar model?
- ...
- [many variations have been developed]

What to do when ST fills up?

- throw away and start over. GIF
- throw away when not effective. Unix compress
- ...
- [many other variations]

Why not put longer substrings in ST?

- ...
- [many variations have been developed]

LZW in the real world

Lempel-Ziv and friends.

- LZ77.
- LZ78.
- LZW.
- Deflate = LZ77 variant + Huffman.

LZ77 not patented ⇒ widely used in open source
LZW patent #4,558,302 expired in US on June 20, 2003
some versions copyrighted

PNG: LZ77.

Winzip, gzip, jar: deflate.

Unix compress: LZW.

Pkzip: LZW + Shannon-Fano.

GIF, TIFF, V.42bis modem: LZW.

Google: zlib which is based on deflate.

never expands a file

Lossless compression ratio benchmarks

Calgary corpus: standard data compression benchmark

Year	Scheme	Bits / char
1967	ASCII	7.00
1950	Huffman	4.70
1977	LZ77	3.94
1984	LZMW	3.32
1987	LZH	3.30
1987	Move-to-front	3.24
1987	LZB	3.18
1987	Gzip	2.71
1988	PPMC	2.48
1988	SAKDC	2.47
1994	PPM	2.34
1995	Burrows-Wheeler	2.29
1997	BOA	1.99
1999	RK	1.89

← next assignment

Entropy	Bits/char
Char by char	4.5
8 chars at a time	2.4
Asymptotic	1.3

Data compression summary

Lossless compression.

- Represent fixed length symbols with variable length codes. [Huffman]
- Represent variable length symbols with fixed length codes. [LZW]

Lossy compression. [not covered in this course]

- JPEG, MPEG, MP3.
- FFT, wavelets, fractals, SVD, …

Limits on compression. Shannon entropy.

Theoretical limits closely match what we can achieve in practice.

Practical compression: Use extra knowledge whenever possible.

Butch: I don't mean to be a sore loser, but when it's done, if I'm dead, kill him.
Sundance: Love to.
Butch: No, no, not yet. Not until me and Harvey get the rules straightened out.
Harvey: Rules? In a knife fight? No rules.
Butch: Well, if there ain't going to be any rules, let's get the fight started...

Pattern Matching

- exact pattern matching
- Knuth-Morris-Pratt
- RE pattern matching
- grep

References:
Algorithms in C (2nd edition), Chapter 19
http://www.cs.princeton.edu/introalgsds/63long
http://www.cs.princeton.edu/introalgsds/72regular

- exact pattern matching
- Knuth-Morris-Pratt
- RE pattern matching
- grep

Exact pattern matching

Problem:

Find first match of a pattern of length M in a text stream of length N.

typically N >> M

pattern

```
n e e d l e
```

M = 6

text

```
i n a h a y s t a c k a n e e d l e i n a
```

N = 21

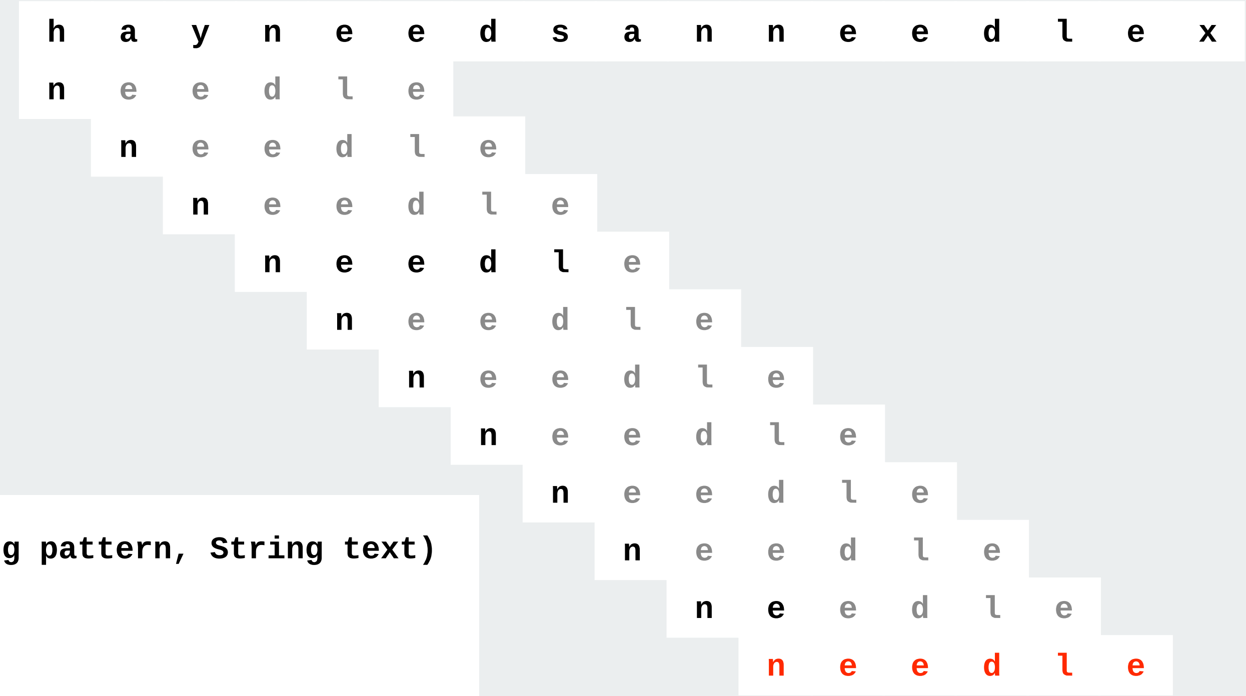
Applications.

- parsers.
- spam filters.
- digital libraries.
- screen scrapers.
- word processors.
- web search engines.
- natural language processing.
- computational molecular biology.
- feature detection in digitized images.

. . .

Brute-force exact pattern match

Check for pattern starting at each text position.

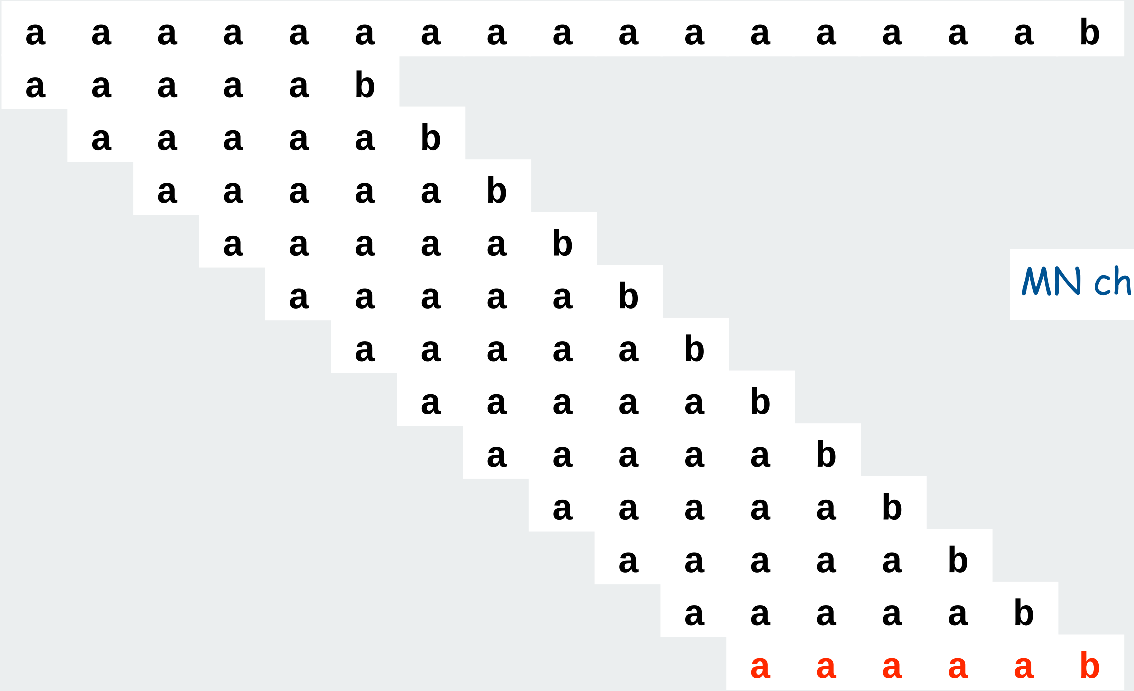


```
h a y n e e d s a n n e e d l e x
n e e d l e
  n e e d l e
    n e e d l e
      n e e d l e
        n e e d l e
          n e e d l e
            n e e d l e
              n e e d l e
                n e e d l e
                  n e e d l e
                    n e e d l e
```

```
public static int search(String pattern, String text)
{
   int M = pattern.length();
   int N = text.length();

   for (int i = 0; i < N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (text.charAt(i+j) != pattern.charAt(j))
            break;
      if (j == M) return i;  ←  pattern start index in text
   }
   return -1;  ←  not found
}
```

Brute-force exact pattern match: worst case

Brute-force algorithm can be slow if text and pattern are repetitive



```
a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  b
a  a  a  a  a  b
   a  a  a  a  a  b
      a  a  a  a  a  b
         a  a  a  a  a  b
            a  a  a  a  a  b
               a  a  a  a  a  b
                  a  a  a  a  a  b
                     a  a  a  a  a  b
                        a  a  a  a  a  b
                           a  a  a  a  a  b
                              a  a  a  a  a  b
                                 a  a  a  a  a  b
```

text length N

MN char compares

pattern length M

but this situation is rare in typical applications

Hence, the `indexOf()` method in Java's `String` class uses brute-force

Exact pattern matching in Java

Exact pattern matching is implemented in Java's **`String`** class

`s.indexOf(t, i)`: index of first occurrence of pattern **`t`** in string s, starting at offset **`i`**.

Ex: Screen scraping. Exact match to extract info from website

```
public class StockQuote
{
   public static void main(String[] args)
   {
      String name = "http://finance.yahoo.com/q?s=";
      In in = new In(name + args[0]);
      String input = in.readAll();
      int start    = input.indexOf("Last Trade:", 0);
      int from     = input.indexOf("<b>",  start);
      int to       = input.indexOf("</b>", from);
      String price = input.substring(from + 3, to);
      System.out.println(price);
   }
}
```

```
% java StockQuote goog
688.04

% java StockQuote msft
33.75
```

http://finance.yahoo.com/q?s=goog

```
...
<tr>
<td class= "yfnc_tablehead1"
width= "48%">
Last Trade:
</td>
<td class= "yfnc_tabledata1">
<big><b>688.04</b></big>
</td></tr>
<td class= "yfnc_tablehead1"
width= "48%">
Trade Time:
</td>
<td class= "yfnc_tabledata1">
```

Algorithmic challenges in pattern matching

Brute-force is not good enough for all applications

Theoretical challenge: Linear-time guarantee. ← fundamental algorithmic problem

Practical challenge: Avoid backup in text stream. ← often no room or time to save text

Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for each good person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their attack at dawn party. Now is the time for each person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.

- exact pattern matching
- **Knuth-Morris-Pratt**
- RE pattern matching
- grep

Knuth-Morris-Pratt (KMP) exact pattern-matching algorithm

Classic algorithm that meets both challenges

- linear-time guarantee
- no backup in text stream

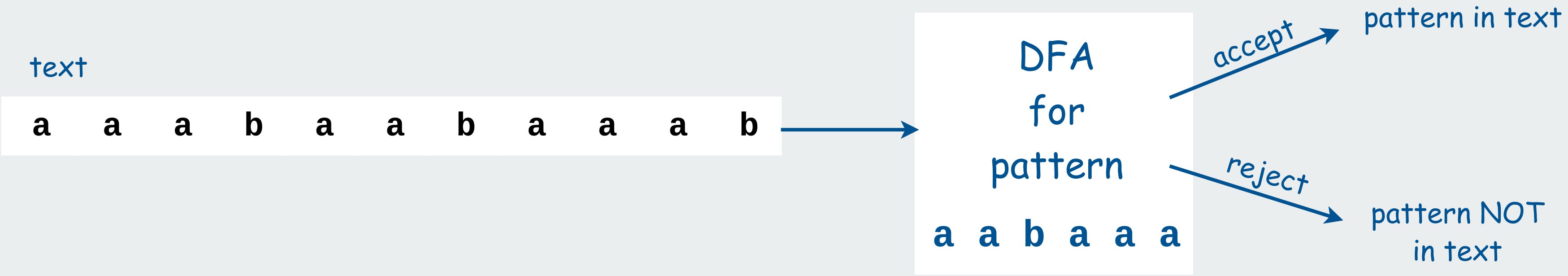
Don Knuth

Jim Morris

Vaughan Pratt

Basic plan (for binary alphabet)

- build DFA from pattern
- simulate DFA with text as input



No backup in a DFA

Linear-time because each step is just a state change

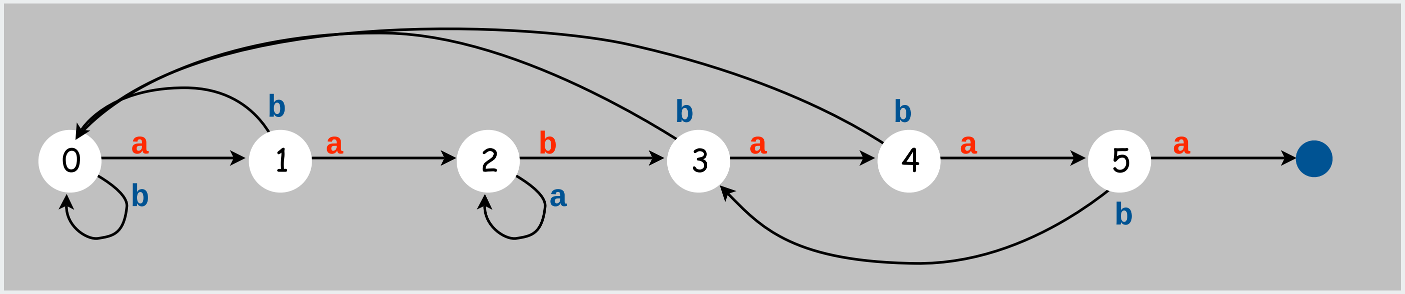
Knuth-Morris-Pratt DFA example

One state for each pattern character

- Match input character: move from i to i+1
- Mismatch: move to previous state

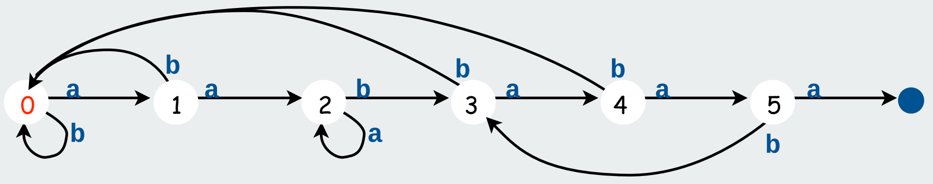
DFA
for
pattern

a a b a a a

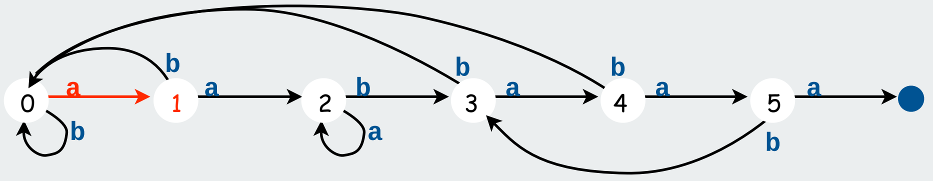


How to construct? Stay tuned

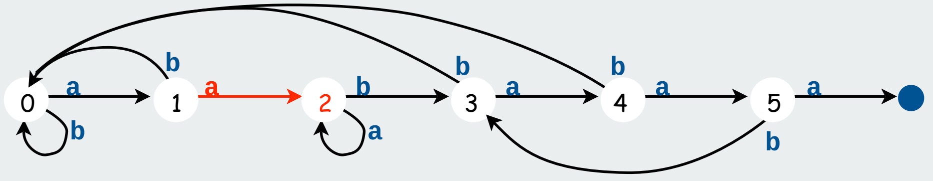
Knuth-Morris-Pratt DFA simulation



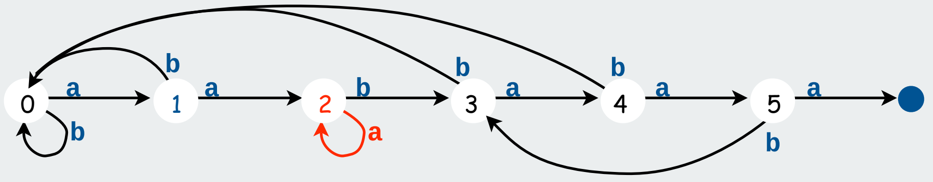
0 a a a b a a b a a a b



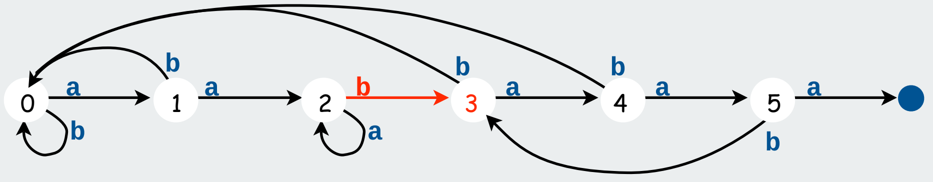
1 a a a b a a b a a a b



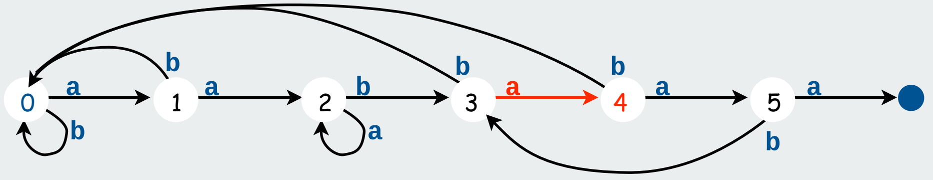
2 a a a b a a b a a a b



2 a a a b a a b a a a b



3 a a a b a a b a a a b



Knuth-Morris-Pratt DFA simulation

4 a a a b a a b a a a b

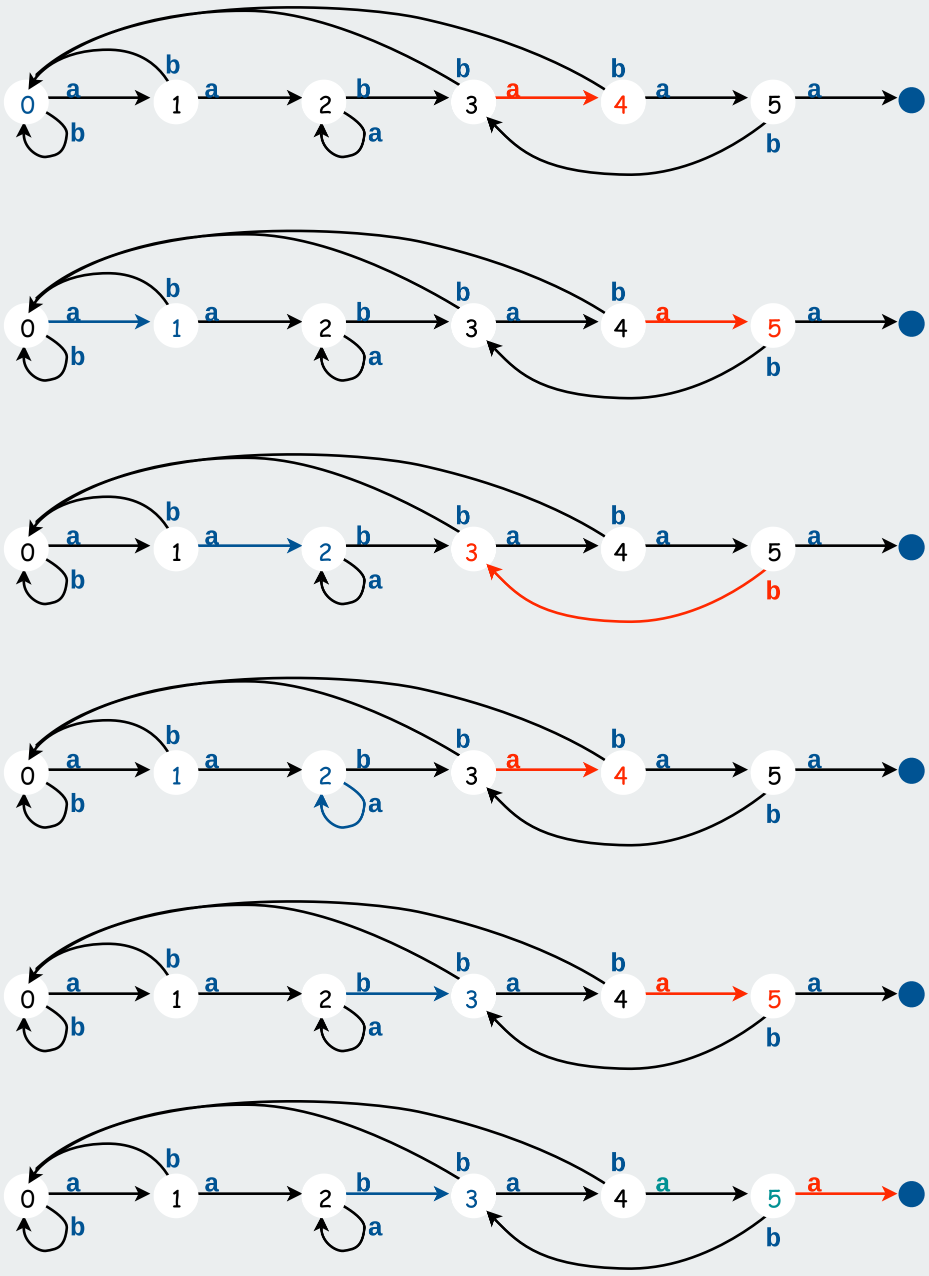
5 a a a b a a b a a a b

3 a a a b a a b a a a b

4 a a a b a a b a a a b

5 a a a b a a b a a a b

accept!



Knuth-Morris-Pratt DFA simulation

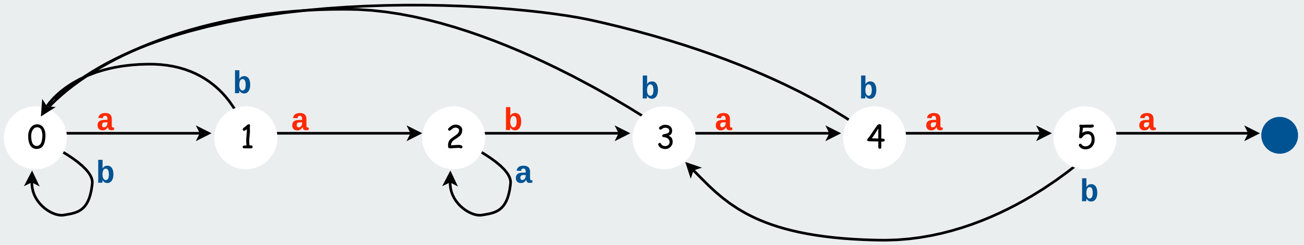
When in state i:

- have found match in i previous input chars
- that is the longest such match

Ex. End in state 4 iff text ends in **aaba**.

Ex. End in state 2 iff text ends in **aa** (but not **aabaa** or **aabaaa**).

```
0  a a a b a a b a a a b
1  a a a b a a b a a a b
2  a a a b a a b a a a b
2  a a a b a a b a a a b
3  a a a b a a b a a a b
4  a a a b a a b a a a b
5  a a a b a a b a a a b
3  a a a b a a b a a a b
4  a a a b a a b a a a b
5  a a a b a a b a a a b
   a a a b a a b a a a b
```



KMP implementation

DFA representation: a single state-indexed array `next[]`

- Upon character match in state `j`, go forward to state `j+1`.
- Upon character mismatch in state `j`, go back to state `next[j]`.

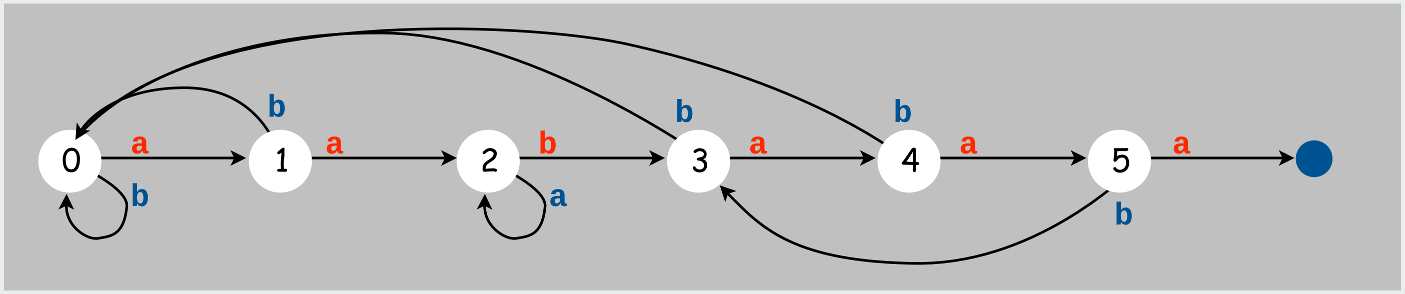
	0	1	2	3	4	5
a	1	2	2	4	5	6
b	0	0	3	0	0	3

DFA for pattern

a a b a a a →

next	0	0	2	0	0	3

← only need to store mismatches



KMP implementation

Two key differences from brute-force implementation:

- Text pointer `i` never decrements
- Need to precompute `next[]` table (DFA) from pattern.

```
int j = 0;
for (int i = 0; i < N; i++)
{
   if (t.charAt(i) == p.charAt(j)) j++;   // match
   else j = next[j];                      // mismatch
   if (j == M) return i - M + 1;          // found
}
return -1;                                // not found
```

Simulation of KMP DFA

Knuth-Morris-Pratt: Iterative DFA construction

DFA for first i states contains the information needed to build state i+1

Ex: given DFA for pattern **aabaaa**.
how to compute DFA for pattern **aabaaab** ?

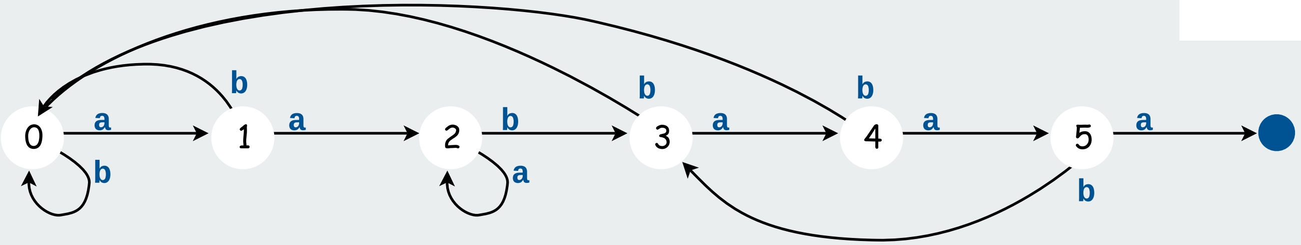
Key idea

- on mismatch at 7th char, need to simulate 6-char backup
- previous 6 chars are known (**abaaaa** in example)
- 6-state DFA (known) determines next state!

Keep track of DFA state for start at 2nd char of pattern

- compare char at that position with next pattern char
- match/mismatch provides all needed info

0	a	b	a	a	a	a
1	a	b	a	a	a	a
0	a	b	a	a	a	a
1	a	b	a	a	a	a
2	a	b	a	a	a	a
2	a	b	a	a	a	a
2	a	b	a	a	a	a



KMP iterative DFA construction: two cases

Let X be the next state in the simulation and j the next state to build.

If p[X] and p[j] match, copy and increment

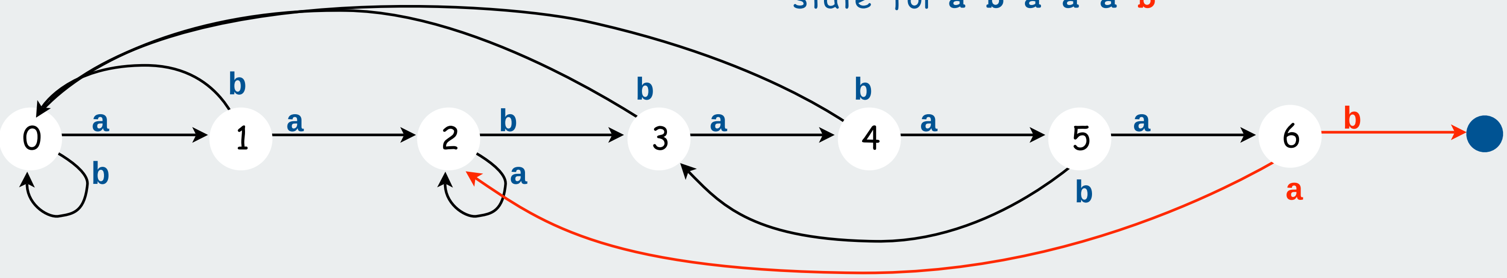
```
next[j] = next[X];
X = X+1
```

	0	1	2	3	4	5	6
p[]	a	a	b	a	a	a	b
next[]	0	0	2	0	0	3	2

↑ x ↑ X ↑ j

state for a b a a a b ↗ X

DFA for a a b a a a b



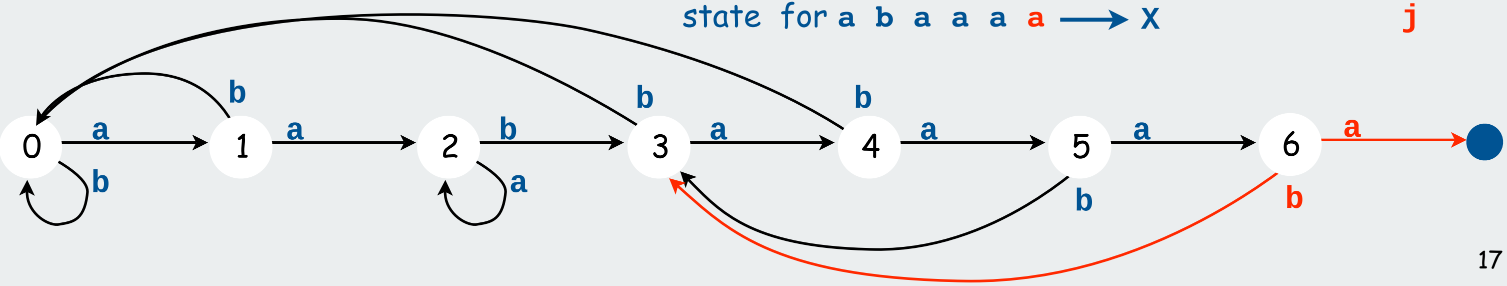
If p[X] and p[j] mismatch, do the opposite

```
next[j] = X+1;
X = next[X];
```

	0	1	2	3	4	5	6
p[]	a	a	b	a	a	a	a
next[]	0	0	2	0	0	3	3

state for a b a a a a → X ↑ j

DFA for a a b a a a a



Knuth-Morris-Pratt DFA construction

DFA

```
0
a
0
```

```
0 1
a a
0 0
↑ ↑
X j
```

match

```
0 1 2
a a b
0 0 2
  ↑ ↑
```

mismatch

```
0 1 2 3
a a b a
0 0 2 0
↑     ↑
```

match

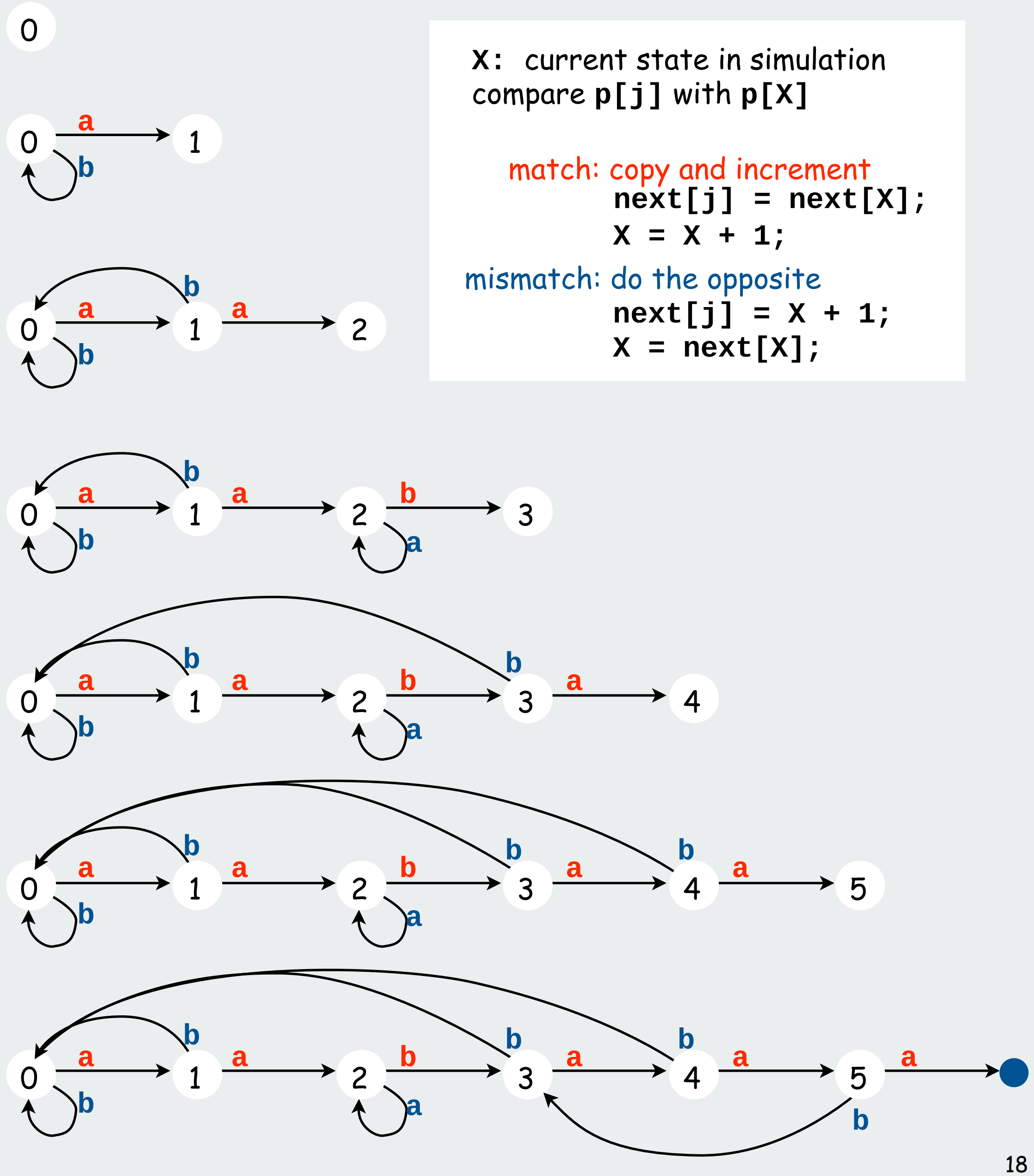
```
0 1 2 3 4
a a b a a
0 0 2 0 0
  ↑     ↑
```

match

```
0 1 2 3 4 5
a a b a a a
0 0 2 0 0 3
    ↑     ↑
```

mismatch

p[1..j-1]	X
	0
a	1
a b	0
a b a	1
a b a a	2



0
0 a 1 b
0 a 1 a 2 b b
0 a 1 a 2 b 3 b a b
0 a 1 a 2 b 3 a 4 b a b b
0 a 1 a 2 b 3 a 4 a 5 b a b b b
0 a 1 a 2 b 3 a 4 a 5 a b a b b b b

X: current state in simulation
compare **p[j]** with **p[X]**

match: copy and increment

```
next[j] = next[X];
X = X + 1;
```

mismatch: do the opposite

```
next[j] = X + 1;
X = next[X];
```

Knuth-Morris-Pratt DFA construction examples

```
        ex: a a b a a a b

        0
        a
        0

        0 1
        a a
next[]  0 0            match
        ↑ ↑
        X j

        0 1 2
        a a b          mismatch
        0 0 2
          ↑ ↑

        0 1 2 3
        a a b a        match
        0 0 2 0
        ↑     ↑

        0 1 2 3 4
        a a b a a      match
        0 0 2 0 0
          ↑     ↑

        0 1 2 3 4 5
        a a b a a a    mismatch
        0 0 2 0 0 3
            ↑     ↑

        0 1 2 3 4 5 6
        a a b a a a b  match
        0 0 2 0 0 3 2
            ↑       ↑
```

```
ex: a b b a b b b

0
a
0

0 1
a b            mismatch
0 1
↑ ↑
X j

0 1 2
a b b          mismatch
0 1 1
↑   ↑

0 1 2 3
a b b a        match
0 1 1 0
↑     ↑

0 1 2 3 4
a b b a b      match
0 1 1 0 1
  ↑     ↑

0 1 2 3 4 5
a b b a b b    match
0 1 1 0 1 1
    ↑     ↑

0 1 2 3 4 5 6
a b b a b b b  mismatch
0 1 1 0 1 1 4
      ↑     ↑
```

X: current state in simulation
compare **p[j]** with **p[X]**

match: copy and increment

```
next[j] = next[X];
X = X + 1;
```

mismatch: do the opposite

```
next[j] = X + 1;
X = next[X];
```

DFA construction for KMP: Java implementation

Takes time and space proportional to pattern length.

```
int X = 0;
int[] next = new int[M];
for (int j = 1; j < M; j++)
{
   if (p.charAt(X) == p.charAt(j))
   {  // match
      next[j] = next[X];
      X = X + 1;
   }
   else
   {  // mismatch
      next[j] = X + 1;
      X = next[X];
   }
}
```

DFA Construction for KMP (assumes binary alphabet)

Optimized KMP implementation

Ultimate search program for any given pattern:

- one statement comparing each pattern character to next
- match: proceed to next statement
- mismatch: go back as dictated by DFA
- translates to machine language (three instructions per pattern char)

```
int kmpsearch(char t[])
{
   int i = 0;
   s0: if (t[i++] != 'a') goto s0;
   s1: if (t[i++] != 'a') goto s0;
   s2: if (t[i++] != 'b') goto s2;
   s3: if (t[i++] != 'a') goto s0;
   s4: if (t[i++] != 'a') goto s0;
   s5: if (t[i++] != 'a') goto s3;
   s6: if (t[i++] != 'b') goto s2;
   s7: if (t[i++] != 'b') goto s4;
   return i - 8;
}
```

↑ pattern[]

↑ next[]

assumes pattern is in text
(o/w use sentinel)

Lesson: Your computer is a DFA!

KMP summary

General alphabet

- more difficult
- easy with **next[][]** indexed by mismatch position, character
- KMP paper has ingenious solution that is not difficult to implement
  [build NFA, then prove that it finishes in 2N steps]

Bottom line: linear-time pattern matching is possible (and practical)

Short history:

- inspired by esoteric theorem of Cook
  [linear time 2-way pushdown automata simulation is possible]
- discovered in 1976 independently by two theoreticians and a hacker
  Knuth: discovered linear time algorithm
  Pratt: made running time independent of alphabet
  Morris: trying to build a text editor.
- theory meets practice

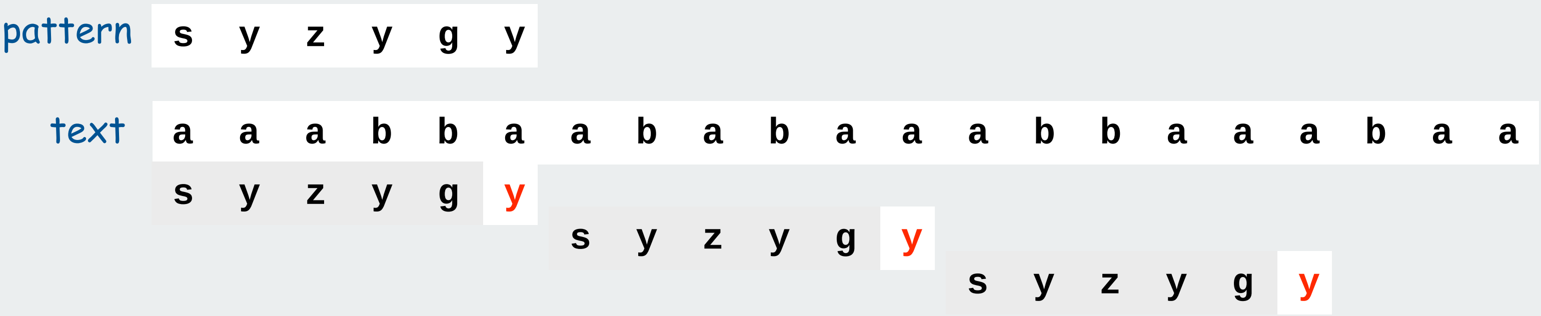
Exact pattern matching: other approaches

Rabin-Karp: make a digital signature of the pattern

- hashing without the table
- linear-time probabilistic guarantee
- plus: extends to 2D patterns
- minus: arithmetic ops much slower than char comparisons

Boyer-Moore: scan from right to left in pattern

- main idea: can skip M text chars when finding one not in the pattern
- needs additional KMP-like heuristic
- plus: possibility of sublinear-time performance (~ N/M)
- used in Unix, emacs



```
pattern  s y z y g y

text     a a a b b a a b a b a a a b b a a a b a a
         s y z y g y
                     s y z y g y
                                 s y z y g y
```

Exact pattern match cost summary

Cost of searching for M-character pattern in N-character text

algorithm	typical	worst-case
brute-force	1.1 N char compares †	M N char compares
Karp-Rabin	3N arithmetic ops	3N arithmetic ops ‡
KMP	1.1 N char compares †	2N char compares
Boyer-Moore	~ N/M char compares †	3N char compares

† assumes appropriate model
‡ randomized

- exact pattern matching
- Knuth-Morris-Pratt
- **RE pattern matching**
- grep

Regular-expression pattern matching

Exact pattern matching:

Search for occurrences of a single pattern in a text file.

Regular expression (RE) pattern matching:

Search for occurrences of one of multiple patterns in a text file.

Ex. (genomics)

- Fragile X syndrome is a common cause of mental retardation.
- human genome contains triplet repeats of **`cgg`** or **`agg`** bracketed by **`gcg`** at the beginning and **`ctg`** at the end
- number of repeats is variable, and correlated with syndrome
- use regular expression to specify pattern: **`gcg(cgg|agg)*ctg`**
- do RE pattern match on person's genome to detect Fragile X

pattern (RE) **`gcg(cgg|agg)*ctg`**

text **`gcggcgtgtgtgcgagagagtgggtttaaagctggcgcggaggcggctggcgcggaggctg`**

RE pattern matching: applications

Test if a string matches some pattern.

- Process natural language.
- Scan for virus signatures.
- Search for information using Google.
- Access information in digital libraries.
- Retrieve information from Lexis/Nexis.
- Search-and-replace in a word processors.
- Filter text (spam, NetNanny, Carnivore, malware).
- Validate data-entry fields (dates, email, URL, credit card).
- Search for markers in human genome using PROSITE patterns.

Parse text files.

- Compile a Java program.
- Crawl and index the Web.
- Read in data stored in ad hoc input file format.
- Automatically create Java documentation from Javadoc comments.

Regular expression examples

A regular expression is a notation to specify a set of strings.

operation	example RE	in set	not in set
concatenation	`aabaab`	`aabaab`	every other string
wildcard	`.u.u.u.`	`cumulus` `jugulum`	`succubus` `tumultuous`
union	`aa | baab`	`aa` `baab`	every other string
closure	`ab*a`	`aa` `abbba`	`ab` `ababa`
parentheses	`a(a|b)aab`	`aaaab` `abaab`	every other string
	`(ab)*a`	`a` `ababababa`	`aa` `abbba`

Regular expression examples (continued)

Notation is surprisingly expressive

regular expression	in set	not in set
`.*spb.*` contains the trigraph **spb**	`raspberry` `crispbread`	`subspace` `subspecies`

`a* | (a*ba*ba*ba*)*`
number of **b**'s is a multiple of 3
	`bbb`
	`aaa`
	`bbbaababbaa`
		`b`
		`bb`
		`baabbbaa`

`.*0....`
fifth to last digit is **0**
	`1000234`
	`98701234`
		`111111111`
		`403982772`

`gcg(cgg|agg)*ctg`
fragile X syndrome indicator
	`gcgctg`
	`gcgcggctg`
	`gcgcggaggctg`
		`gcgcgg`
		`cggcggcggctg`
		`gcgcaggctg`

and plays a well-understood role in the theory of computation

Generalized regular expressions

Additional operations are often added

- Ex: `[a-e]+` is shorthand for `(a|b|c|d|e)(a|b|c|d|e)*`
- for convenience only
- need to be alert for non-regular additions (Ex: Java /)

operation	example	in set	not in set
one or more	`a(bc)+de`	`abcde` `abcbcde`	`ade` `bcde`

character classes	`[A-Za-z][a-z]*`	`word`
		`Capitalized`
			`camelCase`
			`4illegal`

exactly k	`[0-9]{5}-[0-9]{4}`	`08540-1321`
		`19072-5541`
			`111111111`
			`166-54-111`

negations	`[^aeiou]{6}`	`rhythm`	`decade`

Regular expressions in Java

RE pattern matching is implemented in Java's **`String`** class

- basic: **`match()`** method
- various other methods also available (stay tuned)

Ex: Validity checking. Is **`input`** in the set described by the **`re`**?

```
public class Validate
{
   public static void main(String[] args)
   {
      String re    = args[0];
      String input = args[1];
      System.out.println(input.matches(re));
   }
}
```

```
% java Validate "..oo..oo." bloodroot
true

% java Validate "[$_A-Za-z][$_A-Za-z0-9]*" ident123
true

% java Validate "[a-z]+@([a-z]+\.)+(edu|com)" rs@cs.princeton.edu
true

% java Validate "[0-9]{3}-[0-9]{2}-[0-9]{4}" 166-11-4433
true
```

need help solving crosswords?

legal Java identifier

valid email address (simplified)

Social Security number

Regular expressions in other languages

Broadly applicable programmer's tool.

- originated in UNIX in the 1970s
- many languages support extended regular expressions
- built into grep, awk, emacs, Perl, PHP, Python, JavaScript

```
grep NEWLINE */*.java
```

print all lines containing `NEWLINE` which occurs in any file with a `.java` extension

```
egrep '^[qwertyuiop]*[zxcvbnm]*$' dict.txt | egrep '...........'
```

PERL. Practical Extraction and Report Language.

```
perl -p -i -e 's|from|to|g' input.txt
```

replace all occurrences of `from` with `to` in the file `input.txt`

```
perl -n -e 'print if /^[A-Za-z][a-z]*$/' dict.txt
```

↑ do for each line

Regular expression caveat

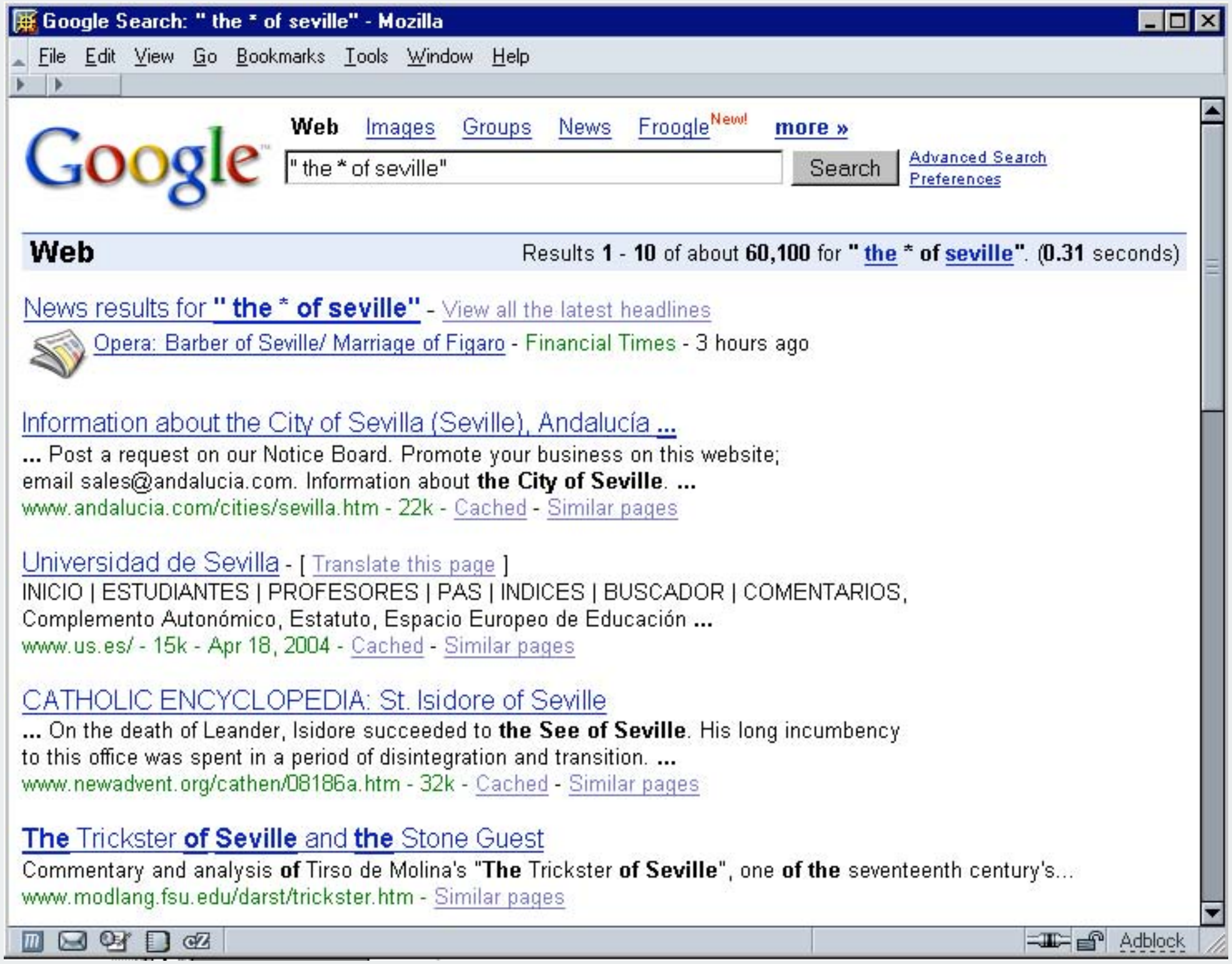
Writing a RE is like writing a program.

- need to understand programming model
- can be easier to write than read
- can be difficult to debug

> "Sometimes you have a programming problem
> and it seems like the best solution is to use
> regular expressions; now you have two problems."

Can the average web surfer learn to use REs?

Google. Supports * for full word wildcard and | for union.



Google Search: " the * of seville" - Mozilla
File Edit View Go Bookmarks Tools Window Help
Google
Web Images Groups News Froogle New! more »
" the * of seville"
Search
Advanced Search
Preferences
Adblock

Web Results **1** - **10** of about **60,100** for **" the * of seville"**. (**0.31** seconds)

News results for **" the * of seville"** - View all the latest headlines

Opera: Barber of Seville/ Marriage of Figaro - Financial Times - 3 hours ago

Information about the City of Sevilla (Seville), Andalucía ...
... Post a request on our Notice Board. Promote your business on this website; email sales@andalucia.com. Information about **the City of Seville**. ...
www.andalucia.com/cities/sevilla.htm - 22k - Cached - Similar pages

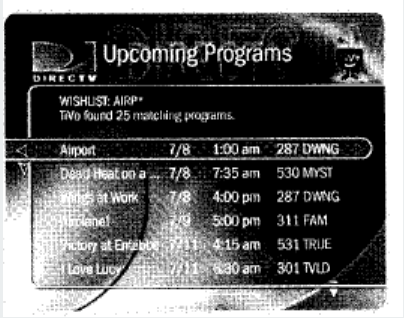
Universidad de Sevilla - [Translate this page]
INICIO | ESTUDIANTES | PROFESORES | PAS | INDICES | BUSCADOR | COMENTARIOS, Complemento Autonómico, Estatuto, Espacio Europeo de Educación ...
www.us.es/ - 15k - Apr 18, 2004 - Cached - Similar pages

CATHOLIC ENCYCLOPEDIA: St. Isidore of Seville
... On the death of Leander, Isidore succeeded to **the See of Seville**. His long incumbency to this office was spent in a period of disintegration and transition. ...
www.newadvent.org/cathen/08186a.htm - 32k - Cached - Similar pages

The Trickster **of Seville** and **the** Stone Guest
Commentary and analysis **of** Tirso de Molina's "**The** Trickster **of Seville**", one **of the** seventeenth century's...
www.modlang.fsu.edu/darst/trickster.htm - Similar pages

Can the average TV viewer learn to use REs?

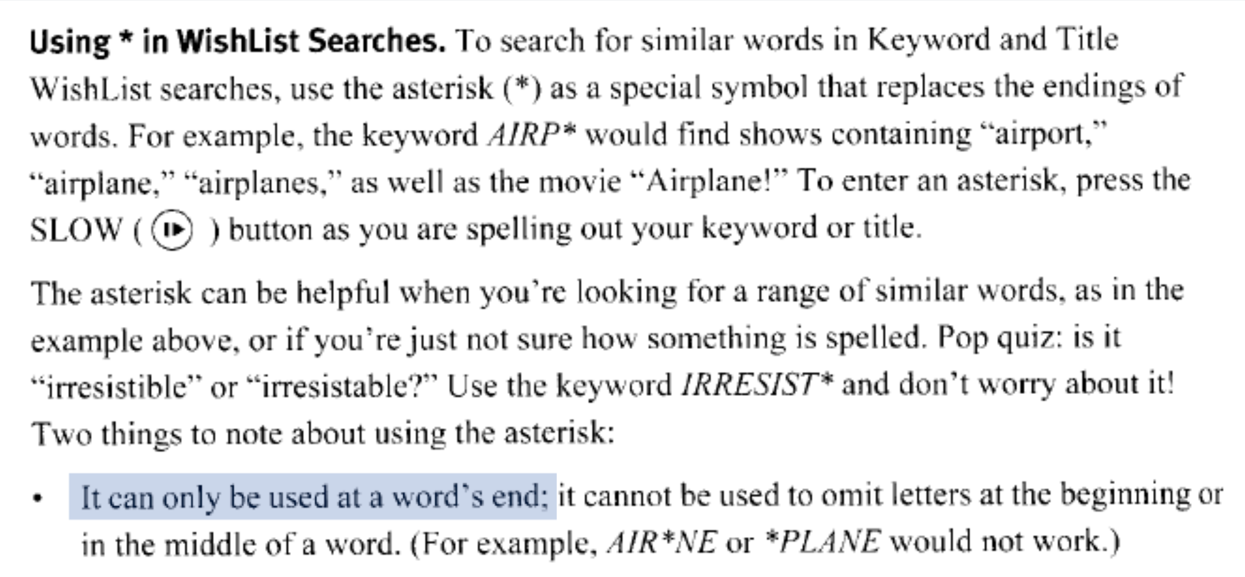
TiVo. WishList has very limited pattern matching.



Upcoming Programs

WISHLIST: AIRP*
TiVo found 25 matching programs.

Airport	7/8	1:00 am	287 DWNG
Dead Heat on a ...	7/8	7:35 am	530 MYST
Wings at Work	7/8	4:00 pm	287 DWNG
Airplane!	7/9	5:00 pm	311 FAM
Victory at Entebbe	7/11	4:15 am	531 TRUE
I Love Lucy	7/11	6:30 am	301 TVLD



Using * in WishList Searches. To search for similar words in Keyword and Title WishList searches, use the asterisk (*) as a special symbol that replaces the endings of words. For example, the keyword *AIRP** would find shows containing “airport,” “airplane,” “airplanes,” as well as the movie “Airplane!” To enter an asterisk, press the SLOW (⏵) button as you are spelling out your keyword or title.

The asterisk can be helpful when you’re looking for a range of similar words, as in the example above, or if you’re just not sure how something is spelled. Pop quiz: is it “irresistible” or “irresistable?” Use the keyword *IRRESIST** and don’t worry about it! Two things to note about using the asterisk:

- It can only be used at a word’s end; it cannot be used to omit letters at the beginning or in the middle of a word. (For example, *AIR*NE* or **PLANE* would not work.)

Reference: page 76, Hughes DirectTV TiVo manual

Can the average programmer learn to use REs?

Perl RE for Valid RFC822 Email Addresses

Reference: http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)
?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
 \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t
]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031
]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](
?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?
:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?
:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?
:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|
\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>
@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"
(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?
:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[
\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(
?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;
:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([
^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\"
.\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\
]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\
[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\
r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
```

"Implementing validation with regular expressions somewhat pushes the limits of what it is sensible to do with regular expressions, although Perl copes well."

37 more lines
↓

- exact pattern matching
- Knuth-Morris-Pratt
- RE pattern matching
- grep

GREP implementation: basic plan

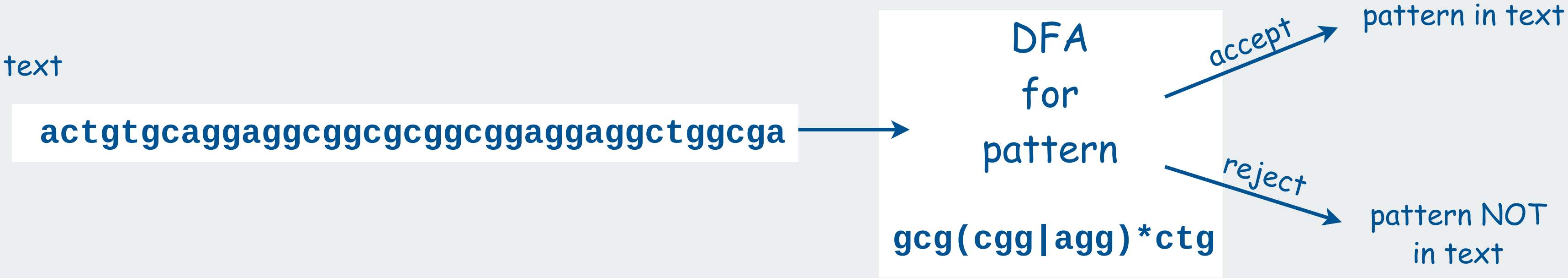
Overview is the same as for KMP !

- linear-time guarantee
- no backup in text stream

Ken Thompson

Basic plan for GREP

- build DFA from RE
- simulate DFA with text as input

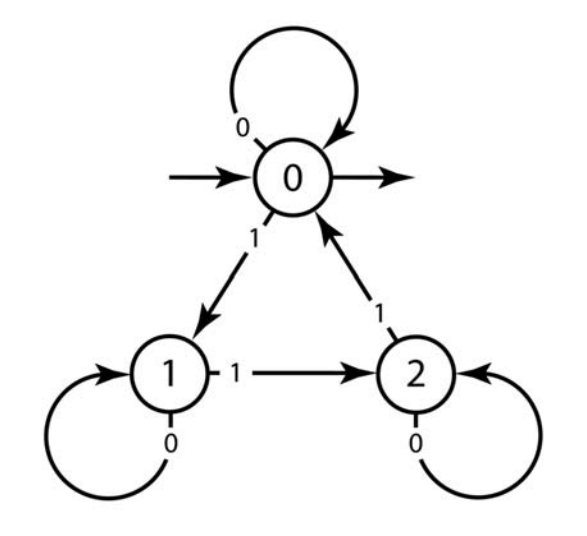


No backup in a DFA

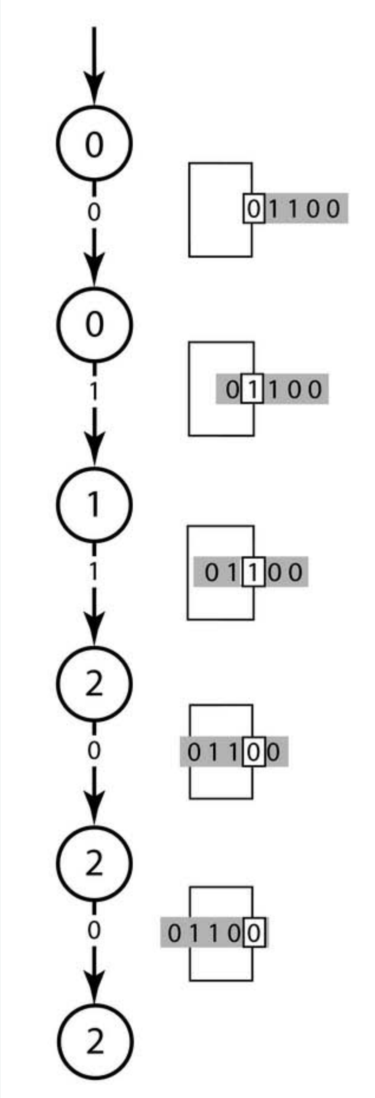
Linear-time because each step is just a state change

Deterministic finite-state automata

DFA review.



```
int pc = 0;
while (!tape.isEmpty())
{
   boolean bit = tape.read();
   if      (pc == 0) { if (!bit) pc = 0; else pc = 1; }
   else if (pc == 1) { if (!bit) pc = 1; else pc = 2; }
   else if (pc == 2) { if (!bit) pc = 2; else pc = 0; }
}
if (pc == 0) System.out.println("accepted");
else         System.out.println("rejected");
```



Duality

RE. Concise way to describe a set of strings.

DFA. Machine to recognize whether a given string is in a given set.

Kleene's theorem.

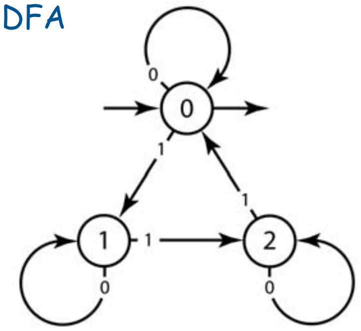
- for any DFA, there exists a RE that describes the same set of strings
- for any RE, there exists a DFA that recognizes the same set of strings

Ex: set of strings whose number of 1's is a multiple of 3

RE

```
0* | (0*10*10*10*)*
```

DFA



Good news: The basic plan works (build DFA from RE and run with text as input)

Bad news : The DFA can be exponentially large (can't afford to build it).

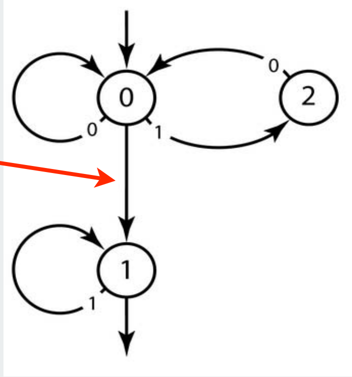
Consequence: We need a smaller abstract machine.

Nondeterministic finite-state automata

NFA.

- may have 0, 1, or more transitions for each input symbol
- may have ε-transitions (move to another state without reading input)
- accept if any sequence of transitions leads to accept state

Ex: set of strings that do not contain **`110`**



convention:
unlabelled arrows
are ε - transitions

in set: **`111`**, **`00011`**, **`101001011`**

not in set: **`110`**, **`00011011`**, **`00110`**

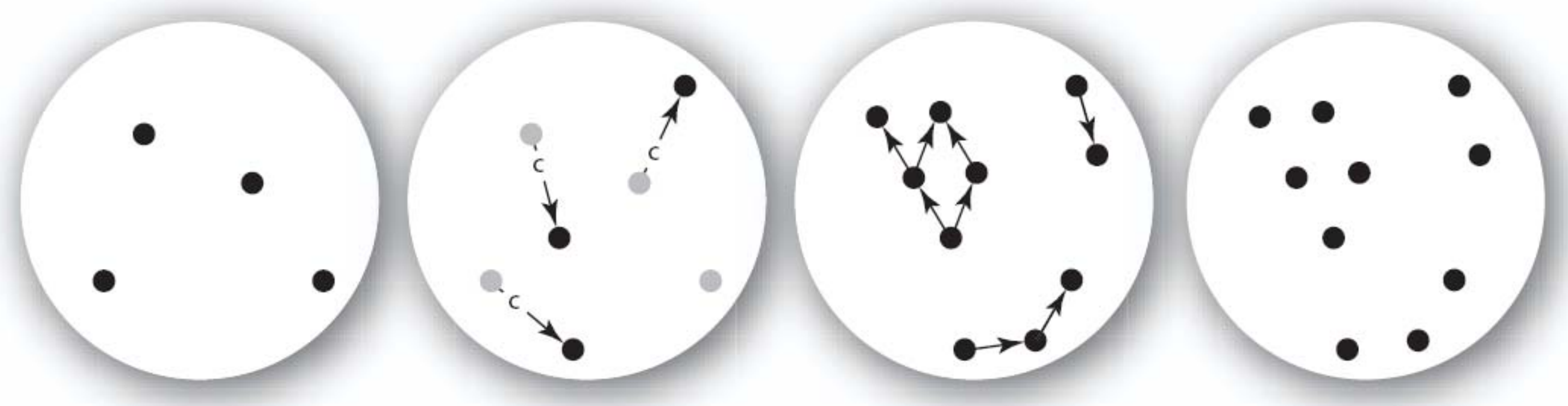
Implication of proof of Kleene's theorem: RE -> NFA -> DFA

Basic plan for GREP (revised)

- build NFA from RE
- simulate NFA with text as input
- give up on linear-time guarantee

Simulating an NFA

How to simulate an NFA? Maintain set of all possible states that NFA could be in after reading in the first i symbols.



all states reachable after reading i symbols

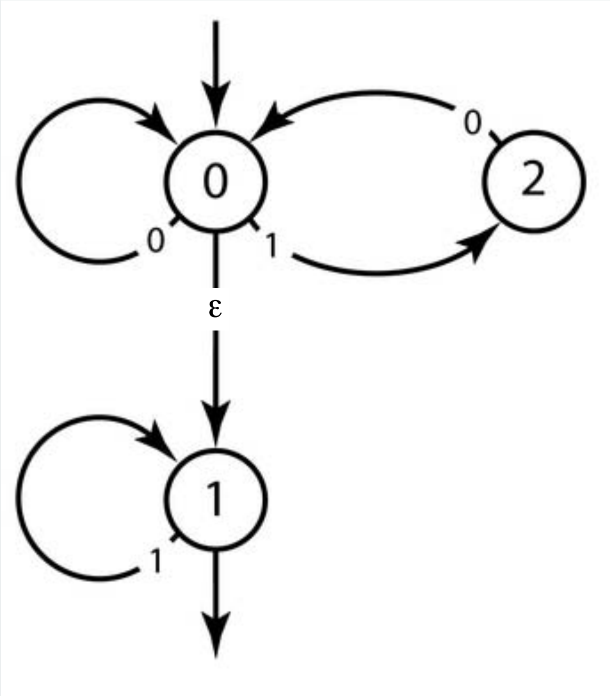
possible transitions on reading (i+1)st symbol c

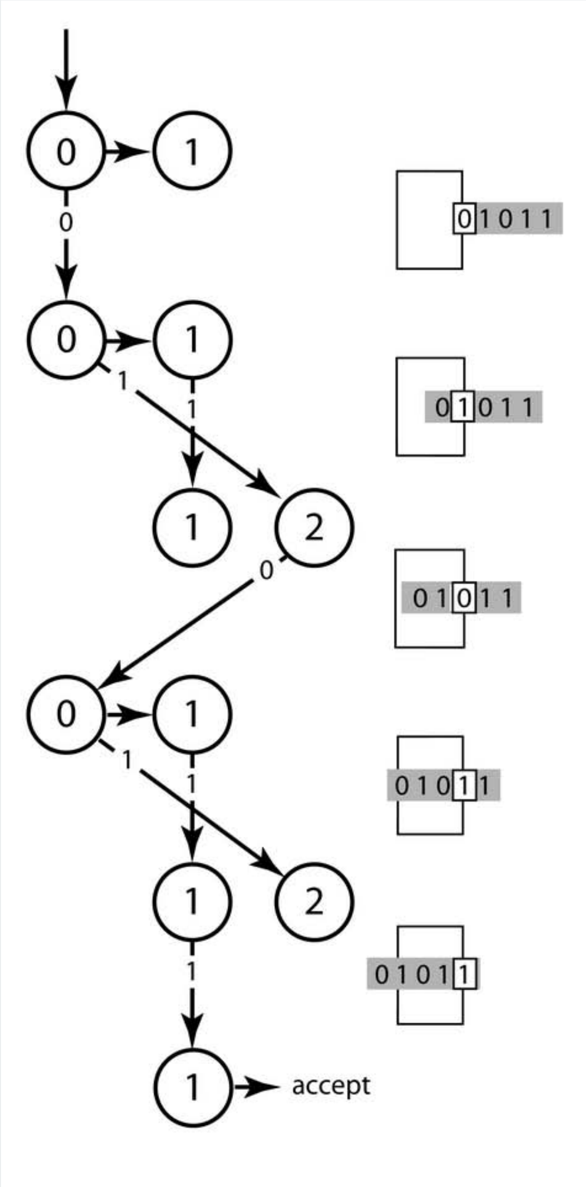
possible null transitions before reading next symbol

all states reachable after reading i+1 symbols

One step in simulating an NFA

NFA Simulation

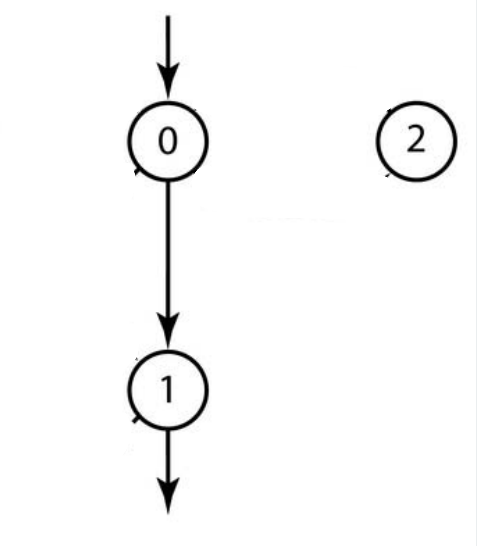




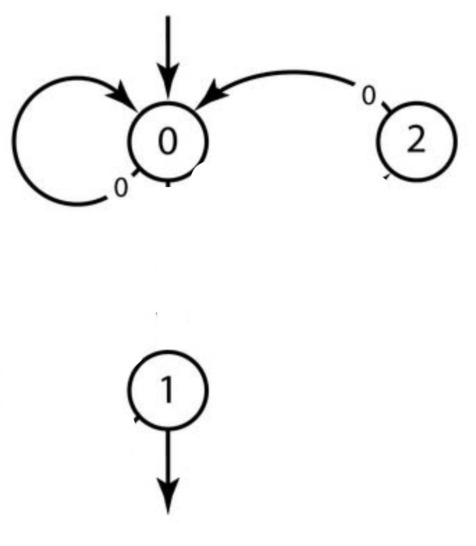
An NFA trace

NFA Representation

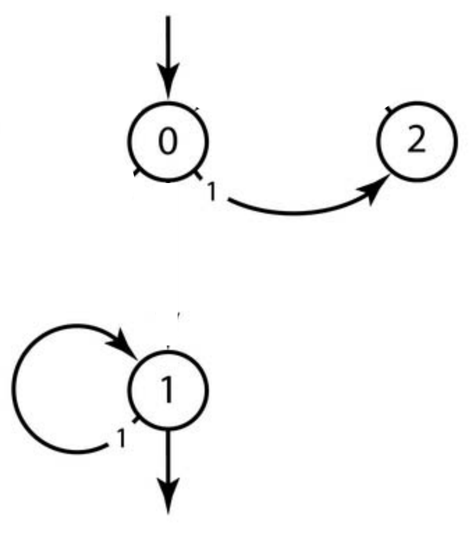
NFA representation. Maintain several digraphs, one for each symbol in the alphabet, plus one for ε.



ε-graph



0-graph



1-graph

NFA: Java Implementation

```
public class NFA
{
   private int START  = 0;              // start state
   private int ACCEPT = 1;              // accept state
   private int N      = 2;              // number of states
   private String ALPHABET = "01";      // RE alphabet
   private int EPS = ALPHABET.length(); // symbols in alphabet
   private Digraph[] G;

   public NFA(String re)
   {
      G = new Digraph[EPS + 1];
      for (int i = 0; i <= EPS; i++)
         G[i] = new Digraph();
      build(0, 1, re);
   }

   private void build(int from, int to, String re) { }
   public boolean simulate(Tape tape)               { }
}
```

NFA Simulation

How to simulate an NFA?

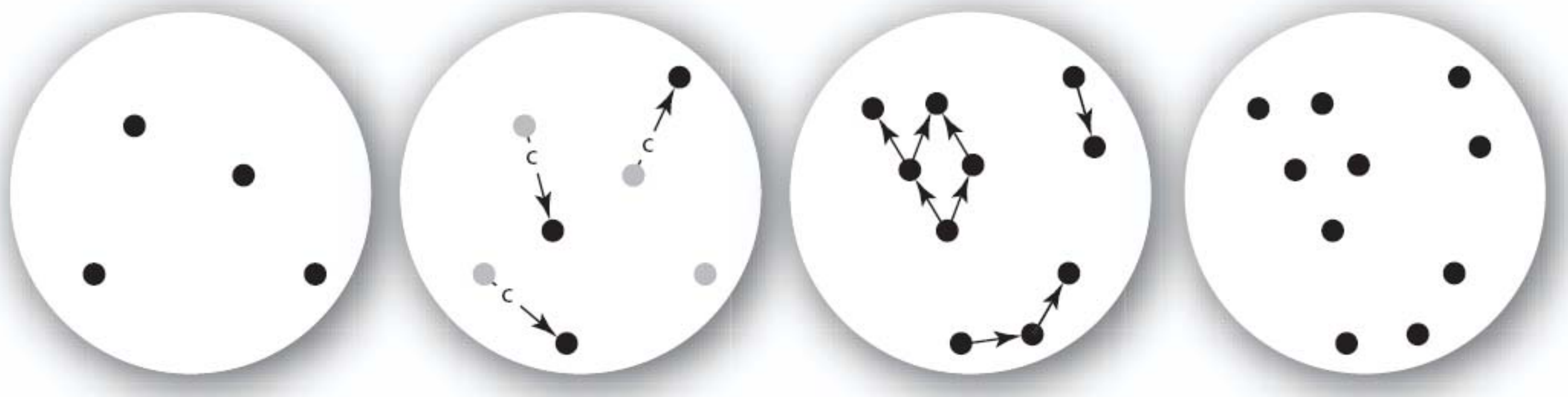
- Maintain a **SET** of all possible states that NFA could be in after reading in the first i symbols.
- Use **Digraph** adjacency and reachability ops to update.

pc

next = neighbors of **pc** in **G[c]**

states reachable from **next** in **G[ε]**

updated **pc**



all states reachable after reading i symbols

possible transitions on reading (i+1)st symbol c

possible null transitions before reading next symbol

all states reachable after reading i+1 symbols

NFA Simulation: Java Implementation

```
public boolean simulate(Tape tape)
{
   SET<Integer> pc = G[EPS].reachable(START);

   while (!tape.isEmpty())
   {  // Simulate NFA taking input from tape.
      char c = tape.read();
      int  i = ALPHABET.indexOf(c);
      SET<Integer> next = G[i].neighbors(pc);

      pc = G[EPS].reachable(next);
   }

   for (int state : pc)
      if (state == ACCEPT) return true;
   return false;
}
```

states reachable from start by ε-transitions

all possible states after reading character c from tape

follow ε-transitions

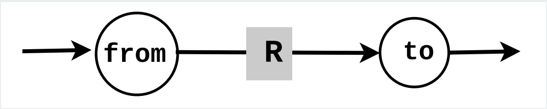
check whether in accept state at end

Converting from an RE to an NFA: basic transformations

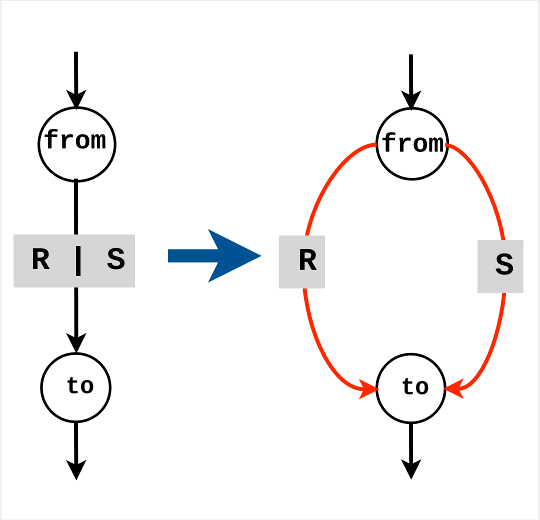
Use generalized NFA with full RE on trasitions arrows

- start with one transition having given RE
- remove operators with transformations given below
- goal: standard NFA (all single-character or epsilon-transitions)

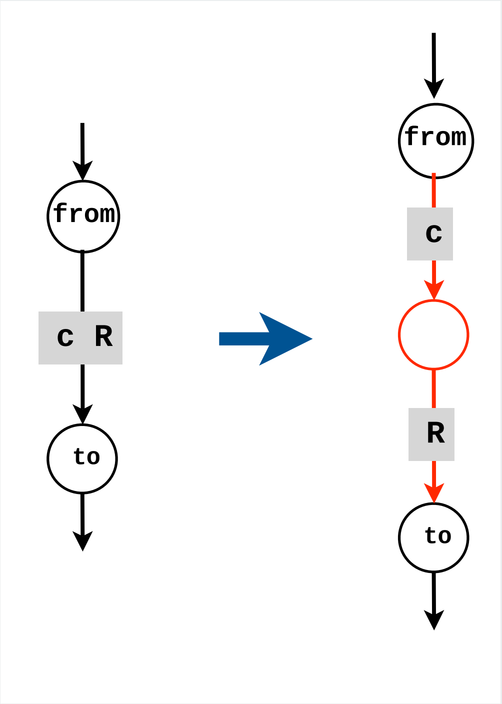
start



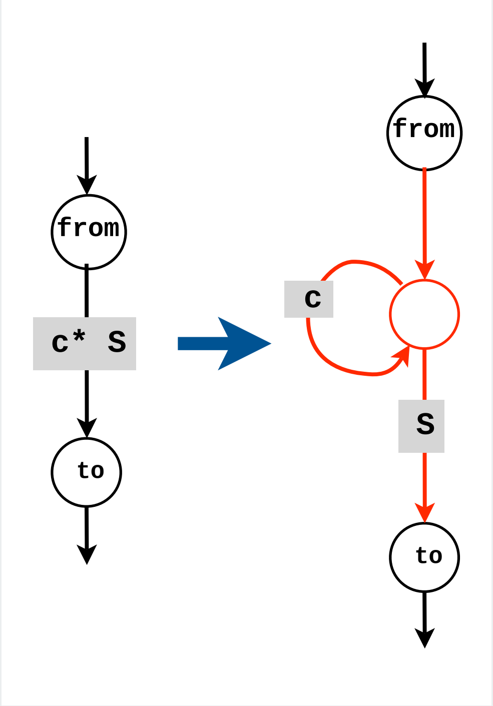
union



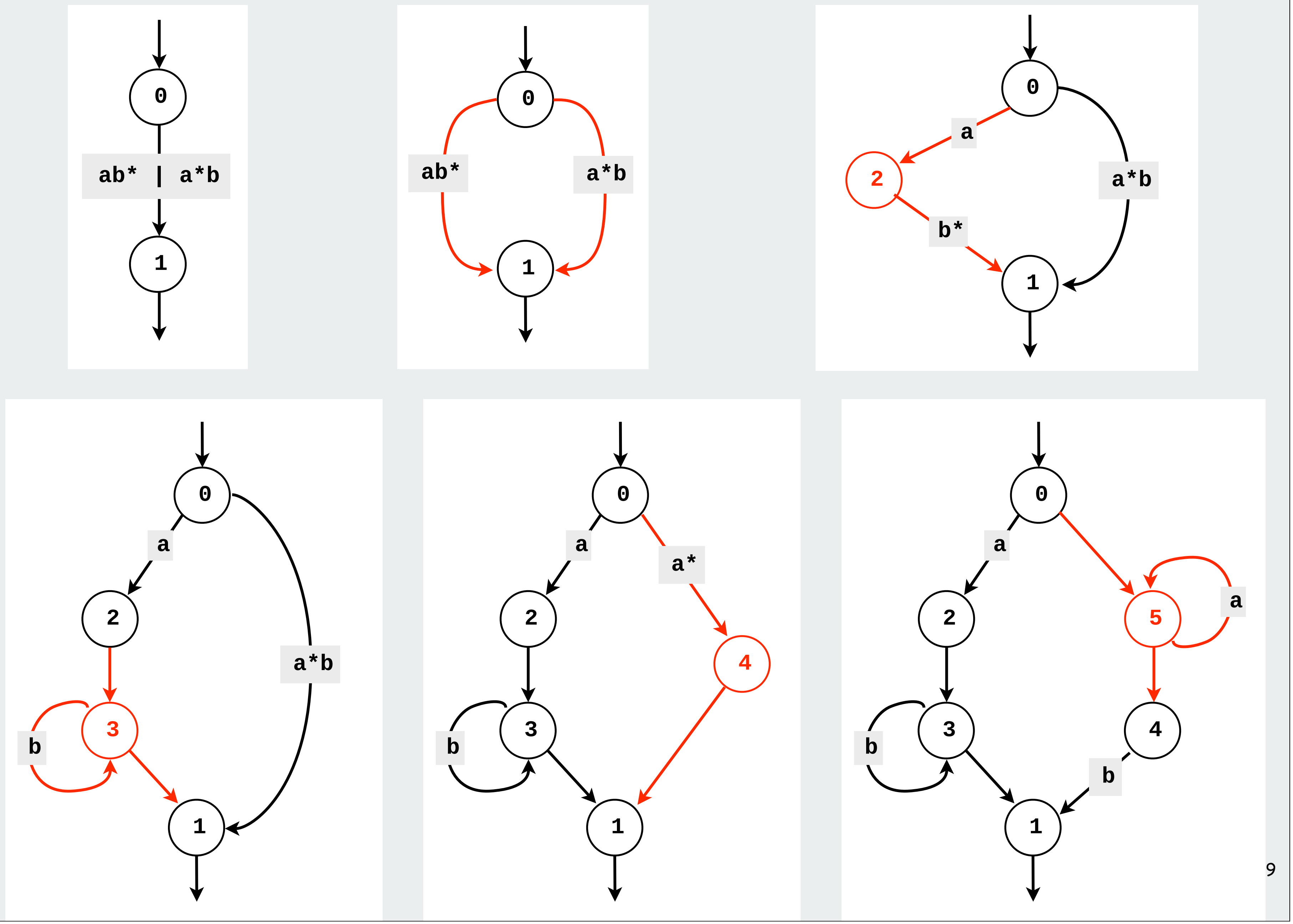
concatenation



closure



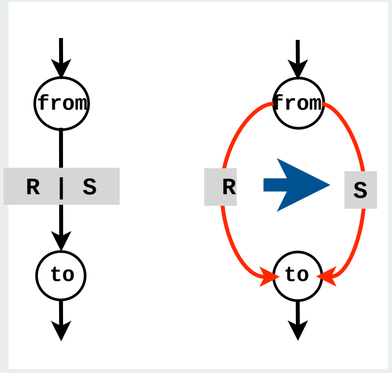
Converting from an RE to an NFA example: **ab* | ab***



NFA Construction: Java Implementation

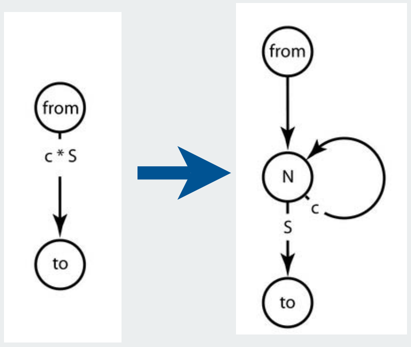
```
private void build(int from, int to, String re)
{
   int or = re.indexOf('|');

   if (re.length() == 0) G[EPSILON].addEdge(from, to);

   else if (re.length() == 1)
   {                                                   single char
     char c = re.charAt(0);
     for (int i = 0; i < EPSILON; i++)
        if (c == ALPHABET.charAt(i) || c == '.')
           G[i].addEdge(from, to);
   }

   else if (or != -1)
   {                                                        union
     build(from, to, re.substring(0, or));
     build(from, to, re.substring(or + 1));
   }

   else if (re.charAt(1) == '*')
   {                                                      closure
     G[EPSILON].addEdge(from, N);
     build(N, N, re.substring(0, 1));
     build(N++, to, re.substring(2));
   }

   else
   {                                                concatenation
     build(from, N, re.substring(0, 1));
     build(N++, to, re.substring(1));
   }
}
```

union



R | S

re.charAt(or)



c*   S

re.charAt(1)

Grep running time

Input. Text with N characters, RE with M characters.

Claim. The number of edges in the NFA is at most 2M.

- Single character: consumes 1 symbol, creates 1 edge.
- Wildcard character: consumes 1 symbol, creates 2 edges.
- Concatenation: consumes 1 symbols, creates 0 edges.
- Union: consumes 1 symbol, creates 1 edges.
- Closure: consumes one symbol, creates 2 edges.

NFA simulation. O(MN) since NFA has 2M transitions

- bottleneck: 1 graph reachability per input character
- can be substantially faster in practice if few ε-transitions

NFA construction. Ours is $O(M^2)$ but not hard to make O(M).

Surprising bottom line:

Worst-case cost for grep is the same as for elementary exact match!

Industrial-strength grep implementation

To complete the implementation,

- Deal with parentheses.
- Extend the alphabet.
- Add character classes.
- Add capturing capabilities.
- Deal with meta characters.
- Extend the closure operator.
- Error checking and recovery.
- Greedy vs. reluctant matching.

Regular expressions in Java (revisited)

RE pattern matching is implemented in Java's **`Pattern`** and **`Matcher`** classes

Ex: Harvesting. Print substrings of **`input`** that match **`re`**

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Harvester
{
   public static void main(String[] args)
   {
      String re       = args[0];
      In in           = new In(args[1]);
      String input    = in.readAll();
      Pattern pattern = Pattern.compile(re);
      Matcher matcher = pattern.matcher(input);
      while (matcher.find())
         System.out.println(matcher.group());
   }
}
```

`compile()` creates a **`Pattern`** (NFA) from RE

`matcher()` creates a **`Matcher`** (NFA simulator) from NFA and text

`find()` looks for the next match

`group()` returns the substring most recently found by **`find()`**

```
% java Harvester "gcg(cgg|agg)*ctg" chromosomeX.txt
gcgcggcggcggcggcggctg
gcgctg
gcgctg
gcgcggcggcggaggcggaggcggctg

% java Harvester "http://(\\w+\\.)*(\\w+)" http://www.cs.princeton.edu
http://www.princeton.edu
http://www.google.com
```

harvest patterns from DNA

harvest links from website

Typical application: Parsing a data file

Example. NCBI genome file, …

```
LOCUS AC146846 128142 bp DNA linear HTG 13-NOV-2003
DEFINITION Ornithorhynchus anatinus clone CLM1-393H9,
ACCESSION AC146846
KEYWORDS HTG; HTGS_PHASE2; HTGS_DRAFT.
SOURCE Ornithorhynchus anatinus (platypus)
ORIGIN
     1 tgtatttcat ttgaccgtgc tgttttttcc cggtttttca gtacggtgtt agggagccac
    61 gtgattctgt ttgttttatg ctgccgaata gctgctcgat gaatctctgc atagacagct  // a comment
   121 gccgcaggga gaaatgacca gtttgtgatg acaaaatgta ggaaagctgt ttcttcataa
   ...
128101 ggaaatgcga cccccacgct aatgtacagc ttctttagat tg
```

```
String regexp   = "[ ]*[0-9]+([actg ]*).*";
Pattern pattern = Pattern.compile(regexp);
In in = new In(filename);
while (!in.isEmpty())
{
   String line = in.readLine();
   Matcher matcher = pattern.matcher(line);
   if (matcher.find())
   {
      String s = matcher.group(1).replaceAll(" ", "");
      // Do something with s.
   }
}
```

replace this RE

with this string

the part of the match delimited
by the first group of parentheses

Algorithmic complexity attacks

Warning. Typical implementations do not guarantee performance!

grep, Java, Perl

```
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac             1.6 seconds
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac           3.7 seconds
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac         9.7 seconds
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac      23.2 seconds
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac    62.2 seconds
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 161.6 seconds
```

SpamAssassin regular expression.

```
java RE "[a-z]+@[a-z]+([a-z\.]+\.)+[a-z]+" spammer@x......................
```

- Takes exponential time.
- Spammer can use a pathological email address to DOS a mail server.

Not-so-regular expressions

Back-references.

- `\1` notation matches sub-expression that was matched earlier.
- Supported by typical RE implementations.

```
java Harvester "\b(.+)\1\b"  dictionary.txt
beriberi
couscous
```

word boundary

Some non-regular languages.

- set of strings of the form ww for some string w: **beriberi**.
- set of bitstrings with an equal number of 0s and 1s: **01110100**.
- set of Watson-Crick complemented palindromes: **atttcggaaat**.

Remark. Pattern matching with back-references is intractable.

Context

Abstract machines, languages, and nondeterminism.

- basis of the theory of computation
- intensively studied since the 1930s
- basis of programming languages

Compiler. A program that translates a program to machine code.

- KMP string ⇒ DFA.
- **`grep`** RE ⇒ NFA.
- **`javac`** Java language ⇒ Java byte code.

	KMP	grep	Java
pattern	string	RE	program
parser	unnecessary	check if legal	check if legal
compiler output	DFA	NFA	byte code
simulator	DFA simulator	NFA simulator	JVM

Summary of pattern-matching algorithms

Programmer:

- Implement exact pattern matching by DFA simulation (KMP).
- REs are a powerful pattern matching tool.
- Implement RE pattern matching by NFA simulation (grep).

Theoretician:

- RE is a compact description of a set of strings.
- NFA is an abstract machine equivalent in power to RE.
- DFAs and REs have limitations.

You: Practical application of core CS principles.

Example of essential paradigm in computer science.

- Build intermediate abstractions.
- Pick the right ones!
- Solve important practical problems.

Linear Programming

- ‣ brewer's problem
- ‣ simplex algorithm
- ‣ implementation
- ‣ linear programming

References:

The Allocation of Resources by Linear Programming,
Scientific American, by Bob Bland
Algs in Java, Part 5

Overview: introduction to advanced topics

Main topics

- linear programming: the ultimate practical problem-solving model
- reduction: design algorithms, prove limits, classify problems
- NP: the ultimate theoretical problem-solving model
- combinatorial search: coping with intractability

Shifting gears

- from linear/quadratic to polynomial/exponential scale
- from individual problems to problem-solving models
- from details of implementation to conceptual framework

Goals

- place algorithms we've studied in a larger context
- introduce you to important and essential ideas
- inspire you to learn more about algorithms!

Linear Programming

What is it?

see ORF 307

- Quintessential tool for optimal allocation of scarce resources, among a number of competing activities.
- Powerful and general problem-solving method that encompasses:

  shortest path, network flow, MST, matching, assignment...

  Ax = b, 2-person zero sum games

Why significant?

- Widely applicable problem-solving model
- Dominates world of industry.
- Fast commercial solvers available: CPLEX, OSL.
- Powerful modeling languages available: AMPL, GAMS.
- Ranked among most important scientific advances of 20th century.

Ex: Delta claims that LP saves $100 million per year.

Applications

Agriculture. Diet problem.

Computer science. Compiler register allocation, data mining.

Electrical engineering. VLSI design, optimal clocking.

Energy. Blending petroleum products.

Economics. Equilibrium theory, two-person zero-sum games.

Environment. Water quality management.

Finance. Portfolio optimization.

Logistics. Supply-chain management.

Management. Hotel yield management.

Marketing. Direct mail advertising.

Manufacturing. Production line balancing, cutting stock.

Medicine. Radioactive seed placement in cancer treatment.

Operations research. Airline crew assignment, vehicle routing.

Physics. Ground states of 3-D Ising spin glasses.

Plasma physics. Optimal stellarator design.

Telecommunication. Network design, Internet routing.

Sports. Scheduling ACC basketball, handicapping horse races.

- **brewer’s problem**
- simplex algorithm
- implementation
- linear programming

Toy LP example: Brewer's problem

Small brewery produces ale and beer.

- Production limited by scarce resources: corn, hops, barley malt.
- Recipes for ale and beer require different proportions of resources.

	corn (lbs)	hops (oz)	malt (lbs)	profit ($)
available	480	160	1190	
ale (1 barrel)	5	4	35	13
beer (1 barrel)	15	4	20	23

Brewer's problem: choose product mix to maximize profits.

all ale (34 barrels)	179	136	1190	442
all beer (32 barrels)	480	128	640	736
20 barrels ale 20 barrels beer	400	160	1100	720
12 barrels ale 28 barrels beer	480	160	980	800
more profitable product mix?	?	?	?	>800 ?

34 barrels times 35 lbs malt per barrel is 1190 lbs
[amount of available malt]

Brewer's problem: mathematical formulation

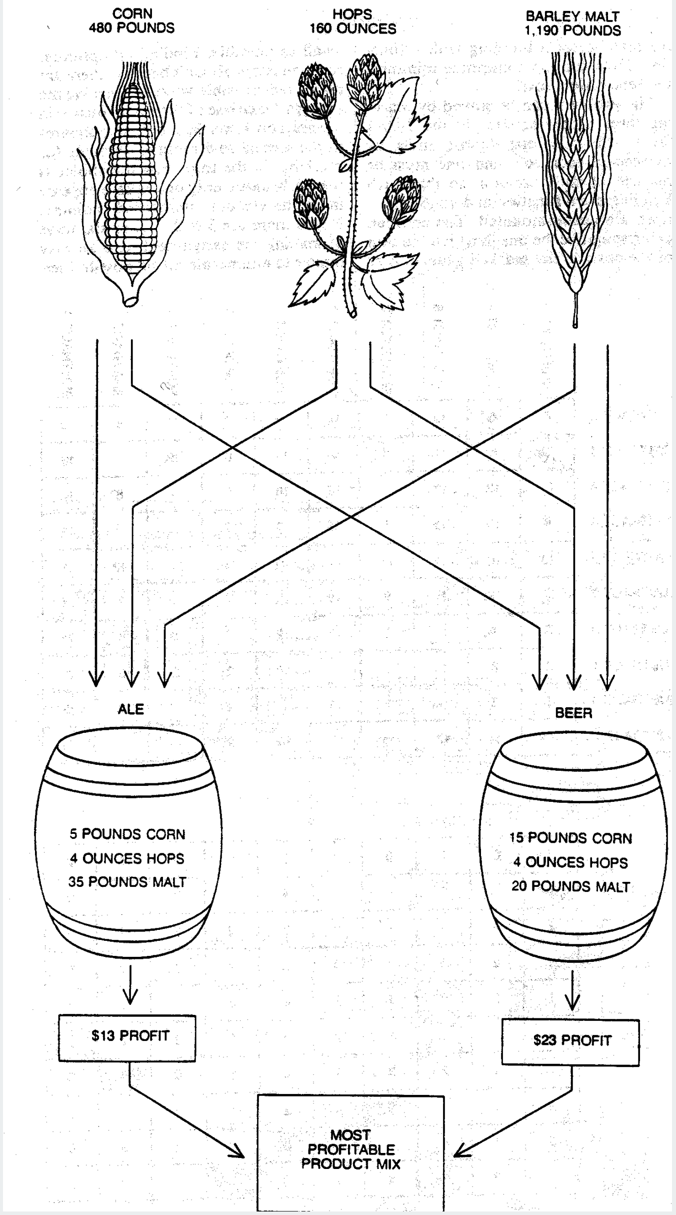
Small brewery produces ale and beer.

- Production limited by scarce resources: corn, hops, barley malt.
- Recipes for ale and beer require different proportions of resources.

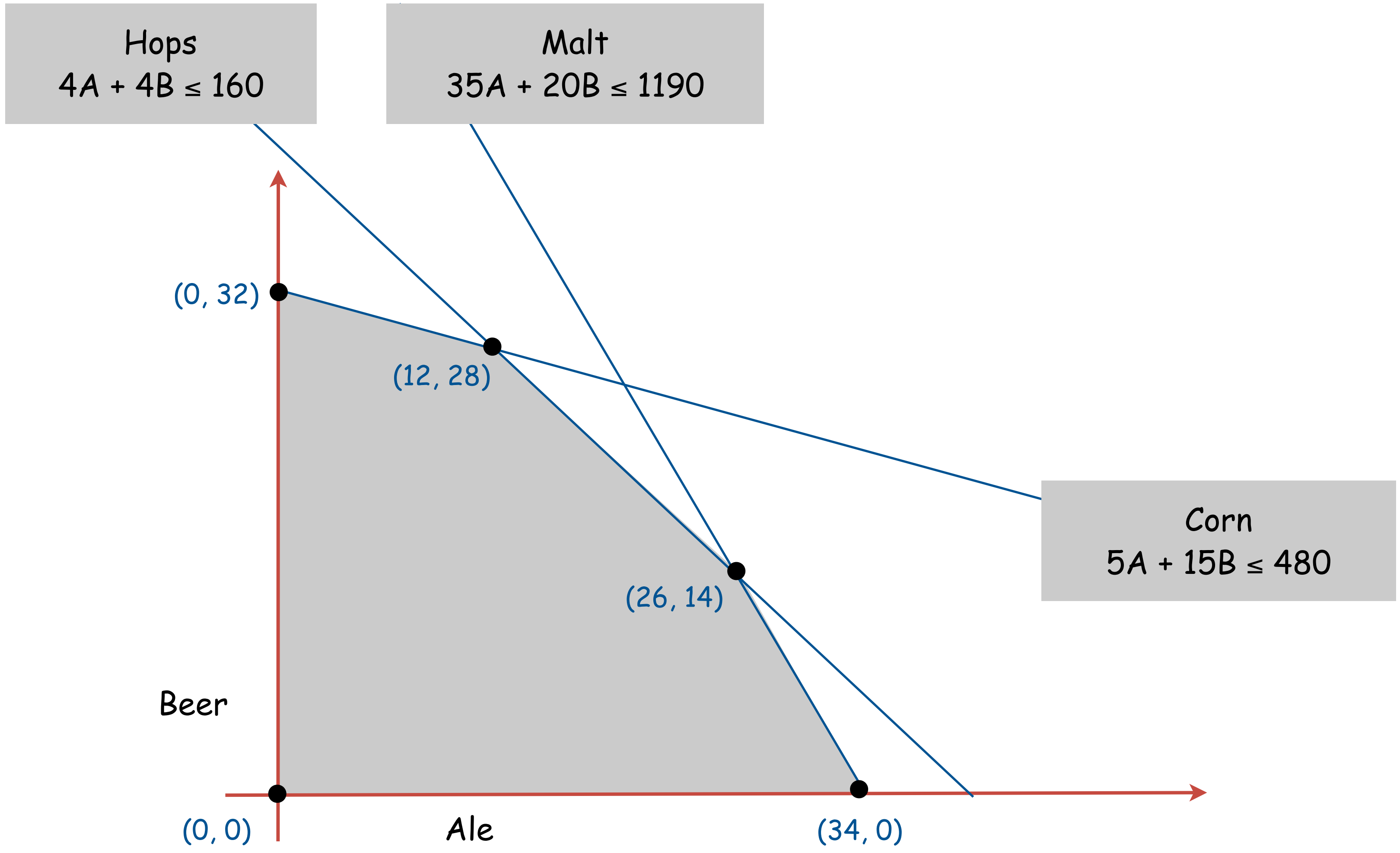
Mathematical formulation

- let A be the number of barrels of beer
- and B be the number of barrels of ale

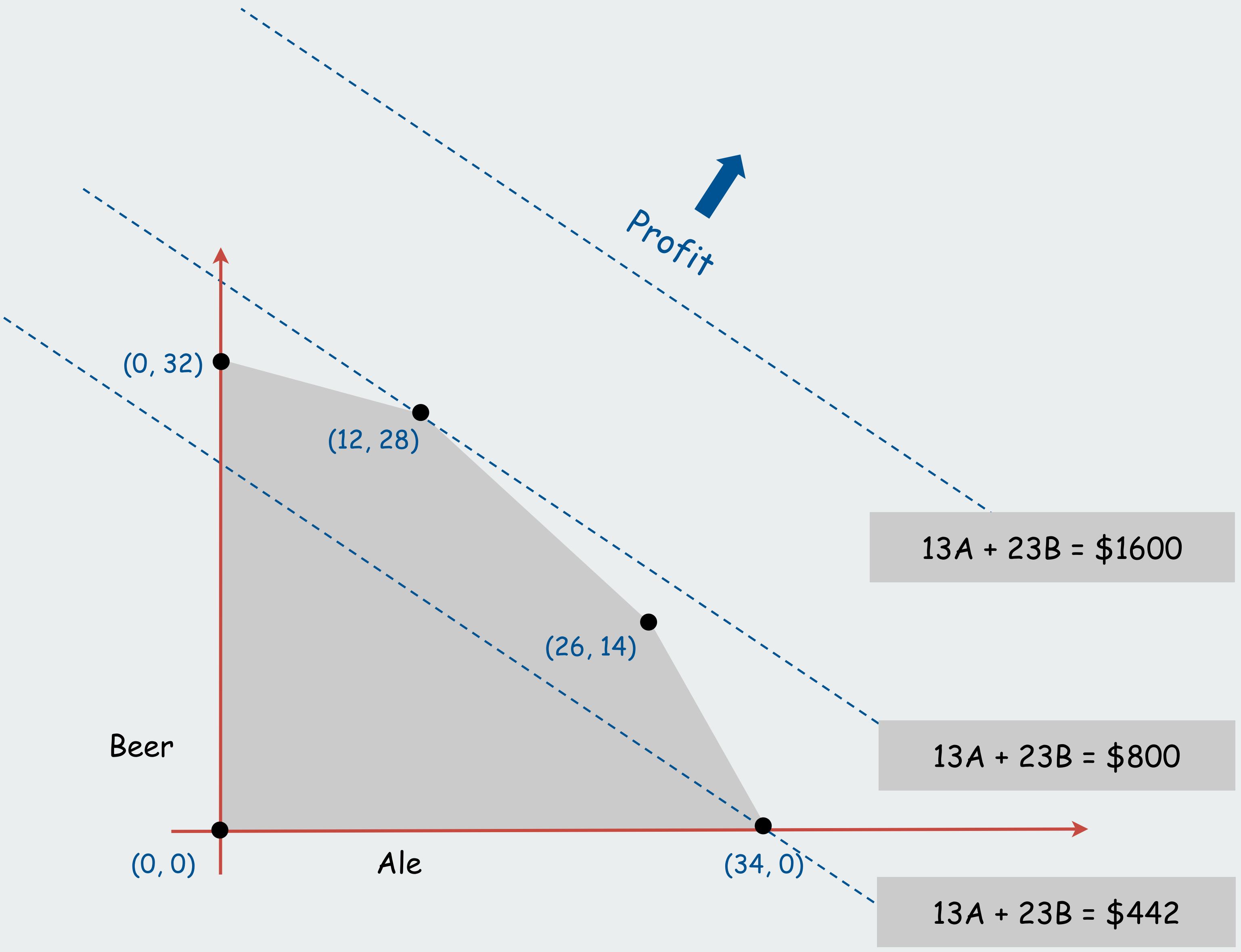
	ale		beer			
maximize	13A	+	23B			profit
subject to the constraints	5A	+	15B	≤	480	corn
	4A	+	4B	≤	160	hops
	35A	+	20B	≤	1190	malt
			A	≥	0	
			B	≥	0	



Brewer's problem: Feasible region

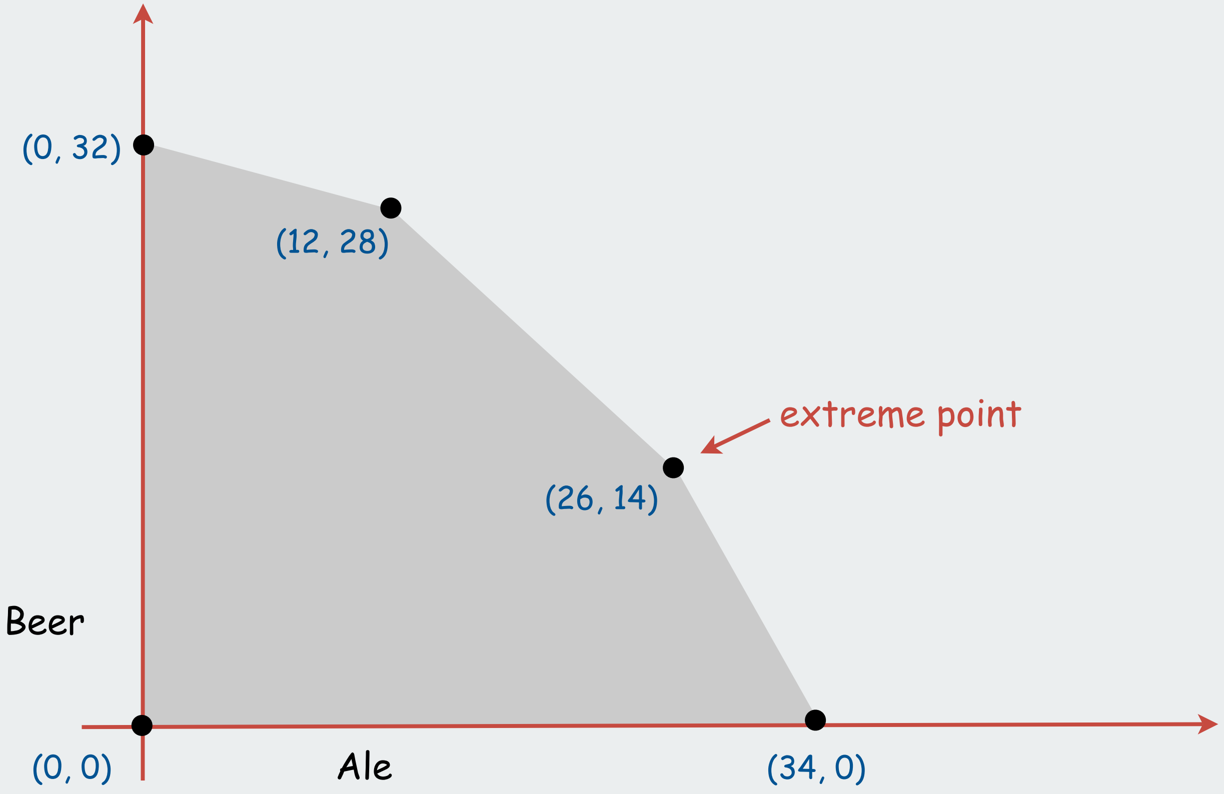


Brewer's problem: Objective function



Brewer's problem: Geometry

Brewer's problem observation. Regardless of objective function coefficients, an optimal solution occurs at an extreme point.



Standard form linear program

Input: real numbers a_{ij}, c_j, b_i.

Output: real numbers x_j.

n = # nonnegative variables, m = # constraints.

Maximize linear objective function subject to linear equations.

n variables

maximize $c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$

subject to the constraints

m equations

$$\begin{aligned} a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n &= b_1 \\ a_{21} x_1 + a_{22} x_2 + \ldots + a_{2n} x_n &= b_2 \\ &\ldots \\ a_{m1} x_1 + a_{m2} x_2 + \ldots + a_{mn} x_n &= b_m \\ x_1 , x_2 ,\ldots , x_n &\geq 0 \end{aligned}$$

matrix version

maximize $c^T x$

subject to the constraints

$$A x = b$$

$$x \geq 0$$

"Linear" No x^2, xy, $\arccos(x)$, etc.

"Programming" " Planning" (term predates computer programming).

Converting the brewer's problem to the standard form

Original formulation

$$
\begin{array}{lrcrcr}
\text{maximize} & 13A & + & 23B & & \\
\text{subject} & 5A & + & 15B & \leq & 480 \\
\text{to the} & 4A & + & 4B & \leq & 160 \\
\text{constraints} & 35A & + & 20B & \leq & 1190 \\
 & & & A,\ B & \geq & 0
\end{array}
$$

Standard form

- add variable Z and equation corresponding to objective function
- add slack variable to convert each inequality to an equality.
- now a 5-dimensional problem.

$$
\begin{array}{lrcrcccccccr}
\text{maximize} & Z & & & & & & & & & & \\
\text{subject} & 13A & + & 23B & & & & & & - Z & = & 0 \\
\text{to the} & 5A & + & 15B & + & S_C & & & & & = & 480 \\
\text{constraints} & 4A & + & 4B & & & + & S_H & & & = & 160 \\
 & 35A & + & 20B & & & & & + \ S_M & & = & 1190 \\
 & & & \multicolumn{6}{c}{A,\ B,\ S_C,\ S_H,\ S_M} & & \geq & 0
\end{array}
$$

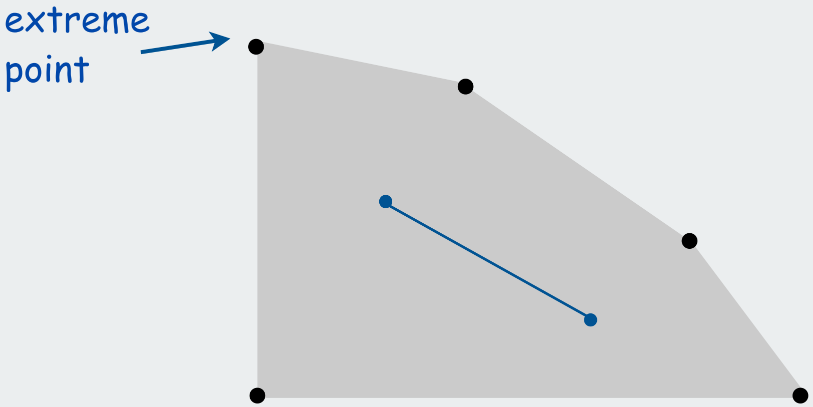
Geometry

A few principles from geometry:

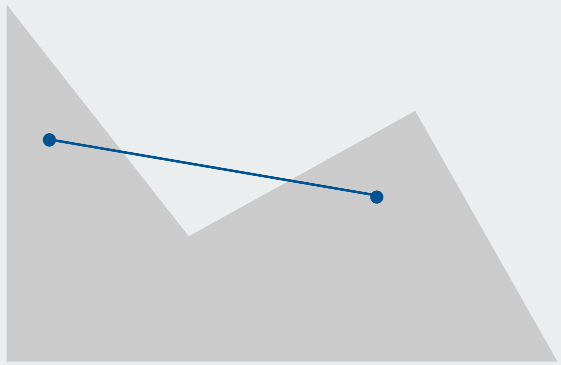
- inequality: halfplane (2D), hyperplane (kD).
- bounded feasible region: convex polygon (2D), convex polytope (kD).

Convex set. If two points a and b are in the set, then so is $\frac{1}{2}(a + b)$.

Extreme point. A point in the set that can't be written as $\frac{1}{2}(a + b)$, where a and b are two distinct points in the set.



convex



not convex

Geometry (continued)

Extreme point property. If there exists an optimal solution to (P), then there exists one that is an extreme point.

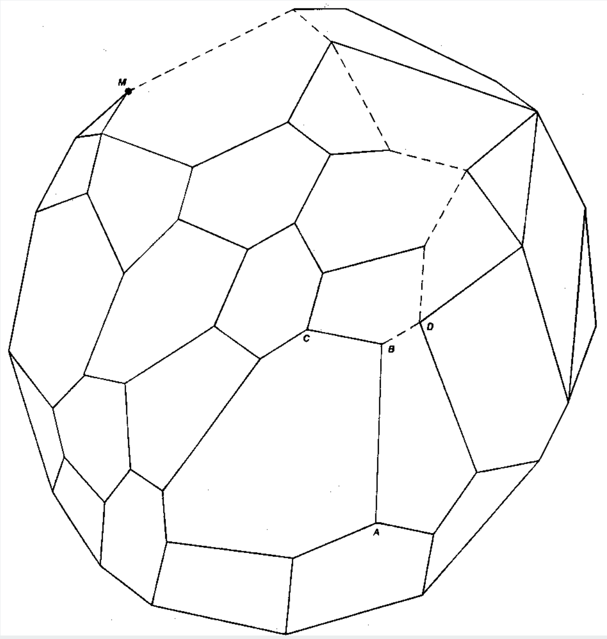
Good news. Only need to consider finitely many possible solutions.

Bad news. Number of extreme points can be exponential !

Ex: n-dimensional hypercube

Greedy property. Extreme point is optimal iff no neighboring extreme point is better.

local optima are global optima



- brewer’s problem
- **simplex algorithm**
- implementation
- linear programming

Simplex Algorithm

Simplex algorithm. [George Dantzig, 1947]

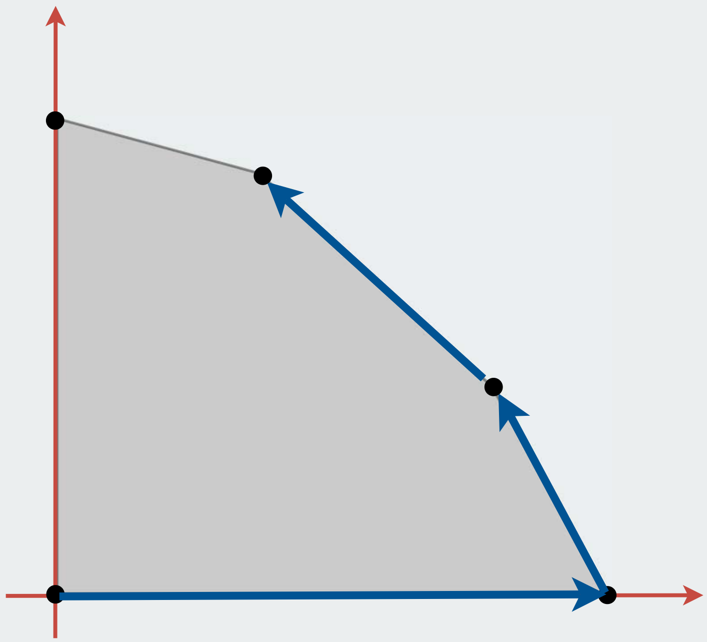
- Developed shortly after WWII in response to logistical problems, including Berlin airlift.
- One of greatest and most successful algorithms of all time.

Generic algorithm.

- Start at some extreme point.
- Pivot from one extreme point to a neighboring one.
- Repeat until optimal.

never decreasing objective function

How to implement? Linear algebra.



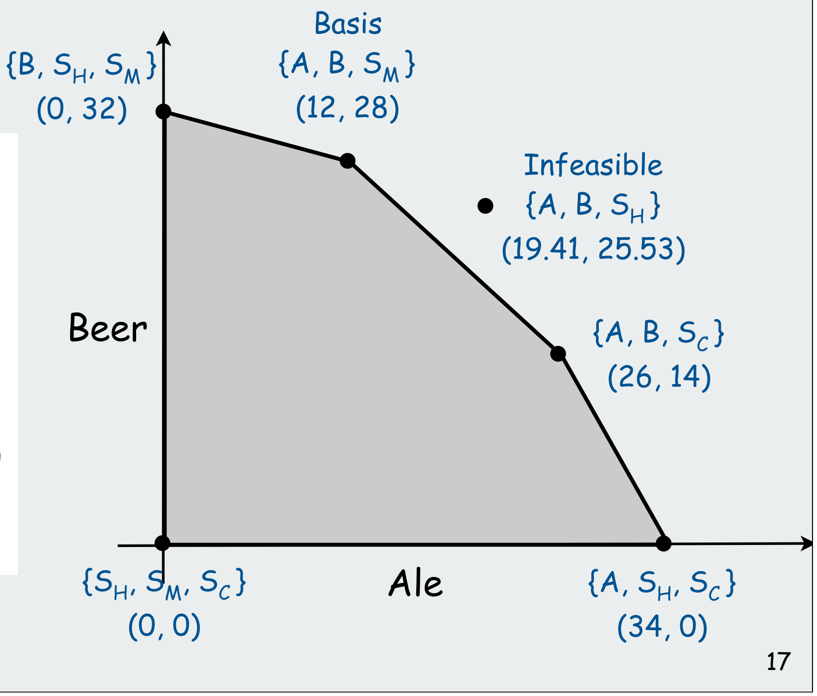
Simplex Algorithm: Basis

Basis. Subset of m of the n variables.

Basic feasible solution (BFS).

- Set n - m nonbasic variables to 0, solve for remaining m variables.
- Solve m equations in m unknowns.
- If unique and feasible solution ⇒ BFS.
- BFS ⇔ extreme point.

$$
\begin{array}{llllllllll}
\text{maximize} & Z \\
\text{subject to the constraints} & 13A & + & 23B & & & & - Z & = & 0 \\
& 5A & + & 15B & + S_C & & & & = & 480 \\
& 4A & + & 4B & & + S_H & & & = & 160 \\
& 35A & + & 20B & & & + S_M & & = & 1190 \\
& & & A, \; B, & S_C, & S_H, & S_M & & \geq & 0
\end{array}
$$



Simplex Algorithm: Initialization

Start with slack variables as the basis.

Initial basic feasible solution (BFS).

- set non-basis variables $A = 0$, $B = 0$ (and $Z = 0$).
- 3 equations in 3 unknowns give $S_C = 480$, $S_C = 160$, $S_C = 1190$ (immediate).
- extreme point on simplex: origin

$$
\begin{array}{lcccccccl}
\text{maximize} & Z & & & & & & & \\
\text{subject to the constraints} & 13A & + & 23B & & & & - Z & = 0 \\
 & 5A & + & 15B & + S_C & & & & = 480 \\
 & 4A & + & 4B & & + S_H & & & = 160 \\
 & 35A & + & 20B & & & + S_M & & = 1190 \\
 & & & A,\ B,\ S_C,\ S_H,\ S_M & & & & & \geq 0
\end{array}
$$

basis = $\{S_C, S_H, S_M\}$

$A = B = 0$

$Z = 0$

$S_C = 480$

$S_H = 160$

$S_M = 1190$

Simplex Algorithm: Pivot 1

$$
\begin{array}{llcrcrcrcrcrcrcr}
\text{maximize} & Z & & & & & & & & & & & & \\
\text{subject to the constraints} & 13A & + & 23B & & & & & & & - & Z & = & 0 \\
 & 5A & + & \boxed{15B} & + & S_C & & & & & & & = & 480 \\
 & 4A & + & 4B & & & + & S_H & & & & & = & 160 \\
 & 35A & + & 20B & & & & & + & S_M & & & = & 1190 \\
 & & & \multicolumn{7}{c}{A,\ B,\ S_C,\ S_H,\ S_M} & & & \geq & 0
\end{array}
$$

basis = $\{S_C, S_H, S_M\}$

$A = B = 0$

$Z = 0$

$S_C = 480$

$S_H = 160$

$S_M = 1190$

Substitution $B = (1/15)(480 - 5A - S_C)$ puts B into the basis ←

which variable does it replace?

(rewrite 2nd equation, eliminate B in 1st, 3rd, and 4th equations)

$$
\begin{array}{llcrcrcrcrcrcr}
\text{maximize} & Z & & & & & & & & & & & \\
\text{subject to the constraints} & (16/3)A & & & - & (23/15)\,S_C & & & & & - & Z & = & -736 \\
 & (1/3)\,A & + & B & + & (1/15)\,S_C & & & & & & & = & 32 \\
 & (8/3)\,A & & & - & (4/15)\,S_C & + & S_H & & & & & = & 32 \\
 & (85/3)\,A & & & - & (4/3)\,S_C & & & + & S_M & & & = & 550 \\
 & & & \multicolumn{7}{c}{A,\ B,\ S_C,\ S_H,\ S_M} & & & \geq & 0
\end{array}
$$

basis = $\{B, S_H, S_M\}$

$A = S_C = 0$

$Z = 736$

$B = 32$

$S_H = 32$

$S_M = 550$

Simplex Algorithm: Pivot 1

$$\begin{array}{lccccccccc}
\text{maximize} & Z & & & & & & & & \\
\text{subject to the constraints} & 13A & + & 23B & & & & -\ Z & = & 0 \\
 & 5A & + & \boxed{15B} & +\ S_C & & & & = & 480 \\
 & 4A & + & 4B & & +\ S_H & & & = & 160 \\
 & 35A & + & 20B & & & +\ S_M & & = & 1190 \\
 & & & A,\ B,\ S_C,\ S_H,\ S_M & & & & & \geq & 0
\end{array}$$

basis = $\{S_C, S_H, S_M\}$

$A = B = 0$

$Z = 0$

$S_C = 480$

$S_H = 160$

$S_M = 1190$

Why pivot on B?

- Its objective function coefficient is positive
  (each unit increase in B from 0 increases objective value by \$23)
- Pivoting on column 1 also OK.

Why pivot on row 2?

- Preserves feasibility by ensuring RHS $\geq$ 0.
- Minimum ratio rule: min { 480/15, 160/4, 1190/20 }.

Simplex Algorithm: Pivot 2

$$
\begin{array}{llcccccccl}
\text{maximize} & Z & & & & & & & & \\
\text{subject to the constraints} & \frac{16}{3}A & & & - & \frac{23}{15}S_C & & & -\ Z & = -736 \\
 & \frac{1}{3}A & + & B & + & \frac{1}{15}S_C & & & & = 32 \\
 & \frac{8}{3}A & & & - & \frac{4}{15}S_C & + S_H & & & = 32 \\
 & \frac{85}{3}A & & & - & \frac{4}{3}S_C & & + S_M & & = 550 \\
 & & & & & A,\ B,\ S_C,\ S_H,\ S_M & & & & \geq 0
\end{array}
$$

basis = $\{B, S_H, S_M\}$

$A = S_C = 0$

$Z = 736$

$B = 32$

$S_H = 32$

$S_M = 550$

Substitution $A = (3/8)(32 + (4/15)\,S_C - S_H)$ puts A into the basis

(rewrite 3nd equation, eliminate A in 1st, 2rd, and 4th equations)

$$
\begin{array}{llccccccl}
\text{maximize} & Z & & & & & & & \\
\text{subject to the constraints} & & & - & S_C & - & 2S_H & -\ Z & = -800 \\
 & & B & + & \frac{1}{10}S_C & + & \frac{1}{8}S_H & & = 28 \\
 & A & & - & \frac{1}{10}S_C & + & \frac{3}{8}S_H & & = 12 \\
 & & & - & \frac{25}{6}S_C & - & \frac{85}{8}S_H & + S_M & = 110 \\
 & & & & A,\ B,\ S_C,\ S_H,\ S_M & & & & \geq 0
\end{array}
$$

basis = $\{A, B, S_M\}$

$S_C = S_H = 0$

$Z = 800$

$B = 28$

$A = 12$

$S_M = 110$

Simplex algorithm: Optimality

Q. When to stop pivoting?

A. When all coefficients in top row are non-positive.

Q. Why is resulting solution optimal?

A. Any feasible solution satisfies system of equations in tableaux.

- In particular: $Z = 800 - S_C - 2\ S_H$
- Thus, optimal objective value $Z^* \le 800$ since $S_C, S_H \ge 0$.
- Current BFS has value 800 ⇒ optimal.

$$
\begin{array}{lllllllllll}
\text{maximize} & Z & & & & & & & & & \\
\text{subject to the constraints} & & & - & S_C & - & 2S_H & & - Z & = & -800 \\
 & & B & + & (1/10)\ S_C & + & (1/8)\ S_H & & & = & 28 \\
 & A & & - & (1/10)\ S_C & + & (3/8)\ S_H & & & = & 12 \\
 & & & - & (25/6)\ S_C & - & (85/8)\ S_H & + S_M & & = & 110 \\
 & & & & A,\ B,\ S_C,\ S_H,\ S_M & & & & & \ge & 0
\end{array}
$$

basis = $\{A, B, S_M\}$

$S_C = S_H = 0$

$Z = 800$

$B = 28$

$A = 12$

$S_M = 110$

- brewer’s problem
- simplex algorithm
- **implementation**
- linear programming

Simplex tableau

Encode standard form LP in a single Java 2D array

$$
\begin{array}{llllllll}
\text{maximize} & Z & & & & & & \\
\text{subject to the constraints} & 13A & + & 23B & & & - Z = & 0 \\
 & 5A & + & 15B & + S_C & & = & 480 \\
 & 4A & + & 4B & & + S_H & = & 160 \\
 & 35A & + & 20B & & + S_M & = & 1190 \\
 & & & A,\ B,\ S_C,\ S_H,\ S_M & & & \geq & 0
\end{array}
$$

5	15	1	0	0	480
4	4	0	1	0	160
35	20	0	0	1	1190
13	23	0	0	0	0

$$
\begin{array}{c|c|c|c}
m & A & I & b \\
\hline
1 & c & 0 & 0 \\
\hline
 & n & m & 1
\end{array}
$$

Simplex tableau

Encode standard form LP in a single Java 2D array (solution)

$$
\begin{array}{llcccccccccc}
\text{maximize} & Z & & & & & & & & & & \\
\text{subject to the constraints} & & & - & S_C & - & 2S_H & & - Z & = & -800 \\
 & & B & + & (1/10)\, S_C & + & (1/8)\, S_H & & & = & 28 \\
 & A & & - & (1/10)\, S_C & + & (3/8)\, S_H & & & = & 12 \\
 & & & - & (25/6)\, S_C & - & (85/8)\, S_H & + & S_M & = & 110 \\
 & & & & \multicolumn{5}{l}{A,\ B,\ S_C,\ S_H,\ S_M} & \geq & 0
\end{array}
$$

0	1	1/10	1/8	0	28
1	0	1/10	3/8	0	12
0	0	25/6	85/8	1	110
0	0	-1	-2	0	-800

$$
\begin{array}{c|c|c|c|}
m & A & I & b \\
\hline
1 & c & 0 & 0 \\
 & n & m & 1
\end{array}
$$

Simplex algorithm transforms initial array into solution

Simplex algorithm: Bare-bones implementation

Construct the simplex tableau.

$$\begin{array}{c|c|c|c|}
m & A & I & b \\
\hline
1 & c & 0 & 0 \\
\hline
 & n & m & 1
\end{array}$$

```
public class Simplex
{
   private double[][] a;   // simplex tableaux
   private int M, N;

   public Simplex(double[][] A, double[] b, double[] c)
   {
      M = b.length;
      N = c.length;
      a = new double[M+1][M+N+1];
      for (int i = 0; i < M; i++)
         for (int j = 0; j < N; j++)
            a[i][j] = A[i][j];
      for (int j = N; j < M + N; j++) a[j-N][j] = 1.0;
      for (int j = 0; j < N;     j++) a[M][j]   = c[j];
      for (int i = 0; i < M;     i++) a[i][M+N] = b[i];
   }
```

constructor

put A[][] into tableau

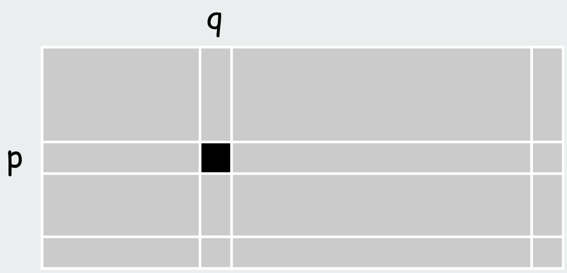
put I[] into tableau

put c[] into tableau

put b[] into tableau

Simplex algorithm: Bare-bones Implementation

Pivot on element (p, q).



```
public void pivot(int p, int q)
{
   for (int i = 0; i <= M; i++)
      for (int j = 0; j <= M + N; j++)
         if (i != p && j != q)
            a[i][j] -= a[p][j] * a[i][q] / a[p][q];

   for (int i = 0; i <= M; i++)
      if (i != p) a[i][q] = 0.0;

   for (int j = 0; j <= M + N; j++)
      if (j != q) a[p][j] /= a[p][q];
   a[p][q] = 1.0;
}
```

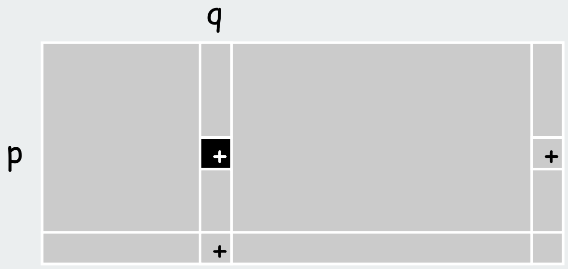
scale all elements but row p and column q

zero out column q

scale row p

Simplex Algorithm: Bare Bones Implementation

Simplex algorithm.



```
public void solve()
{
   while (true)
   {
      int p, q;
      for (q = 0; q < M + N; q++)
         if (a[M][q] > 0) break;
      if (q >= M + N) break;

      for (p = 0; p < M; p++)
         if (a[p][q] > 0) break;
      for (int i = p+1; i < M; i++)
         if (a[i][q] > 0)
            if (a[i][M+N] / a[i][q]
                < a[p][M+N] / a[p][q])
               p = i;

      pivot(p, q);
   }
}
```

find entering variable q
(positive objective function coefficient)

find row p according
to min ratio rule

min ratio test

Simplex Algorithm: Running Time

Remarkable property. In practice, simplex algorithm typically terminates after at most 2(m+n) pivots.

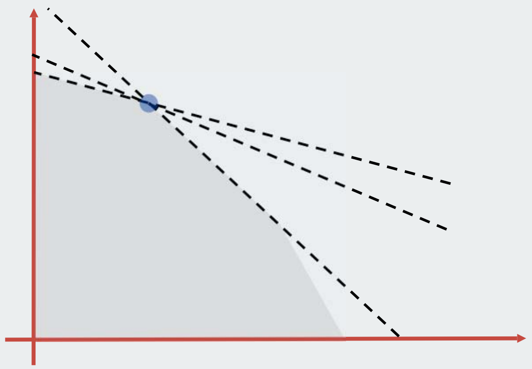
- No pivot rule that is guaranteed to be polynomial is known.
- Most pivot rules known to be exponential (or worse) in worst-case.

Pivoting rules. Carefully balance the cost of finding an entering variable with the number of pivots needed.

Simplex algorithm: Degeneracy

Degeneracy. New basis, same extreme point.

"stalling" is common in practice



Cycling. Get stuck by cycling through different bases that all correspond to same extreme point.

- Doesn't occur in the wild.
- Bland's least index rule guarantees finite # of pivots.

Simplex Algorithm: Implementation Issues

To improve the bare-bones implementation

- Avoid stalling.
- Choose the pivot wisely.
- Watch for numerical stability.
- Maintain sparsity. ← requires fancy data structures
- Detect infeasiblity
- Detect unboundedness.
- Preprocess to reduce problem size.

Basic implementations available in many programming environments.

Commercial solvers routinely solve LPs with millions of variables.

LP solvers: basic implementations

Ex. 1: OR-Objects Java library

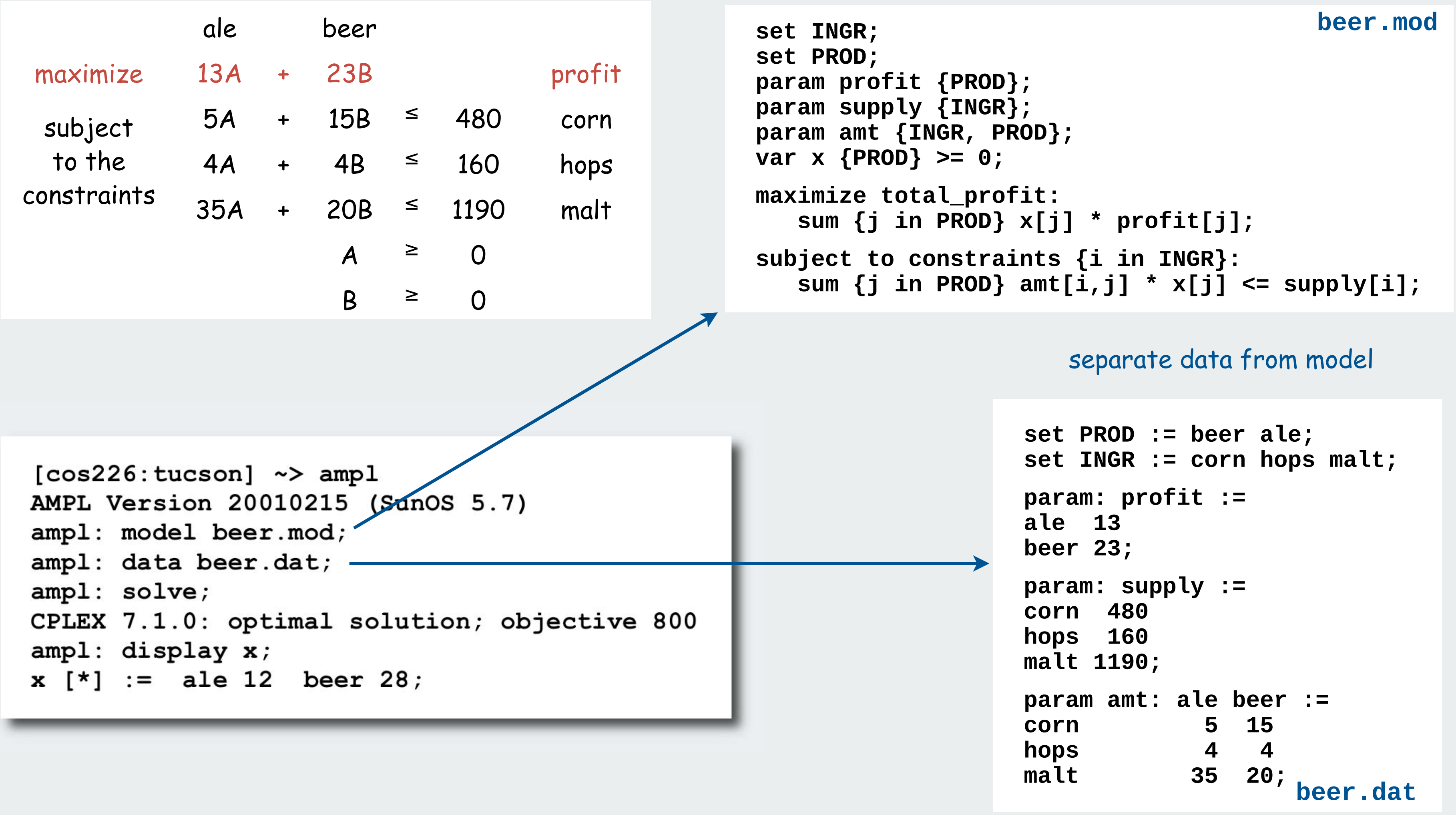
```
import drasys.or.mp.*;
import drasys.or.mp.lp.*;

public class LPDemo
{
   public static void main(String[] args) throws Exception
   {
      Problem prob = new Problem(3, 2);
      prob.getMetadata().put("lp.isMaximize", "true");
      prob.newVariable("x1").setObjectiveCoefficient(13.0);
      prob.newVariable("x2").setObjectiveCoefficient(23.0);
      prob.newConstraint("corn").setRightHandSide( 480.0);
      prob.newConstraint("hops").setRightHandSide( 160.0);
      prob.newConstraint("malt").setRightHandSide(1190.0);

      prob.setCoefficientAt("corn", "x1",  5.0);
      prob.setCoefficientAt("corn", "x2", 15.0);
      prob.setCoefficientAt("hops", "x1",  4.0);
      prob.setCoefficientAt("hops", "x2",  4.0);
      prob.setCoefficientAt("malt", "x1", 35.0);
      prob.setCoefficientAt("malt", "x2", 20.0);

      DenseSimplex lp = new DenseSimplex(prob);
      System.out.println(lp.solve());
      System.out.println(lp.getSolution());
   }
}
```

Ex. 2: MS Excel (!)

LP solvers: commercial strength

AMPL. [Fourer, Gay, Kernighan] An algebraic modeling language.

CPLEX solver. Industrial strength solver.



	ale		beer			
maximize	13A	+	23B			profit
subject to the constraints	5A	+	15B	≤	480	corn
	4A	+	4B	≤	160	hops
	35A	+	20B	≤	1190	malt
			A	≥	0	
			B	≥	0	

beer.mod

```
set INGR;
set PROD;
param profit {PROD};
param supply {INGR};
param amt {INGR, PROD};
var x {PROD} >= 0;

maximize total_profit:
   sum {j in PROD} x[j] * profit[j];

subject to constraints {i in INGR}:
   sum {j in PROD} amt[i,j] * x[j] <= supply[i];
```

separate data from model

```
[cos226:tucson] ~> ampl
AMPL Version 20010215 (SunOS 5.7)
ampl: model beer.mod;
ampl: data beer.dat;
ampl: solve;
CPLEX 7.1.0: optimal solution; objective 800
ampl: display x;
x [*] :=  ale 12  beer 28;
```

```
set PROD := beer ale;
set INGR := corn hops malt;

param: profit :=
ale  13
beer 23;

param: supply :=
corn  480
hops  160
malt 1190;

param amt: ale beer :=
corn        5   15
hops        4    4
malt       35   20;
```

beer.dat

History

1939. Production, planning. [Kantorovich]

1947. Simplex algorithm. [Dantzig]

1950. Applications in many fields.

1979. Ellipsoid algorithm. [Khachian]

1984. Projective scaling algorithm. [Karmarkar]

1990. Interior point methods.

- Interior point faster when polyhedron smooth like disco ball.
- Simplex faster when polyhedron spiky like quartz crystal.

200x. Approximation algorithms, large scale optimization.

- brewer's problem
- simplex algorithm
- implementation
- linear programming

Linear programming

Linear "programming"

- process of formulating an LP model for a problem
- solution to LP for a specific problem gives solution to the problem

1. Identify variables
2. Define constraints (inequalities and equations)
3. Define objective function

easy part [omitted]:
convert to standard form

Examples:

- shortest paths
- maxflow
- bipartite matching
- .
- .
- .
- [a very long list]

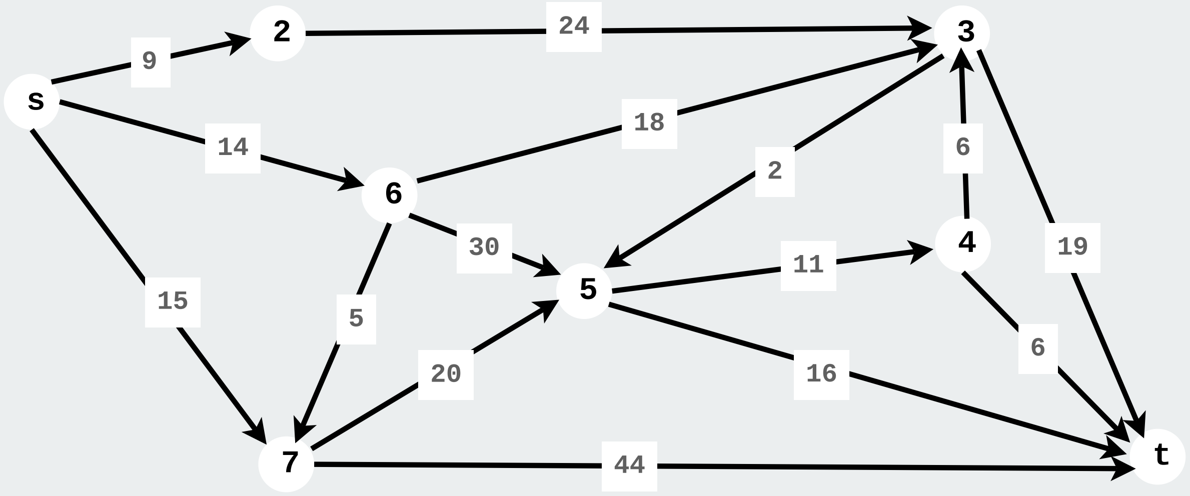
stay tuned [this lecture]

Single-source shortest-paths problem (revisited)

Given. Weighted digraph, single source s.

Distance from s to v: length of the shortest path from s to v .

Goal. Find distance (and shortest path) from s to every other vertex.



LP formulation of single-source shortest-paths problem

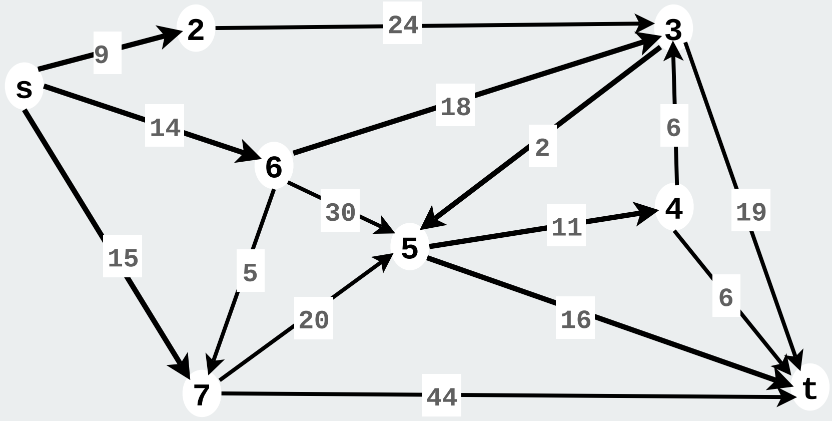
One variable per vertex, one inequality per edge.

minimize x_t

subject to the constraints

$$
\begin{aligned}
x_s + 9 &\le x_2\\
x_s + 14 &\le x_6\\
x_s + 15 &\le x_7\\
x_2 + 24 &\le x_3\\
x_3 + 2 &\le x_5\\
x_3 + 19 &\le x_t\\
x_4 + 6 &\le x_3\\
x_4 + 6 &\le x_t\\
x_5 + 11 &\le x_4\\
x_5 + 16 &\le x_t\\
x_6 + 18 &\le x_3\\
x_6 + 30 &\le x_5\\
x_6 + 5 &\le x_7\\
x_7 + 20 &\le x_5\\
x_7 + 44 &\le x_t\\
x_s &= 0\\
x_2, \dots, x_t &\ge 0
\end{aligned}
$$

interpretation:
x_i = length of shortest path from source to i



LP formulation of single-source shortest-paths problem

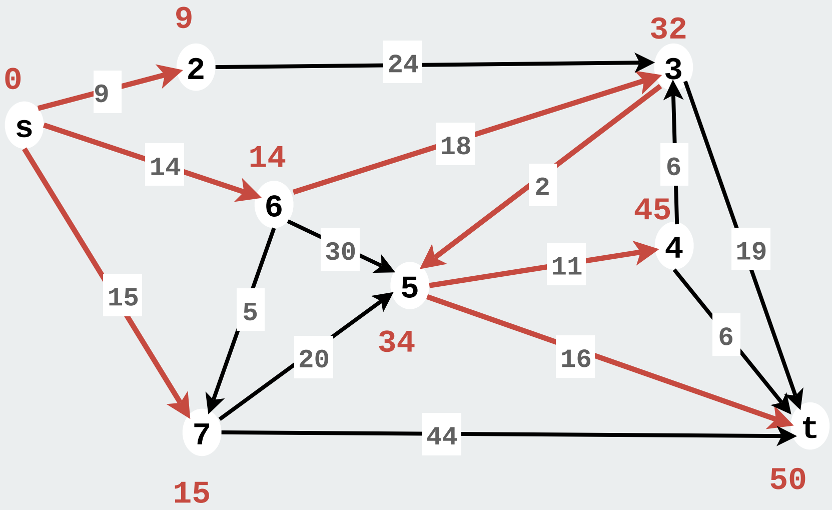
One variable per vertex, one inequality per edge.

minimize x_t

subject to the constraints

$$x_s + 9 \le x_2$$
$$x_s + 14 \le x_6$$
$$x_s + 15 \le x_7$$
$$x_2 + 24 \le x_3$$
$$x_3 + 2 \le x_5$$
$$x_3 + 19 \le x_t$$
$$x_4 + 6 \le x_3$$
$$x_4 + 6 \le x_t$$
$$x_5 + 11 \le x_4$$
$$x_5 + 16 \le x_t$$
$$x_6 + 18 \le x_3$$
$$x_6 + 30 \le x_5$$
$$x_6 + 5 \le x_7$$
$$x_7 + 20 \le x_5$$
$$x_7 + 44 \le x_t$$
$$x_s = 0$$
$$x_2, \ldots, x_t \ge 0$$

interpretation: x_i = length of shortest path from source to i



solution

$x_s = 0$

$x_2 = 9$

$x_3 = 32$

$x_4 = 45$

$x_5 = 34$

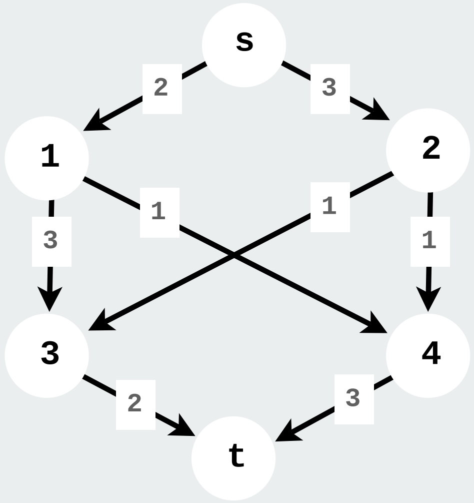
$x_6 = 14$

$x_7 = 15$

$x_t = 50$

Maxflow problem

Given: Weighted digraph, source s, destination t.

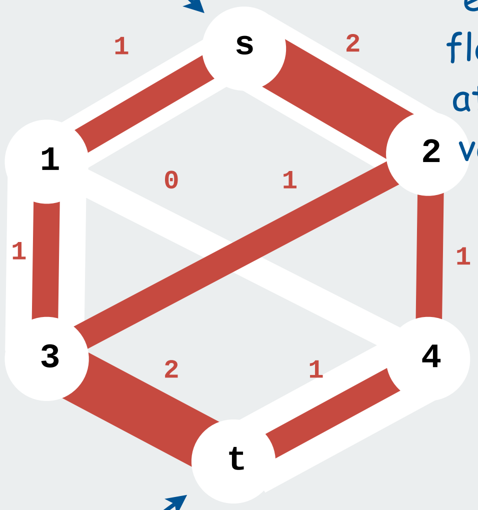


Interpret edge weights as capacities

- Models material flowing through network
- Ex: oil flowing through pipes
- Ex: goods in trucks on roads
- [many other examples]

Flow: A different set of edge weights

- flow does not exceed capacity in any edge
- flow at every vertex satisfies equilibrium [flow in equals flow out]



flow out of s is 3

flow in equals flow out at each vertex

flow ≤ capacity in every edge

flow in to t is 3

Goal: Find maximum flow from s to t

LP formulation of maxflow problem

One variable per edge.

One inequality per edge, one equality per vertex.

maximize x_{ts}

subject to the constraints

$$x_{s1} \le 2$$
$$x_{s2} \le 3$$
$$x_{13} \le 3$$
$$x_{14} \le 1$$
$$x_{23} \le 1$$
$$x_{24} \le 1$$
$$x_{3t} \le 2$$
$$x_{4t} \le 3$$

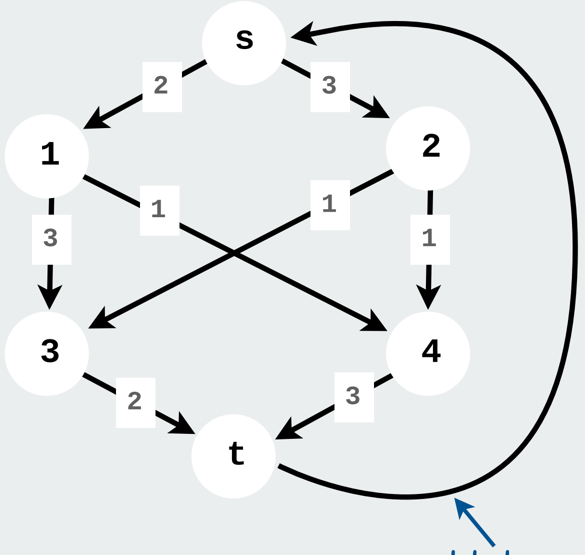
capacity constraints

$$x_{ts} = x_{s1} + x_{s2}$$
$$x_{s1} = x_{13} + x_{14}$$
$$x_{s2} = x_{23} + x_{24}$$
$$x_{13} + x_{23} = x_{3t}$$
$$x_{14} + x_{24} = x_{4t}$$
$$x_{3t} + x_{4t} = x_{ts}$$

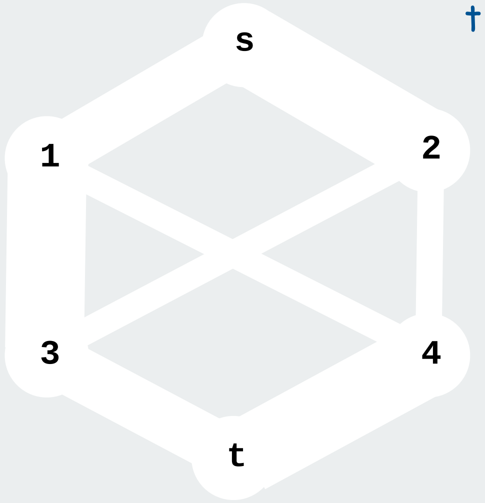
equilibrium constraints

all $x_{ij} \ge 0$

interpretation:
x_{ij} = flow in edge i-j



add dummy edge from t to s



LP formulation of maxflow problem

One variable per edge.
One inequality per edge, one equality per vertex.

maximize x_{ts}

subject to the constraints

$$
\begin{aligned}
x_{s1} &\le 2\\
x_{s2} &\le 3\\
x_{13} &\le 3\\
x_{14} &\le 1\\
x_{23} &\le 1\\
x_{24} &\le 1\\
x_{3t} &\le 2\\
x_{4t} &\le 3
\end{aligned}
$$

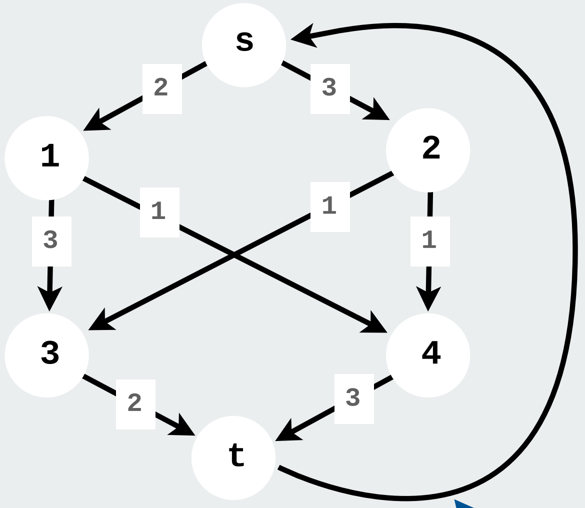
capacity constraints

interpretation: x_{ij} = flow in edge i-j

equilibrium constraints

$$
\begin{aligned}
x_{ts} &= x_{s1} + x_{s2}\\
x_{s1} &= x_{13} + x_{14}\\
x_{s2} &= x_{23} + x_{24}\\
x_{13} + x_{23} &= x_{3t}\\
x_{14} + x_{24} &= x_{4t}\\
x_{3t} + x_{4t} &= x_{ts}
\end{aligned}
$$

all $x_{ij} \ge 0$

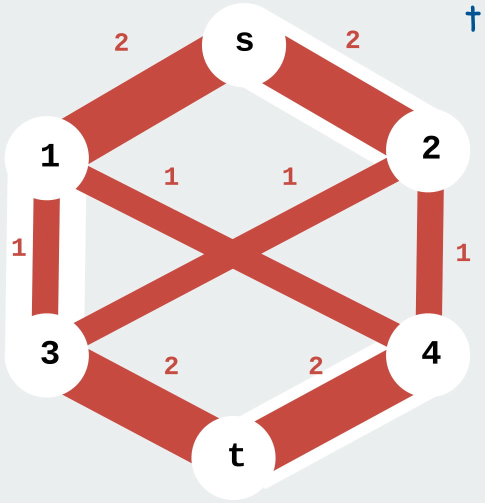


add dummy edge from t to s

solution

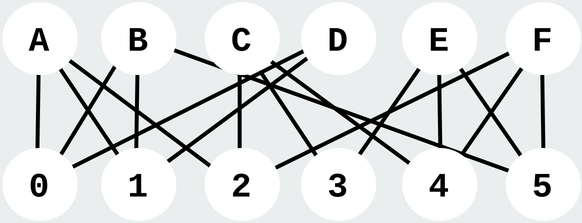
$$
\begin{aligned}
x_{s1} &= 2\\
x_{s2} &= 2\\
x_{13} &= 1\\
x_{14} &= 1\\
x_{23} &= 1\\
x_{24} &= 1\\
x_{3t} &= 2\\
x_{4t} &= 2\\
x_{ts} &= 4
\end{aligned}
$$

maxflow value



Maximum cardinality bipartite matching problem

Given: Two sets of vertices, set of edges
(each connecting one vertex in each set)



Matching: set of edges
with no vertex appearing twice

Interpretation: mutual preference constraints

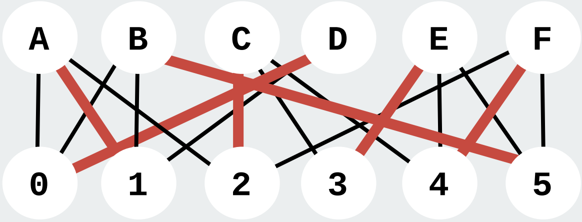
- Ex: people to jobs
- Ex: medical students to residence positions
- Ex: students to writing seminars
- [many other examples]

Alice
Adobe, Apple, Google
Bob
Adobe, Apple, Yahoo
Carol
Google, IBM, Sun
Dave
Adobe, Apple
Eliza
IBM, Sun, Yahoo
Frank
Google, Sun, Yahoo

Adobe
Alice, Bob, Dave
Apple
Alice, Bob, Dave
Google
Alice, Carol, Frank
IBM
Carol, Eliza
Sun
Carol, Eliza, Frank
Yahoo
Bob, Eliza, Frank

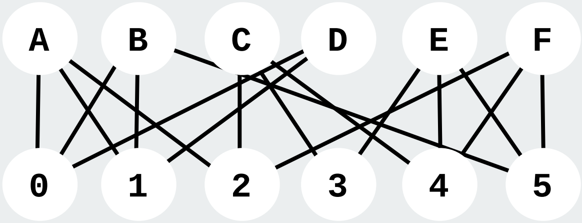
Example: Job offers

Goal: find a maximum cardinality matching



LP formulation of maximum cardinality bipartite matching problem

One variable per edge, one equality per vertex.



$$\text{maximize}\quad \begin{array}{l} x_{A0} + x_{A1} + x_{A2} + x_{B0} + x_{B1} + x_{B5} \\ + x_{C2} + x_{C3} + x_{C4} + x_{D0} + x_{D1} \\ + x_{E3} + x_{E4} + x_{E5} + x_{F2} + x_{F4} + x_{F5} \end{array}$$

subject to the constraints

$$\left.\begin{array}{r} x_{A0} + x_{A1} + x_{A2} = 1 \\ x_{B0} + x_{B1} + x_{B5} = 1 \\ x_{C2} + x_{C3} + x_{C4} = 1 \\ x_{D0} + x_{D1} = 1 \\ x_{E3} + x_{E4} + x_{E5} = 1 \\ x_{F2} + x_{F4} + x_{F5} = 1 \end{array}\right\}\ \text{constraints on top vertices}$$

$$\left.\begin{array}{r} x_{A0} + x_{B0} + x_{D0} = 1 \\ x_{A1} + x_{B1} + x_{D1} = 1 \\ x_{A2} + x_{C2} + x_{F2} = 1 \\ x_{C3} + x_{E3} = 1 \\ x_{C4} + x_{E4} + x_{F4} = 1 \\ x_{B5} + x_{E5} + x_{F5} = 1 \end{array}\right\}\ \text{constraints on bottom vertices}$$

$$\text{all } x_{ij} \geq 0$$

interpretation:
An edge is in the matching iff $x_{ij} = 1$

Crucial point:
not always so lucky!

Theorem. [Birkhoff 1946, von Neumann 1953]
All extreme points of the above polyhedron have integer (0 or 1) coordinates

Corollary. Can solve bipartite matching problem by solving LP

LP formulation of maximum cardinality bipartite matching problem

One variable per edge, one equality per vertex.

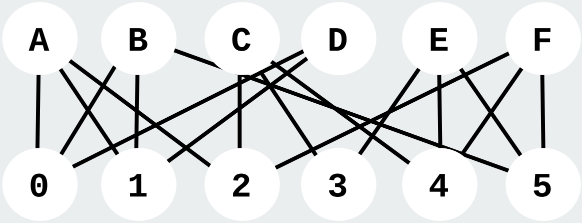
maximize

$$x_{A0} + x_{A1} + x_{A2} + x_{B0} + x_{B1} + x_{B5} + x_{C2} + x_{C3} + x_{C4} + x_{D0} + x_{D1} + x_{E3} + x_{E4} + x_{E5} + x_{F2} + x_{F4} + x_{F5}$$

subject to the constraints

$$\begin{aligned}
x_{A0} + x_{A1} + x_{A2} &= 1 \\
x_{B0} + x_{B1} + x_{B5} &= 1 \\
x_{C2} + x_{C3} + x_{C4} &= 1 \\
x_{D0} + x_{D1} &= 1 \\
x_{E3} + x_{E4} + x_{E5} &= 1 \\
x_{F2} + x_{F4} + x_{F5} &= 1 \\
x_{A0} + x_{B0} + x_{D0} &= 1 \\
x_{A1} + x_{B1} + x_{D1} &= 1 \\
x_{A2} + x_{C2} + x_{F2} &= 1 \\
x_{C3} + x_{E3} &= 1 \\
x_{C4} + x_{E4} + x_{F4} &= 1 \\
x_{B5} + x_{E5} + x_{F5} &= 1 \\
\text{all } x_{ij} &\ge 0
\end{aligned}$$

interpretation: An edge is in the matching iff $x_{ij} = 1$



solution

$x_{A1} = 1$

$x_{B5} = 1$

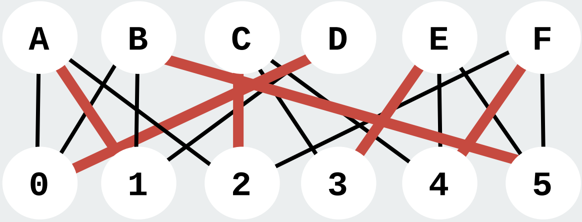
$x_{C2} = 1$

$x_{D0} = 1$

$x_{E3} = 1$

$x_{F4} = 1$

all other $x_{ij} = 0$



Linear programming perspective

Got an optimization problem?
ex: shortest paths, maxflow, matching, . . . [many, many, more]

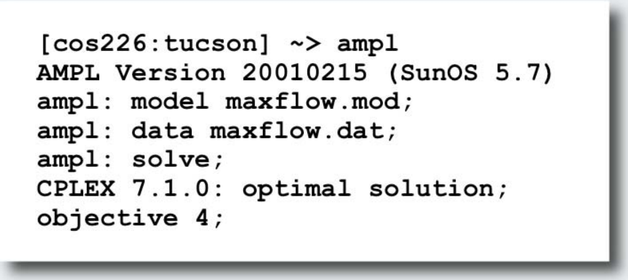
Approach 1: Use a specialized algorithm to solve it

- Algs in Java
- vast literature on complexity
- performance on real problems not always well-understood

Approach 2: Use linear programming

- a direct mathematical representation of the problem often works
- immediate solution to the problem at hand is often available
- might miss specialized solution, but might not care

Got an LP solver? Learn to use it!



```
[cos226:tucson] ~> ampl
AMPL Version 20010215 (SunOS 5.7)
ampl: model maxflow.mod;
ampl: data maxflow.dat;
ampl: solve;
CPLEX 7.1.0: optimal solution;
objective 4;
```

LP: the ultimate problem-solving model (in practice)

Fact 1: Many practical problems are easily formulated as LPs
Fact 2: Commercial solvers can solve those LPs quickly

More constraints on the problem?

- specialized algorithm may be hard to fix
- can just add more inequalities to LP

Ex. Mincost maxflow and other generalized versions

New problem?

- may not be difficult to formulate LP
- may be very difficult to develop specialized algorithm

Today's problem?

- similar to yesterday's
- edit tableau, run solver

Ex. Airline scheduling
[similar to vast number of other business processes]

Too slow?

- could happen
- doesn't happen

Want to learn more?
ORFE 307

Ultimate problem-solving model (in theory)

Is there an ultimate problem-solving model?

- Shortest paths
- Maximum flow
- Bipartite matching
- . . .
- Linear programming
- .
- .
- .
- NP-complete problems
- .
- .
- .

tractable

intractable ?

[see next lecture]

Does P = NP? No universal problem-solving model exists unless P = NP.

Want to learn more?
COS 423

LP perspective

LP is near the deep waters of intractability.

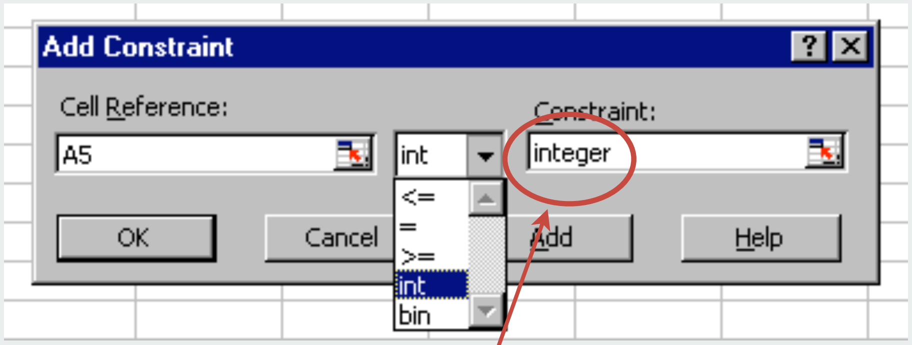
Good news:

- LP has been widely used for large practical problems for 50+ years
- Existence of guaranteed poly-time algorithm known for 25+ years.

Bad news:

constrain variables to have integer values

- Integer linear programming is NP-complete
- (existence of guaranteed poly-time algorithm is highly unlikely).
- [stay tuned]



Add Constraint

Cell Reference: A5

int

Constraint: integer

An unsuspecting MBA student transitions to
the world of intractability with a single mouse click.

Reductions

- designing algorithms
- proving limits
- classifying problems
- NP-completeness

Bird's-eye view

Desiderata.
Classify problems according to their computational requirements.

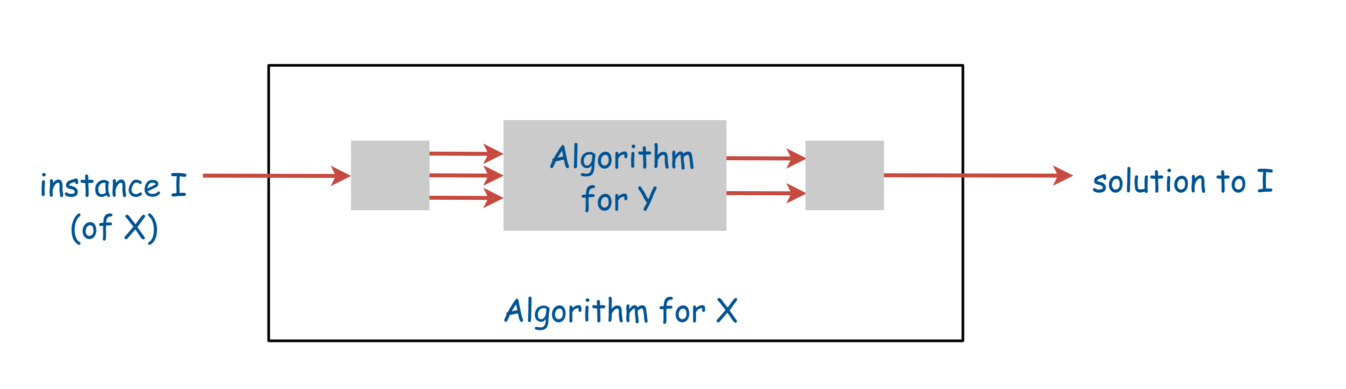
Frustrating news.
Huge number of fundamental problems have defied classification

Desiderata'.
Suppose we could (couldn't) solve problem X efficiently.
What else could (couldn't) we solve efficiently?

> Give me a lever long enough and a fulcrum on which to place it, and I shall move the world. -Archimedes

Reduction

Def. Problem X reduces to problem Y
if you can use an algorithm that solves Y to help solve X



Ex. Euclidean MST reduces to Voronoi.

To solve Euclidean MST on N points

- solve Voronoi for those points
- construct graph with linear number of edges
- use Prim/Kruskal to find MST in time proportional to N log N

Reduction

Def. Problem X reduces to problem Y
if you can use an algorithm that solves Y to help solve X

Cost of solving X = M*(cost of solving Y) + cost of reduction.

↑
number of times Y is used

Applications

- designing algorithms: given algorithm for Y, can also solve X.
- proving limits: if X is hard, then so is Y.
- classifying problems: establish relative difficulty of problems.

- designing algorithms
- proving limits
- classifying problems
- NP-completeness

Reductions for algorithm design

Def. Problem X reduces to problem Y if you can use an algorithm that solves Y to help solve X

Cost of solving X = M*(cost of solving Y) + cost of reduction.

↑ number of times Y is used

Applications.

- designing algorithms: given algorithm for Y, can also solve X.
- proving limits: if X is hard, then so is Y.
- classifying problems: establish relative difficulty of problems.

Mentality: Since I know how to solve Y, can I use that algorithm to solve X?

↑ Programmer's version: I have code for Y. Can I use it for X?

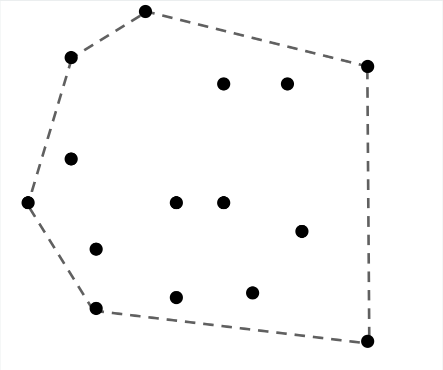
Reductions for algorithm design: convex hull

Sorting. Given N distinct integers, rearrange them in ascending order.

Convex hull. Given N points in the plane, identify the extreme points of the convex hull (in counter-clockwise order).

Claim. Convex hull reduces to sorting.

Pf. Graham scan algorithm.



convex hull

```
1251432
2861534
3988818
4190745
13546464
89885444
```

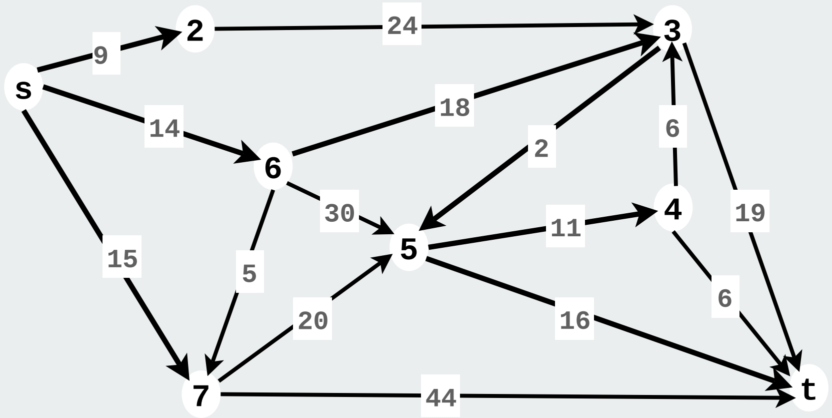
sorting

Cost of convex hull = cost of sort + cost of reduction

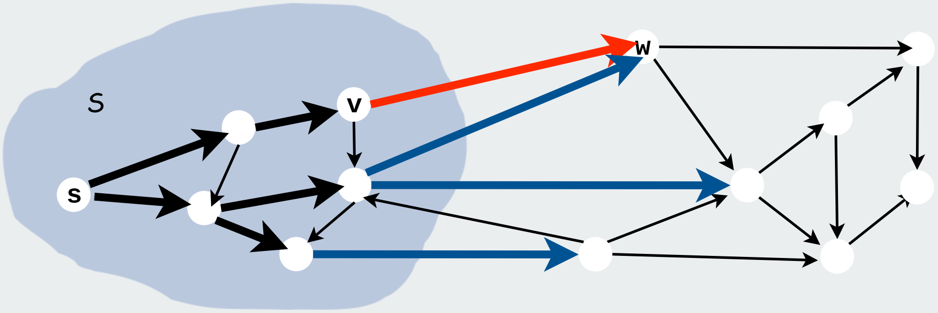
linearithmic linear

Reductions for algorithm design: shortest paths

Claim. Shortest paths reduces to path search in graphs (PFS)



Pf. Dijkstra's algorithm



Cost of shortest paths = cost of search + cost of reduction

linear

length of path

Reductions for algorithm design: maxflow

Claim: Maxflow reduces to PFS (!)

A forward edge is an edge in the same direction of the flow

An backward edge is an edge in the opposite direction of the flow

An augmenting path is along which we can increase flow by adding flow on a forward edge or decreasing flow on a backward edge

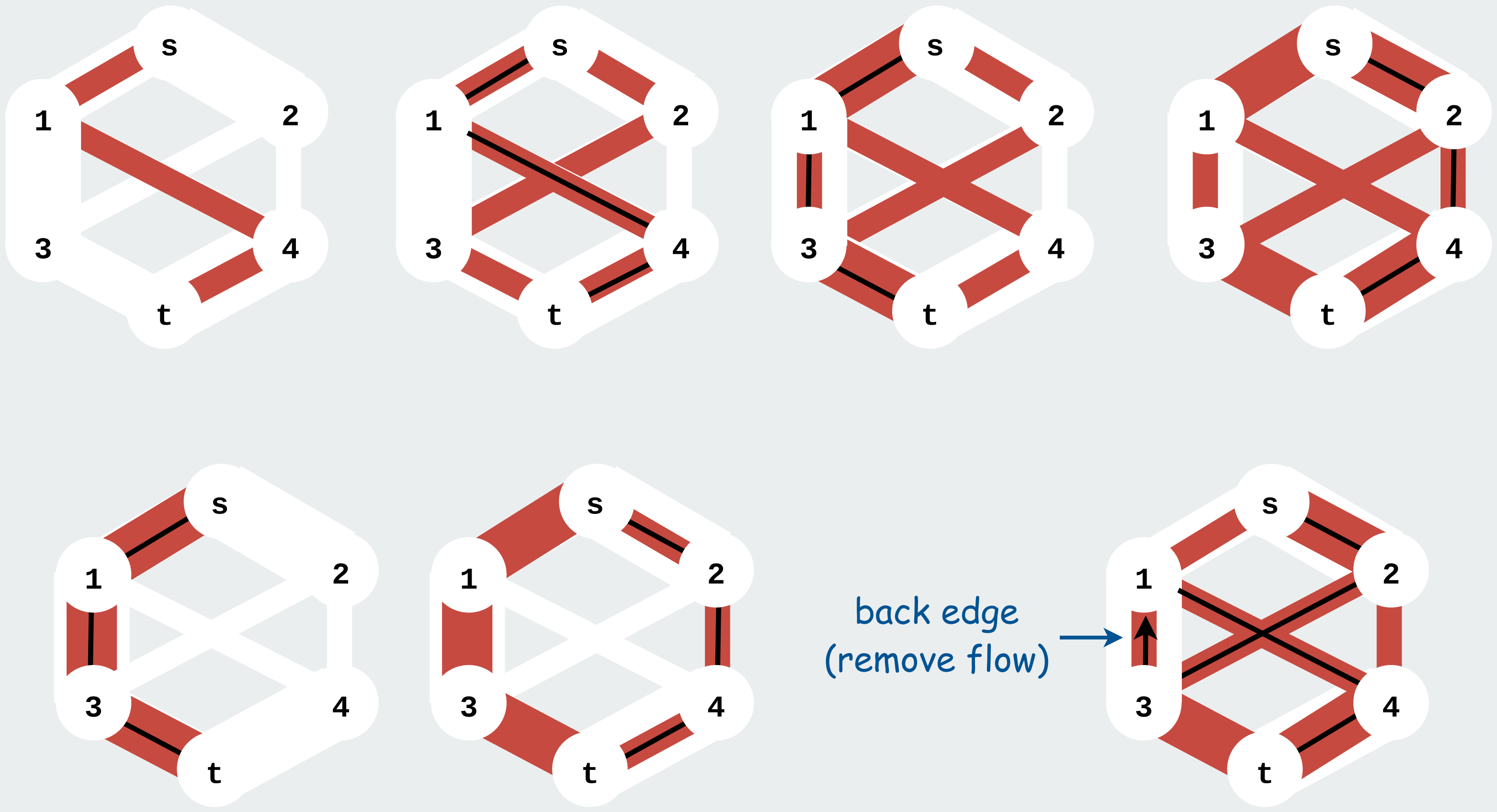
Theorem [Ford-Fulkerson] To find maxflow:

- increase flow along any augmenting path
- continue until no augmenting path can be found

Reduction is not linear because it requires multiple calls to PFS

Reductions for algorithm design: maxflow (continued)

Two augmenting-path sequences



back edge
(remove flow)

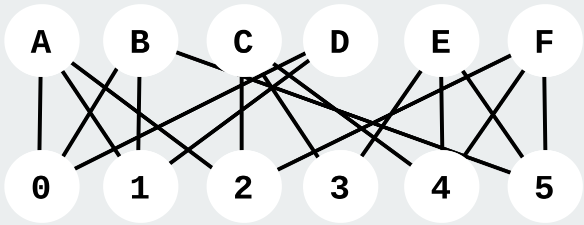
Cost of maxflow = M*(cost of PFS) + cost of reduction

linear linear

depends on path choice!

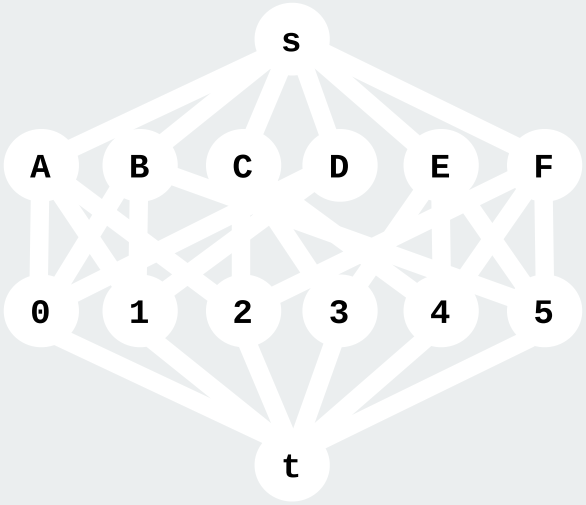
Reductions for algorithm design: bipartite matching

Bipartite matching reduces to maxflow



Proof:

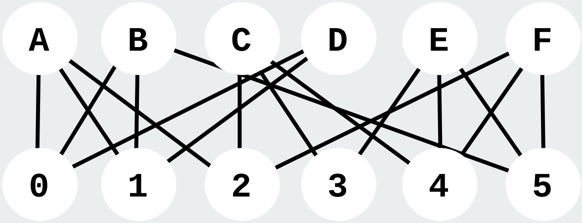
- construct new vertices s and t
- add edges from s to each vertex in one set
- add edges from each vertex in other set to t
- set all edge weights to 1
- find maxflow in resulting network
- matching is edges between two sets



Note: Need to establish that maxflow solution has all integer (0-1) values.

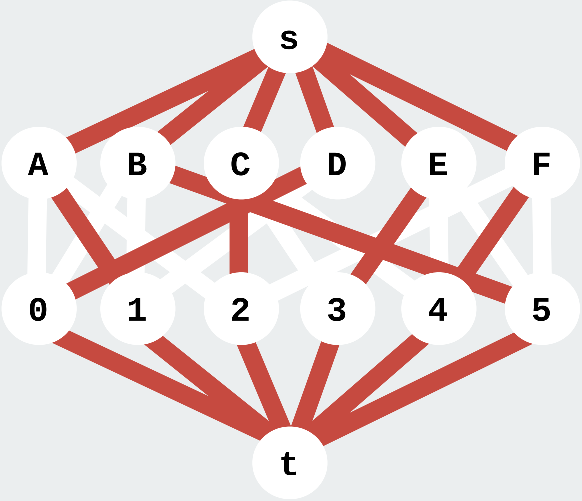
Reductions for algorithm design: bipartite matching

Bipartite matching reduces to maxflow

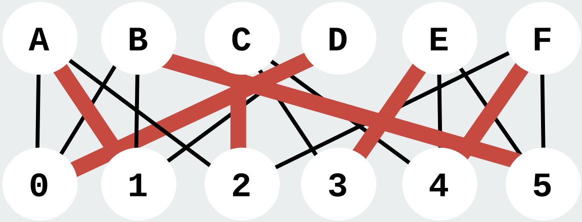


Proof:

- construct new vertices s and t
- add edges from s to each vertex in one set
- add edges from each vertex in other set to t
- set all edge weights to 1
- find maxflow in resulting network
- matching is edges between two sets



Note: Need to establish that maxflow solution has all integer (0-1) values.

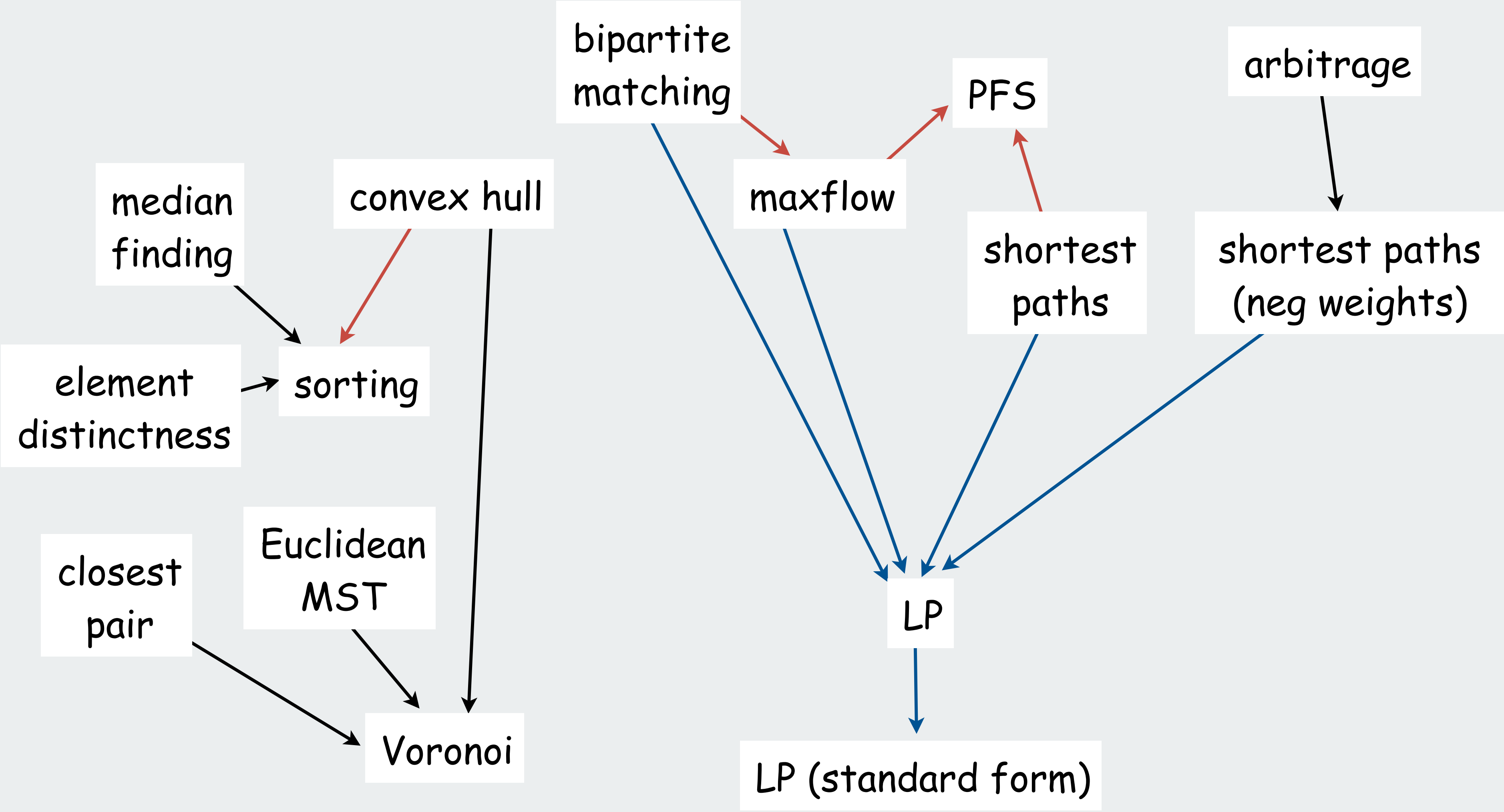


Cost of matching = cost of maxflow + cost of reduction

linear

Reductions for algorithm design: summary

Some reductions we have seen so far:



Reductions for algorithm design: a caveat

PRIME. Given an integer x (represented in binary), is x prime?

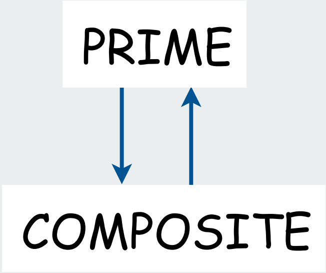
COMPOSITE. Given an integer x, does x have a nontrivial factor?

PRIME reduces to COMPOSITE

```
public static boolean isPrime(BigInteger x)
{
   if (isComposite(x)) return false;
   else                return true;
}
```

COMPOSITE reduces to PRIME

```
public static boolean isComposite(BigInteger x)
{
   if (isPrime(x)) return false;
   else            return true;
}
```



A possible real-world scenario:

- System designer specs the interfaces for project.
- Programmer A implements `isComposite()` using `isPrime()`.
- Programmer B implements `isPrime()` using `isComposite()`.
- Infinite reduction loop!

whose fault?

- designing algorithms
- **proving limits**
- classifying problems
- polynomial-time reductions
- NP-completeness

Linear-time reductions to prove limits

Def. Problem X linear reduces to problem Y if X can be solved with:

- linear number of standard computational steps for reduction
- one call to subroutine for Y.

Applications.

- designing algorithms: given algorithm for Y, can also solve X.
- proving limits: if X is hard, then so is Y.
- classifying problems: establish relative difficulty of problems.

Mentality:

If I could easily solve Y, then I could easily solve X
I can't easily solve X.
Therefore, I can't easily solve Y

Purpose of reduction is to establish that Y is hard

NOT intended for use as an algorithm

Proving limits on convex-hull algorithms

Lower bound on sorting: Sorting N integers requires Ω(N log N) steps.

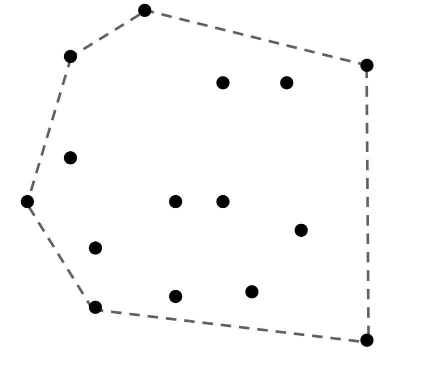
need "quadratic decision tree" model of computation that allows tests of the form $x_i < x_j$ or $(x_j - x_i)(y_k - y_i) - (y_j - y_i)(x_k - x_i) < 0$

Claim. SORTING reduces to CONVEX HULL [see next slide].

Consequence.
Any ccw-based convex hull algorithm requires Ω(N log N) steps.

1251432
2861534
3988818
4190745
13546464
89885444

sorting

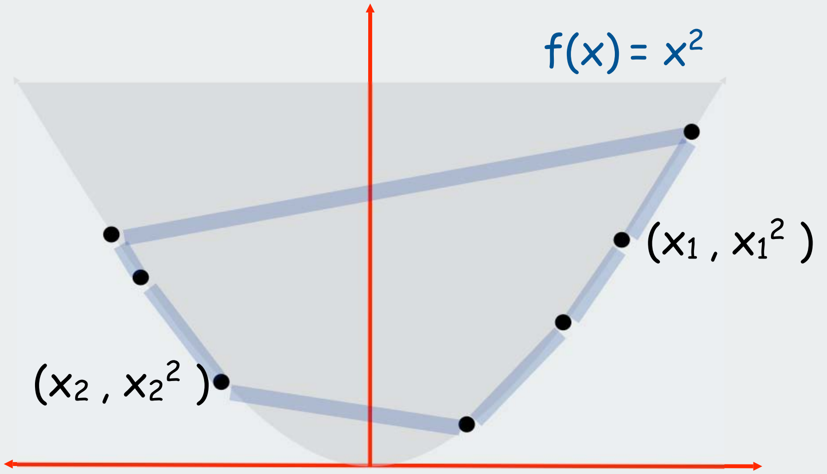


convex hull

Sorting linear-reduces to convex hull

Sorting instance. $X = \{ x_1, x_2, \ldots , x_N \}$

Convex hull instance. $P = \{ (x_1 , x_1^2), (x_2, x_2^2), \ldots , (x_N , x_N^2) \}$



Observation. Region $\{x : x^2 \geq x\}$ is convex $\Rightarrow$ all points are on hull.

Consequence. Starting at point with most negative x, counter-clockwise order of hull points yields items in ascending order.

To sort X, find the convex hull of P.

3-SUM reduces to 3-COLLINEAR

3-SUM. Given N distinct integers, are there three that sum to 0?

3-COLLINEAR. Given N distinct points in the plane, are there 3 that all lie on the same line?

recall Assignment 2

Claim. 3-SUM reduces to 3-COLLINEAR.

see next two slides

Conjecture. Any algorithm for 3-SUM requires $\Omega(N^2)$ time.

Consequence. Sub-quadratic algorithm for 3-COLLINEAR unlikely.

your $N^2 \log N$ algorithm from Assignment 2 was pretty good

3-SUM reduces to 3-COLLINEAR (continued)

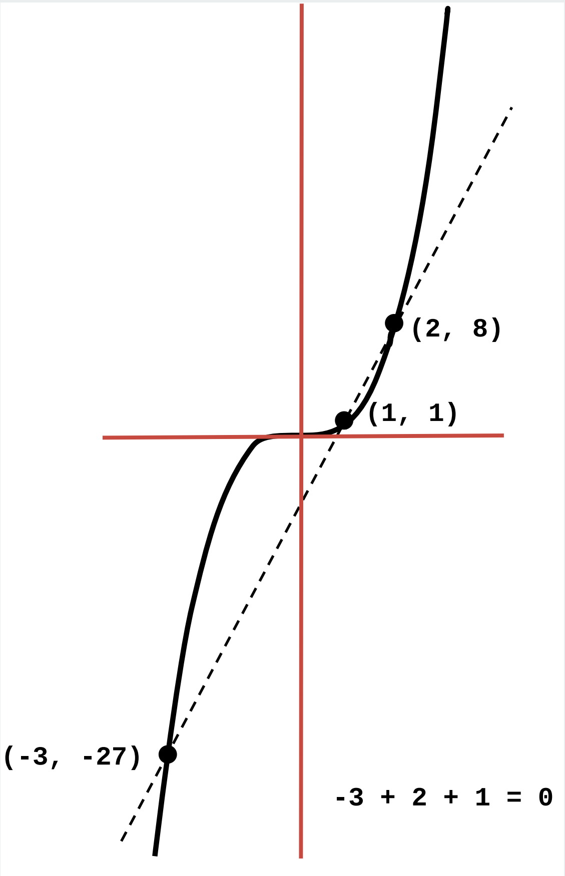
Claim. 3-SUM $\leq_L$ 3-COLLINEAR.

- 3-SUM instance: $x_1, x_2, \ldots, x_N$
- 3-COLLINEAR instance: $(x_1, x_1^3), (x_2, x_2^3), \ldots, (x_N, x_N^3)$

Lemma. If a, b, and c are distinct, then a + b + c = 0 if and only if (a, a^3), (b, b^3), (c, c^3) are collinear.

Pf. [see next slide]

$f(x) = x^3$



3-SUM reduces to 3-COLLINEAR (continued)

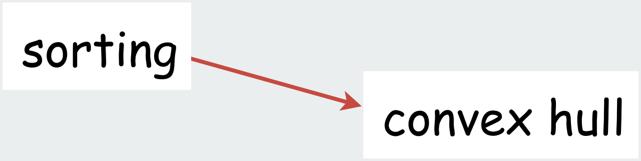
Lemma. If a, b, and c are distinct, then $a + b + c = 0$ if and only if (a, a^3), (b, b^3), (c, c^3) are collinear.

Pf. Three points (a, a^3), (b, b^3), (c, c^3) are collinear iff:

$$\begin{aligned}
(a^3 - b^3) / (a - b) &= (b^3 - c^3) / (b - c) && \text{slopes are equal} \\
(a - b)(a^2 + ab + b^2) / (a - b) &= (b - c)(b^2 + bc + c^2) / (b - c) && \text{factor numerators} \\
(a^2 + ab + b^2) &= (b^2 + bc + c^2) && \text{a-b and b-c are nonzero} \\
a^2 + ab - bc - c^2 &= 0 && \text{collect terms} \\
(a - c)(a + b + c) &= 0 && \text{factor} \\
a + b + c &= 0 && \text{a-c is nonzero}
\end{aligned}$$

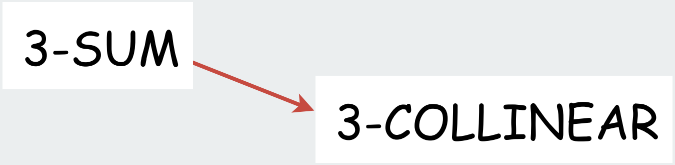
Reductions for proving limits: summary

Establishing limits through reduction is an important tool in guiding algorithm design efforts


sorting

convex hull

Want to be convinced that no linear-time convex hull alg exists?
Hard way: long futile search for a linear-time algorithm
Easy way: reduction from sorting


3-SUM

3-COLLINEAR

Want to be convinced that no subquadratic 3-COLLINEAR alg exists?
Hard way: long futile search for a subquadratic algorithm
Easy way: reduction from 3-SUM

- designing algorithms
- proving limits
- **classifying problems**
- NP-completeness

Reductions to classify problems

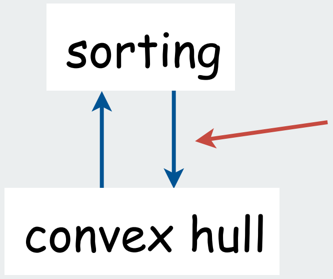
Def. Problem X linear reduces to problem Y if X can be solved with:

- Linear number of standard computational steps.
- One call to subroutine for Y.

Applications.

- Design algorithms: given algorithm for Y, can also solve X.
- Establish intractability: if X is hard, then so is Y.
- Classify problems: establish relative difficulty between two problems.

Ex: Sorting linear-reduces to convex hull.
Convex hull linear-reduces to sorting.
Thus, sorting and convex hull are equivalent



not a loop because this reduction is not intended to give a sorting algorithm

Most often used to classify problems as either

- tractable (solvable in polynomial time)
- intractable (exponential time seems to be required)

Polynomial-time reductions

Def. Problem X polynomial reduces to problem Y if arbitrary instances of problem X can be solved using:

- Polynomial number of standard computational steps for reduction
- One call to subroutine for Y.

critical detail (not obvious why)

Notation. $X \le_P Y$.

Ex. Any linear reduction is a polynomial reduction.

Ex. All algorithms for which we know poly-time algorithms poly-time reduce to one another.

Poly-time reduction of X to Y makes sense
only when X or Y is not known to have a poly-time algorithm

Polynomial-time reductions for classifying problems

Goal. Classify and separate problems according to relative difficulty.

- tractable problems: can be solved in polynomial time.
- intractable problems: seem to require exponential time.

Establish tractability. If $X \leq_P Y$ and Y is tractable then so is X.

- Solve Y in polynomial time.
- Use reduction to solve X.

Establish intractability. If $Y \leq_P X$ and Y is intractable, then so is X.

- Suppose X can be solved in polynomial time.
- Then so could Y (through reduction).
- Contradiction. Therefore X is intractable.

Transitivity. If $X \leq_P Y$ and $Y \leq_P Z$ then $X \leq_P Z$.

Ex: all problems that reduce to LP are tractable

3-satisfiability

Literal: A Boolean variable or its negation. x_i or $\neg x_i$

Clause. A disjunction of 3 distinct literals. $C_j = (x_1 \vee \neg x_2 \vee x_3)$

Conjunctive normal form. A propositional formula Φ that is the conjunction of clauses. $\text{CNF} = (C_1 \wedge C_2 \wedge C_3 \wedge C_4)$

3-SAT. Given a CNF formula Φ consisting of k clauses over n literals, does it have a satisfying truth assignment?

yes instance

$$(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)$$

x_1	x_2	x_3	x_4
T	T	F	T

$$(\neg T \vee T \vee F) \wedge (T \vee \neg T \vee F) \wedge (\neg T \vee \neg T \vee \neg F) \wedge (\neg T \vee \neg T \vee T) \wedge (\neg T \vee F \vee T)$$

no instance

$$(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)$$

Applications: Circuit design, program correctness, [many others]

3-satisfiability is intractable

Good news: easy algorithm to solve 3-SAT

[check all possible solutions]

Bad news: running time is exponential in input size.

[there are 2^n possible solutions]

Worse news:

no algorithm that guarantees subexponential running time is known

Implication:

- suppose 3-SAT poly-reduces to a problem A
- poly-time algorithm for A would imply poly-time 3-SAT algorithm
- we suspect that no poly-time algorithm exists for A!

Want to be convinced that a new problem is intractable?

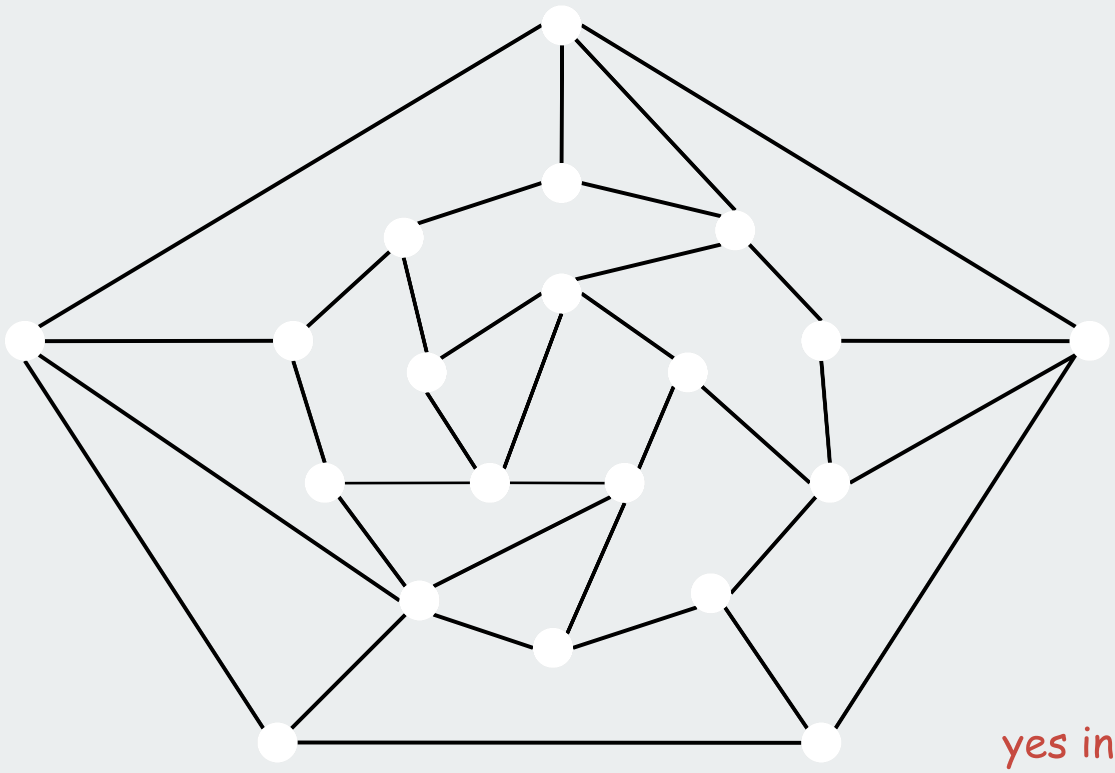
Hard way: long futile search for an efficient algorithm (as for 3-SAT)

Easy way: reduction from a known intractable problem (such as 3-SAT)

hence, intricate reductions are common

Graph 3-colorability

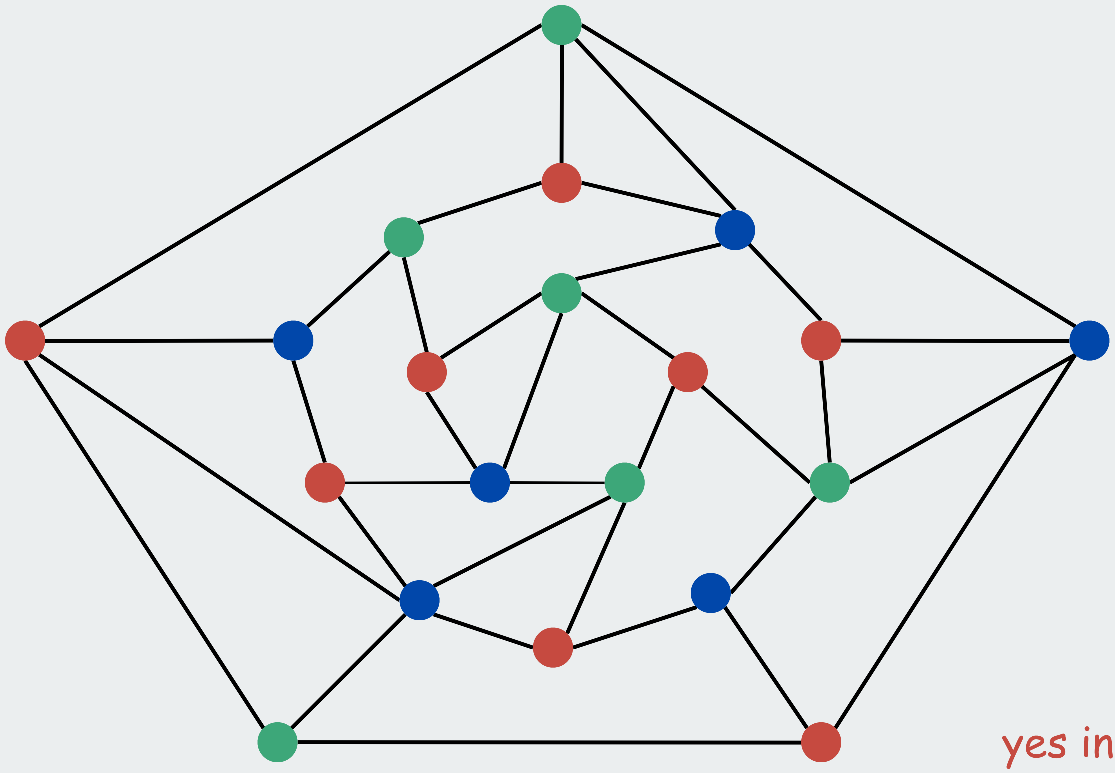
3-COLOR. Given a graph, is there a way to color the vertices red, green, and blue so that no adjacent vertices have the same color?



yes instance

Graph 3-colorability

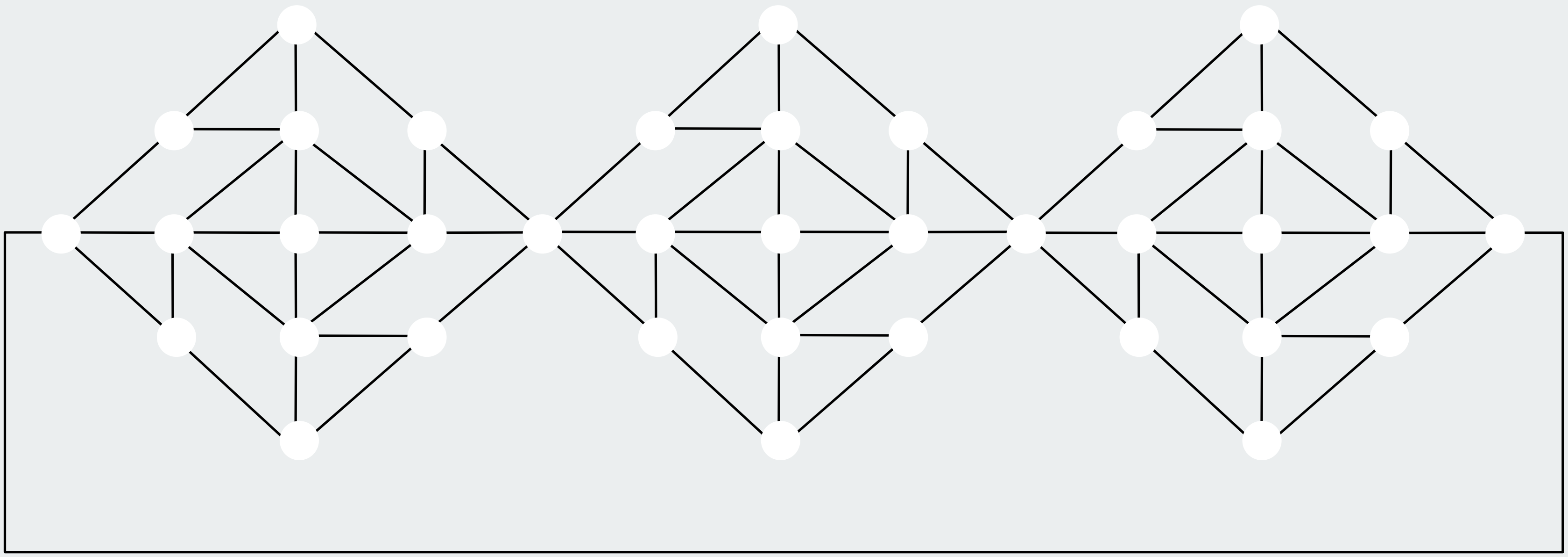
3-COLOR. Given a graph, is there a way to color the vertices red, green, and blue so that no adjacent vertices have the same color?



yes instance

Graph 3-colorability

3-COLOR. Given a graph, is there a way to color the vertices red, green, and blue so that no adjacent vertices have the same color?



no instance

3-satisfiability reduces to graph 3-colorability

Claim. 3-SAT $\le_P$ 3-COLOR.

Pf. Given 3-SAT instance Φ, we construct an instance of 3-COLOR that is 3-colorable if and only if Φ is satisfiable.

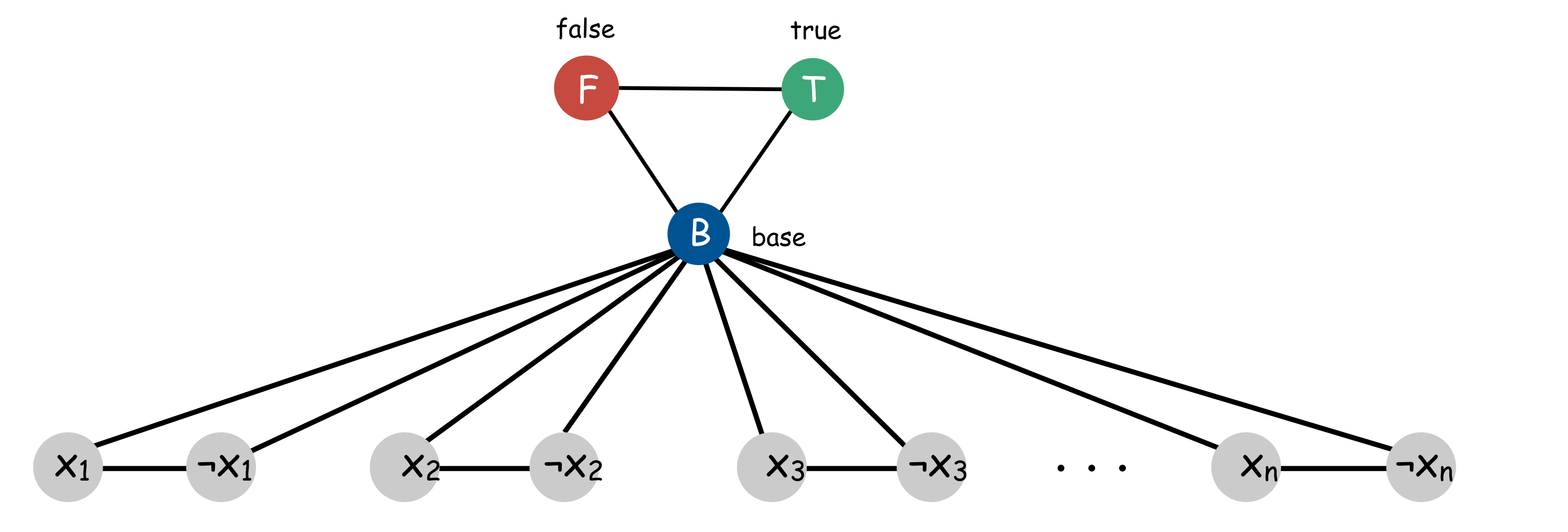
Construction.

(i) Create one vertex for each literal and 3 vertices F T B

(ii) Connect F T B in a triangle and connect each literal to B

(iii) Connect each literal to its negation.

(iv) For each clause, attach a 6-vertex gadget [details to follow].

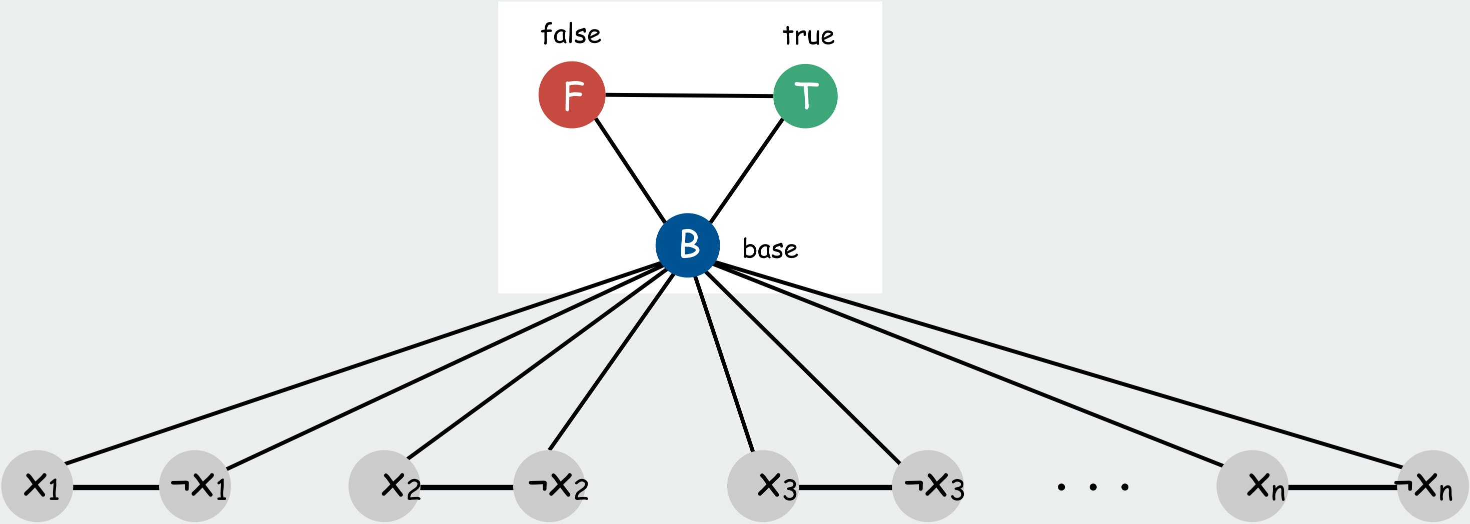


3-satisfiability reduces to graph 3-colorability

Claim. If graph is 3-colorable then Φ is satisfiable..

Pf.

- Consider assignment where F corresponds to false and T to true .
- (ii) [triangle] ensures each literal is true or false.

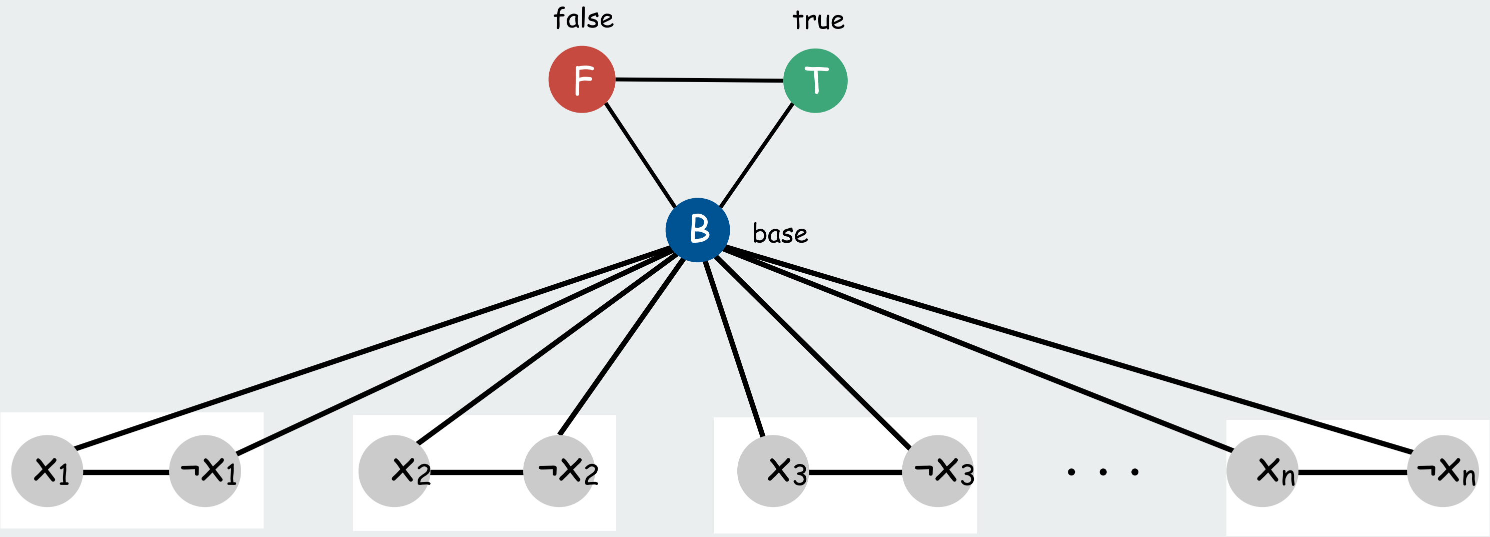


3-satisfiability reduces to graph 3-colorability

Claim. If graph is 3-colorable then Φ is satisfiable..

Pf. ⇒ Suppose graph is 3-colorable.

- Consider assignment where F corresponds to false and T to true .
- (ii) [triangle] ensures each literal is true or false.
- (iii) ensures a literal and its negation are opposites.



3-satisfiability reduces to graph 3-colorability

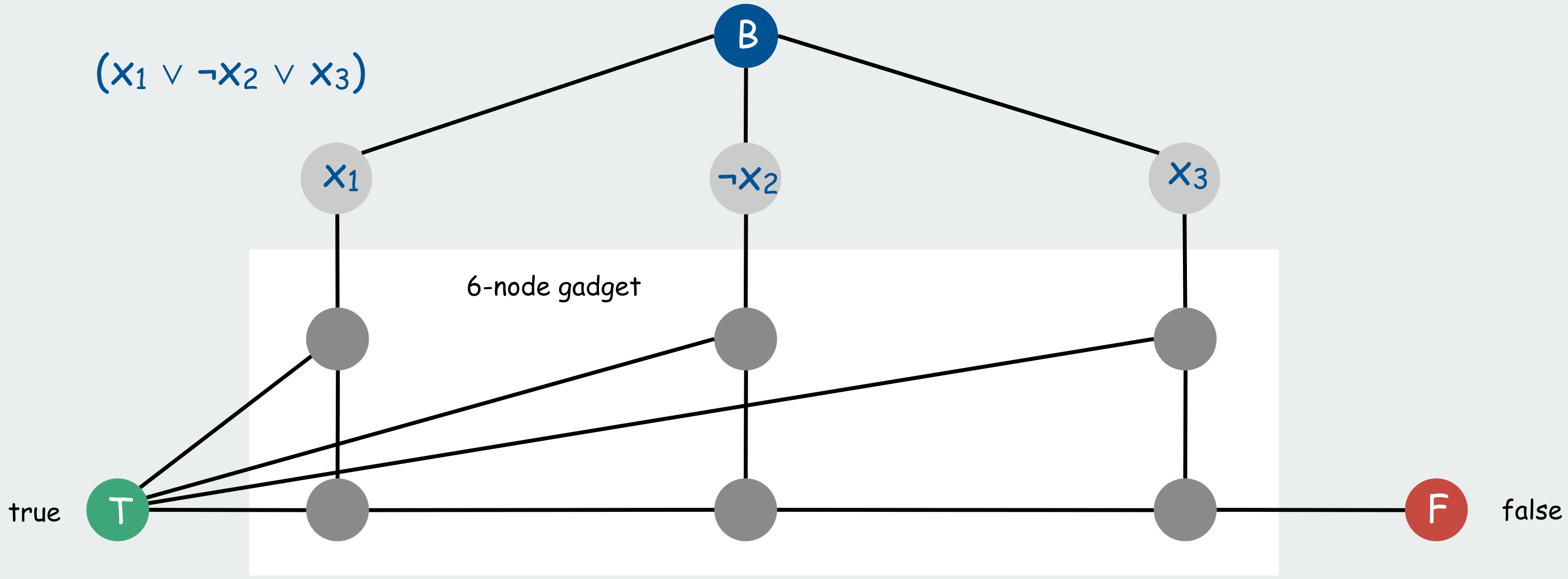
Claim. If graph is 3-colorable then Φ is satisfiable.

Pf.

- Consider assignment where F corresponds to false and T to true .
- (ii) [triangle] ensures each literal is true or false.
- (iii) ensures a literal and its negation are opposites.
- (iv) [gadget] ensures at least one literal in each clause is true.

stay tuned

$(x_1 \vee \neg x_2 \vee x_3)$



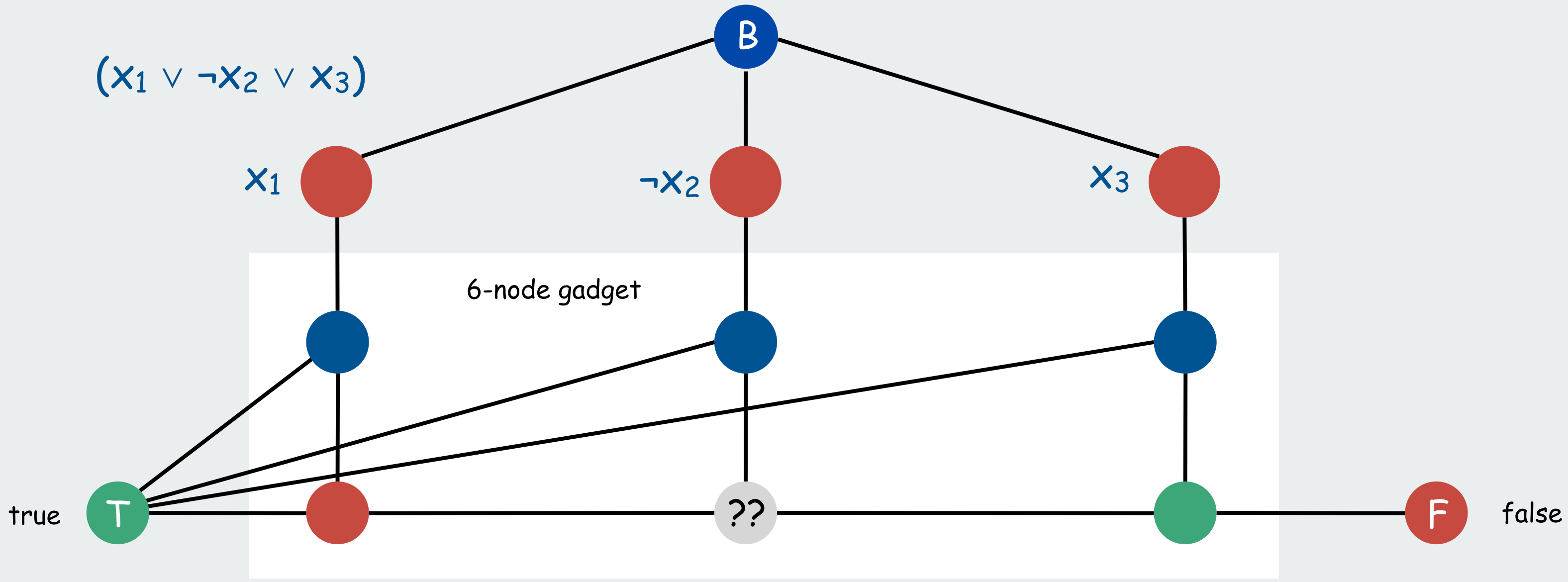
3-satisfiability reduces to graph 3-colorability

Claim. If graph is 3-colorable then Φ is satisfiable.

Pf.

- Consider assignment where F corresponds to false and F to true .
- (ii) [triangle] ensures each literal is true or false.
- (iii) ensures a literal and its negation are opposites.
- (iv) [gadget] ensures at least one literal in each clause is true.

Therefore, Φ is satisfiable.



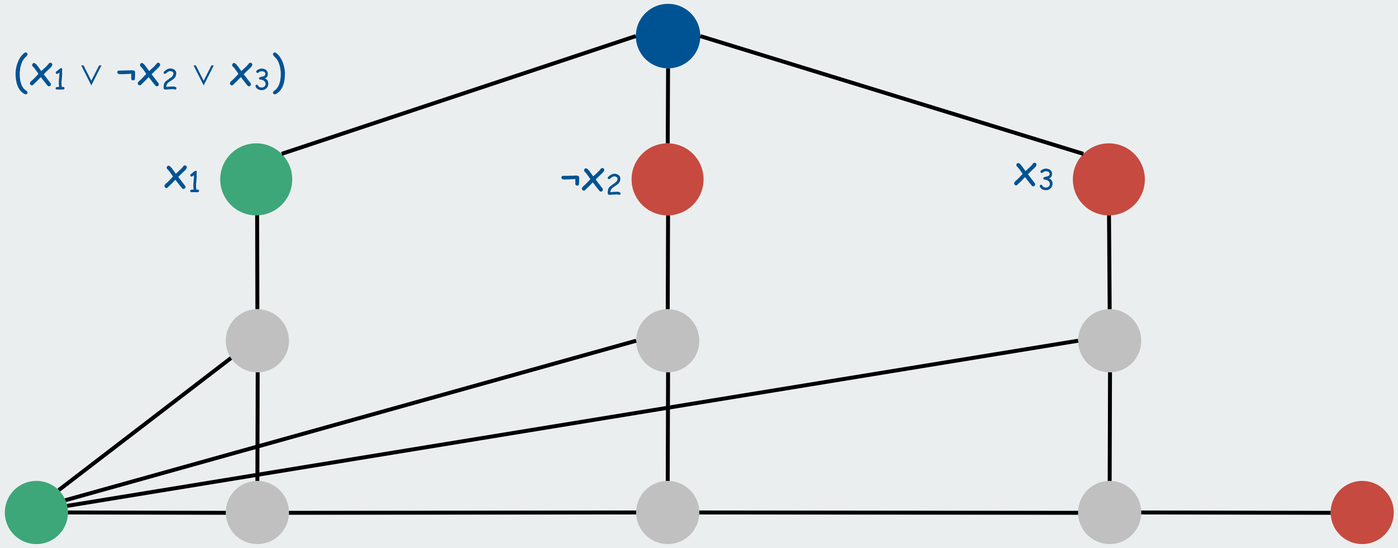
3-satisfiability reduces to graph 3-colorability

Claim. If Φ is satisfiable then graph is 3-colorable.

at least one
in each clause

Pf.

- Color nodes corresponding to false literals 🔴 and to true literals 🟢.

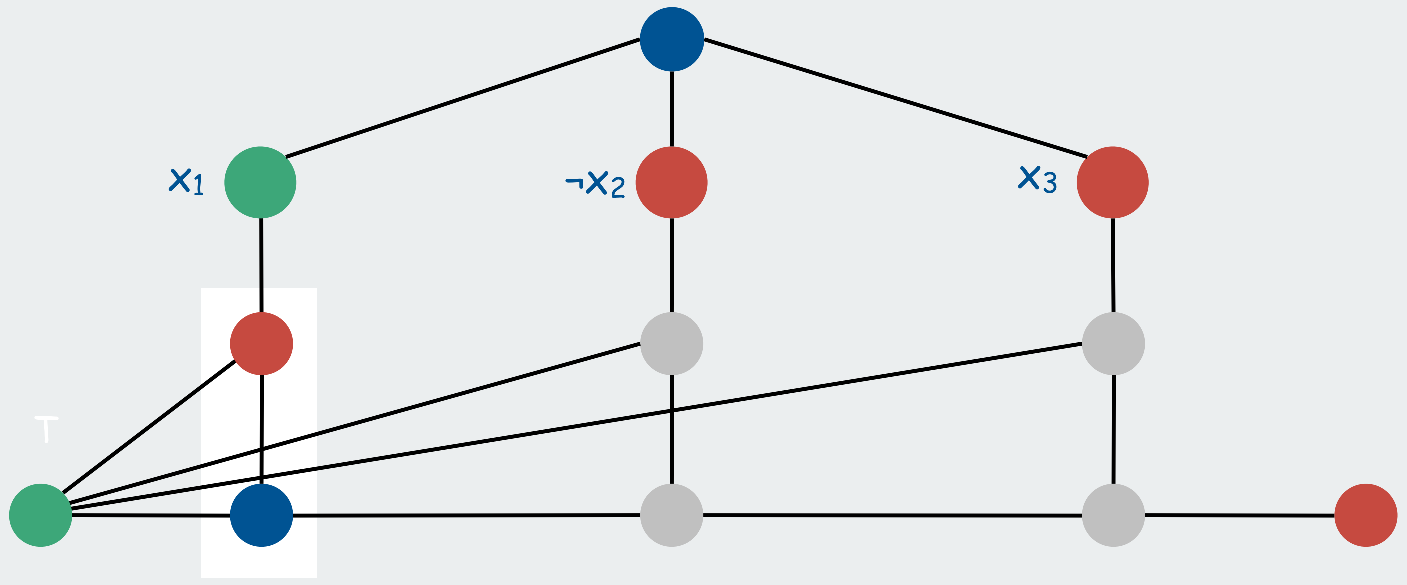


3-satisfiability reduces to graph 3-colorability

Claim. If Φ is satisfiable then graph is 3-colorable.

Pf.

- Color nodes corresponding to false literals 🔴 and to true literals 🟢.
- Color vertex below one 🟢 vertex 🔴, and vertex below that 🔵.

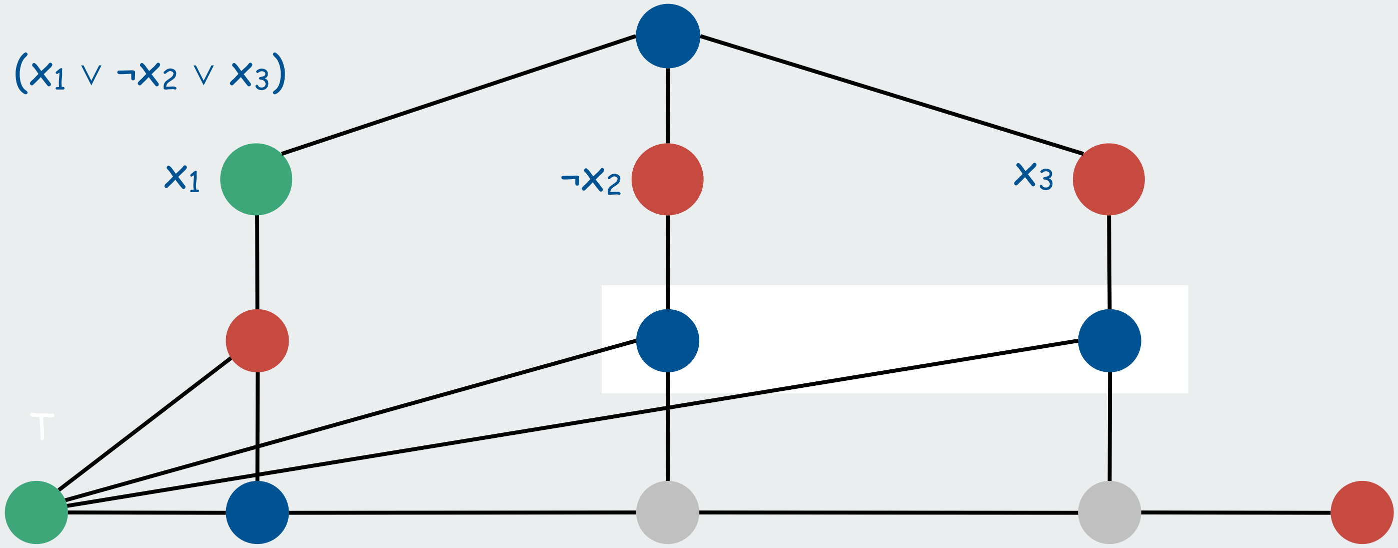


3-satisfiability reduces to graph 3-colorability

Claim. If Φ is satisfiable then graph is 3-colorable.

Pf.

- Color nodes corresponding to false literals 🔴 and to true literals 🟢.
- Color vertex below one 🟢 vertex 🔴, and vertex below that 🔵.
- Color remaining middle row vertices 🔵.



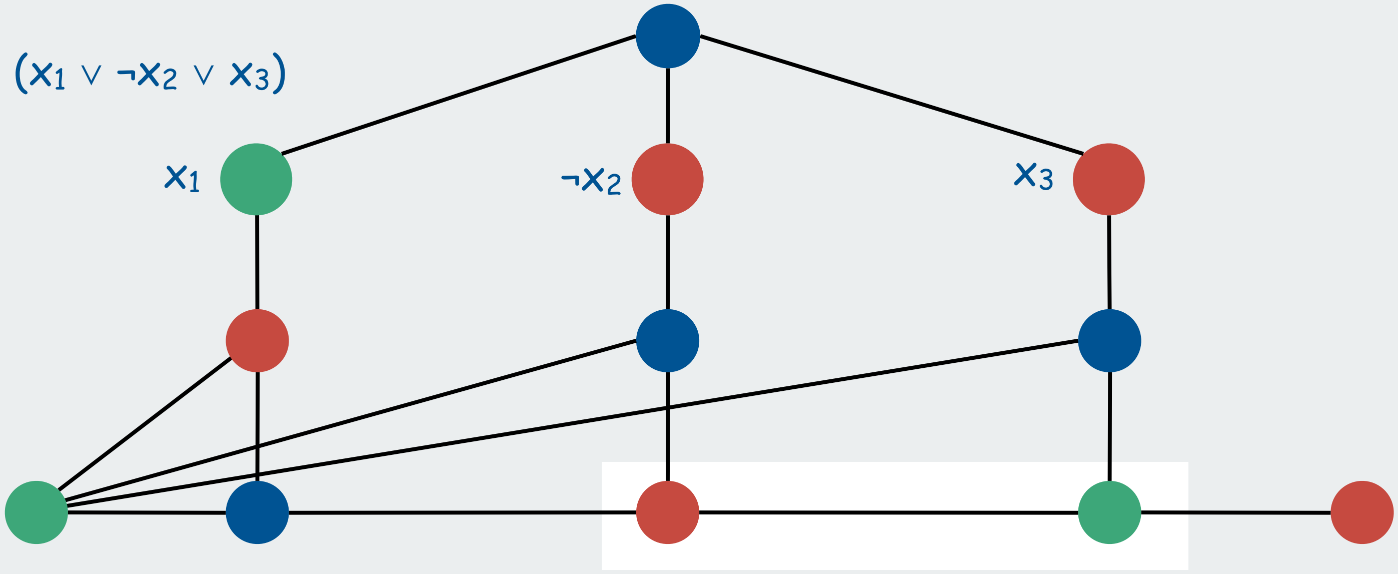
3-satisfiability reduces to graph 3-colorability

Claim. If Φ is satisfiable then graph is 3-colorable.

Pf.

- Color nodes corresponding to false literals 🔴 and to true literals 🟢.
- Color vertex below one 🟢 vertex 🔴, and vertex below that 🔵.
- Color remaining middle row vertices 🔵.
- Color remaining bottom vertices 🟢 or 🔴 as forced.

Works for all gadgets, so graph is 3-colorable. ▪



3-satisfiability reduces to graph 3-colorability

Claim. 3-SAT $\leq_P$ 3-COLOR.

Pf. Given 3-SAT instance Φ, we construct an instance of 3-COLOR that is 3-colorable if and only if Φ is satisfiable.

Construction.

(i) Create one vertex for each literal.

(ii) Create 3 new vertices T, F, and B; connect them in a triangle, and connect each literal to B.

(iii) Connect each literal to its negation.

(iv) For each clause, attach a gadget of 6 vertices and 13 edges

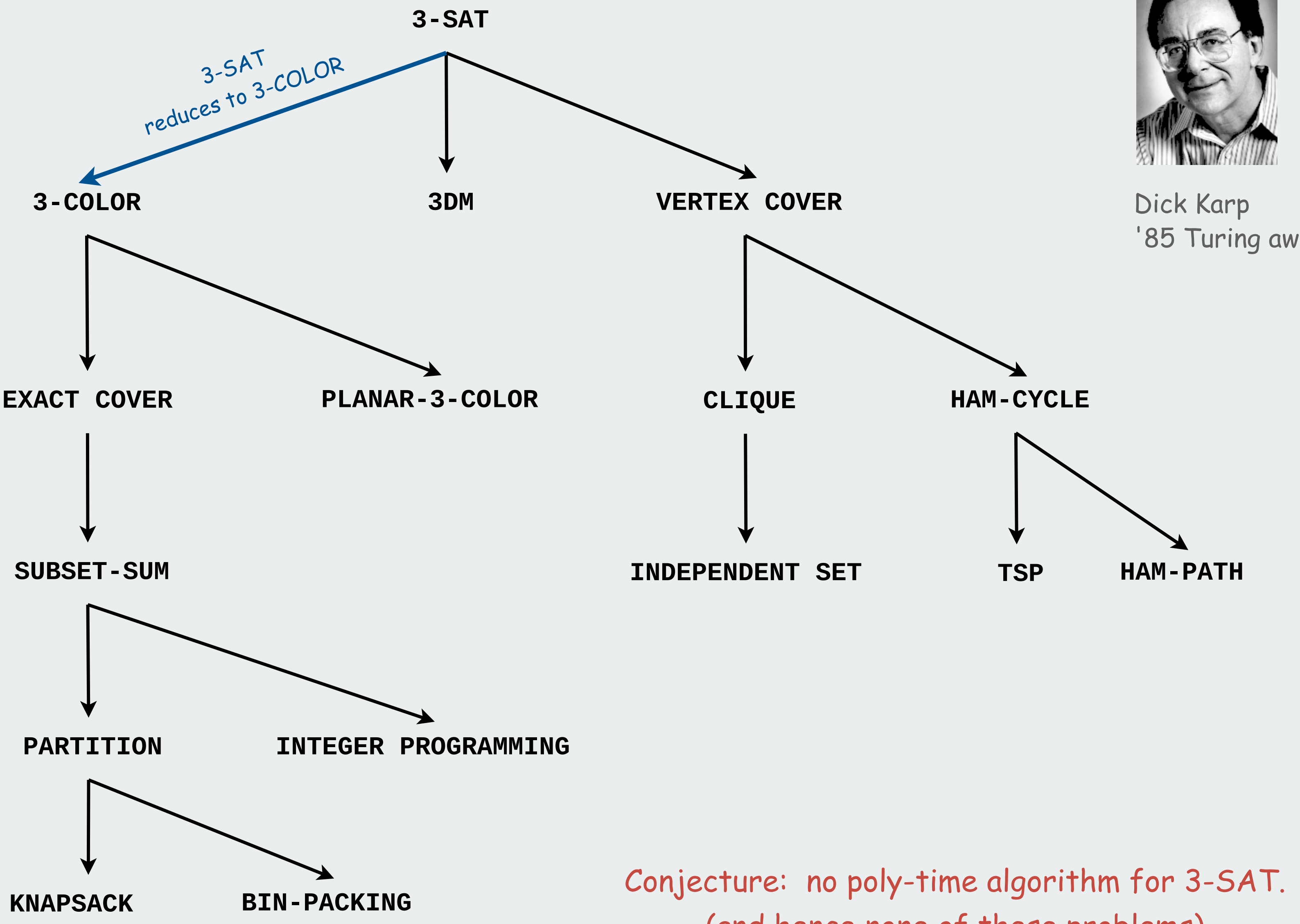
Conjecture: No polynomial-time algorithm for 3-SAT
Implication: No polynomial-time algorithm for 3-COLOR.

Reminder

Construction is not intended for use, just to prove 3-COLOR difficult

- designing algorithms
- proving limits
- classifying problems
- polynomial-time reductions
- **NP-completeness**

More Poly-Time Reductions



3-SAT

3-SAT reduces to 3-COLOR

3-COLOR

3DM

VERTEX COVER

EXACT COVER

PLANAR-3-COLOR

CLIQUE

HAM-CYCLE

SUBSET-SUM

INDEPENDENT SET

TSP

HAM-PATH

PARTITION

INTEGER PROGRAMMING

KNAPSACK

BIN-PACKING

Dick Karp
'85 Turing award

Conjecture: no poly-time algorithm for 3-SAT.
(and hence none of these problems)

Cook's Theorem

NP: set of problems solvable in polynomial time
by a nondeterministic Turing machine

THM. Any problem in NP $\leq_P$ 3-SAT.

Pf sketch.

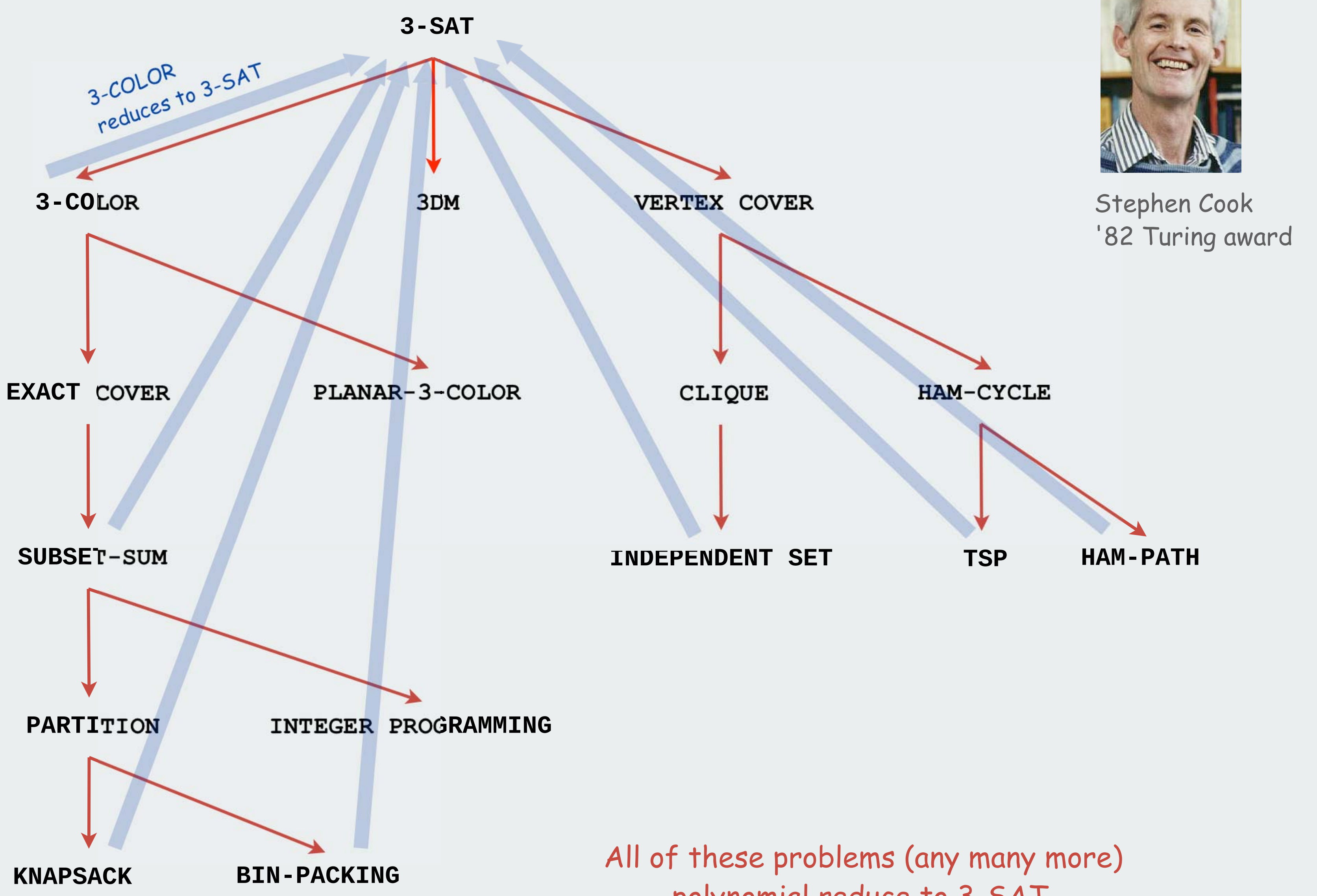
Each problem P in NP corresponds to a TM M that accepts or rejects any input in time polynomial in its size

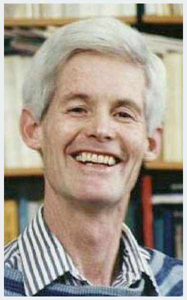
Given M and a problem instance I, construct an instance of 3-SAT that is satisfiable iff the machine accepts I.

Construction.

- Variables for every tape cell, head position, and state at every step.
- Clauses corresponding to each transition.
- [many details omitted]

Implications of Cook's theorem



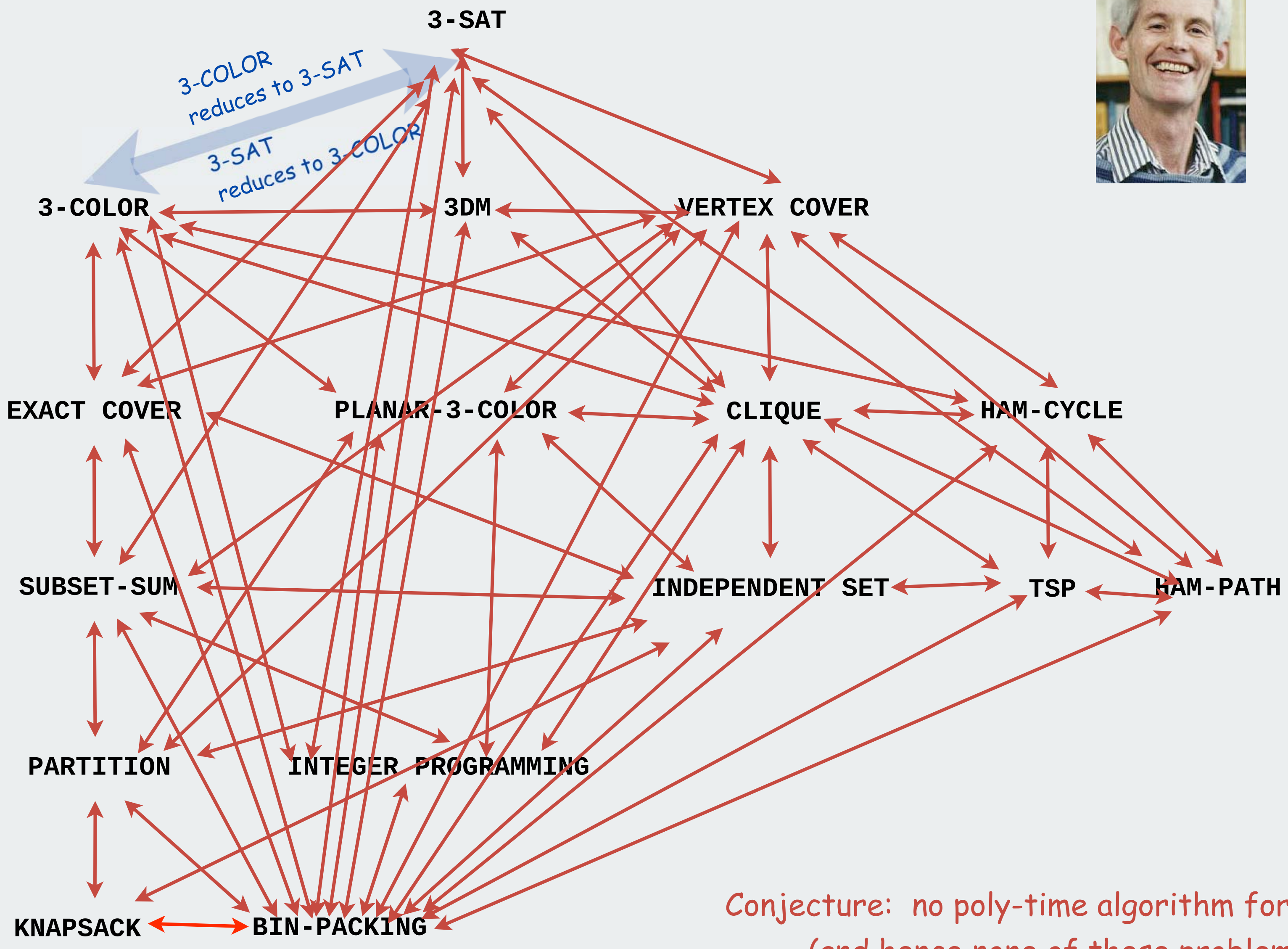


Stephen Cook
'82 Turing award

All of these problems (any many more) polynomial reduce to 3-SAT.

Implications of Karp + Cook

All of these problems poly-reduce to one another!



3-SAT

3-COLOR reduces to 3-SAT

3-SAT reduces to 3-COLOR

3-COLOR

3DM

VERTEX COVER

EXACT COVER

PLANAR-3-COLOR

CLIQUE

HAM-CYCLE

SUBSET-SUM

INDEPENDENT SET

TSP

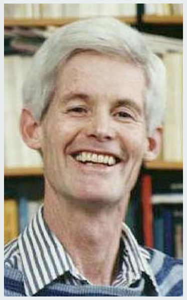
HAM-PATH

PARTITION

INTEGER PROGRAMMING

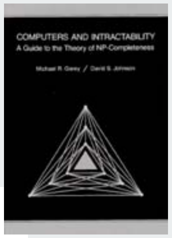
KNAPSACK

BIN-PACKING

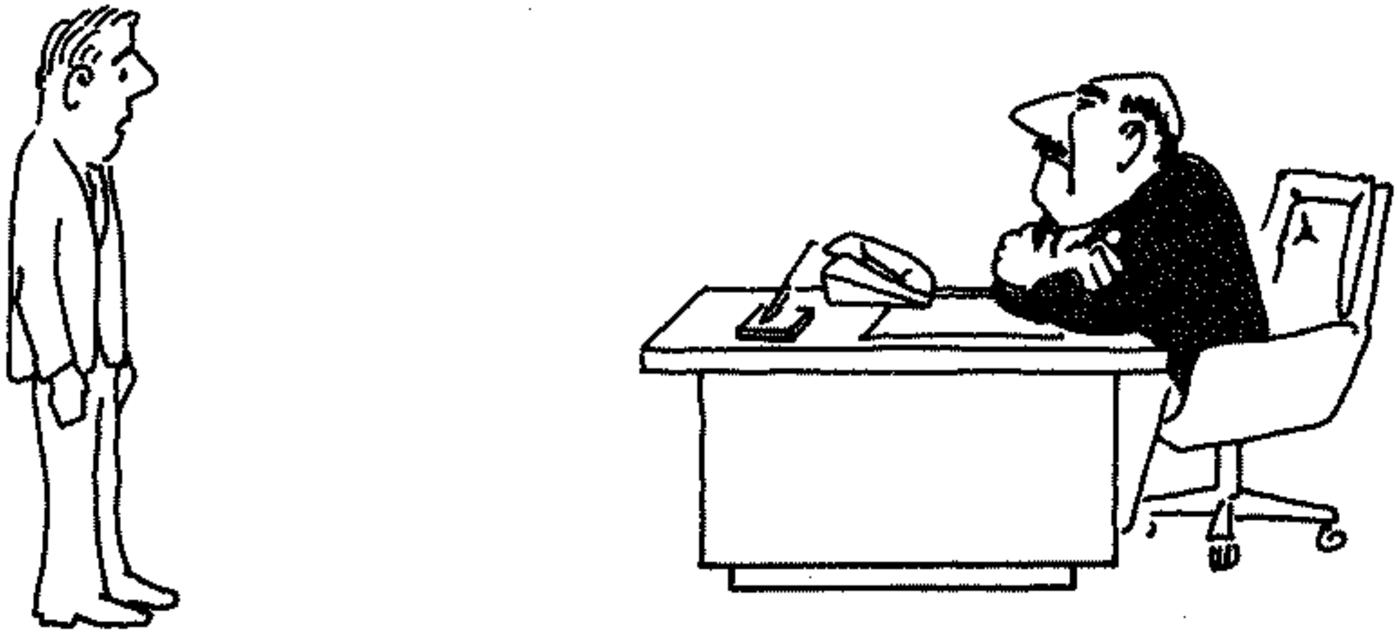


Conjecture: no poly-time algorithm for 3-SAT.
(and hence none of these problems)

Poly-Time Reductions: Implications

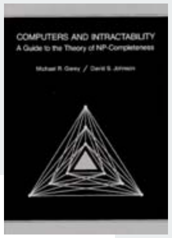


COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Michael R. Garey / David S. Johnson

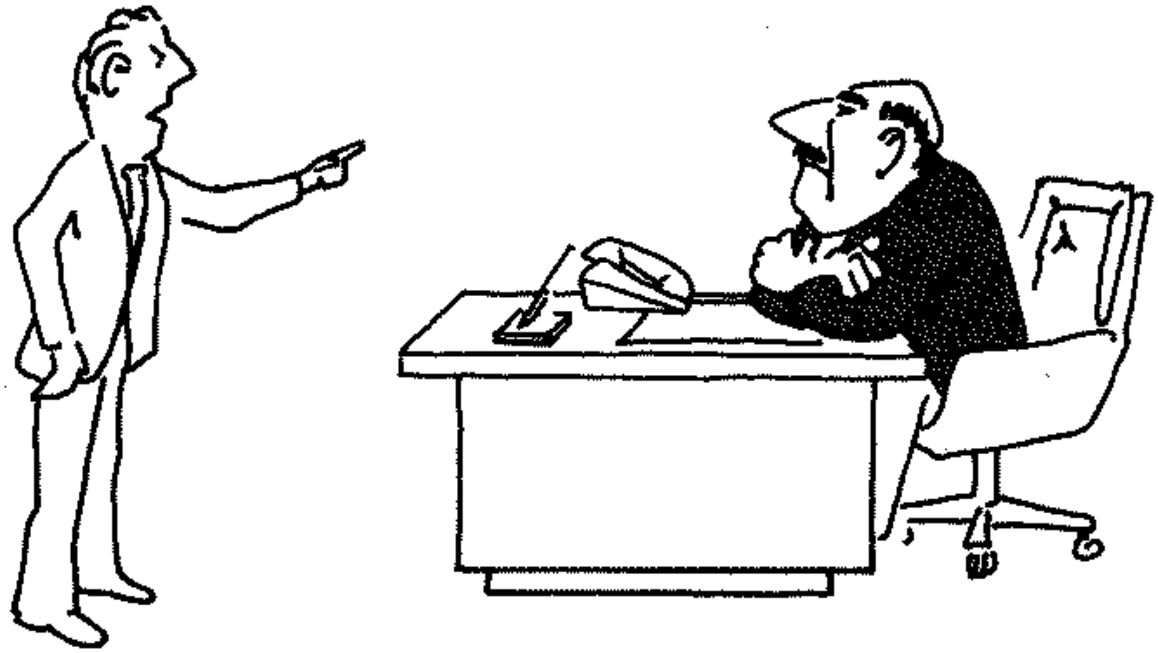


“I can’t find an efficient algorithm, I guess I’m just too dumb.”

Poly-Time Reductions: Implications

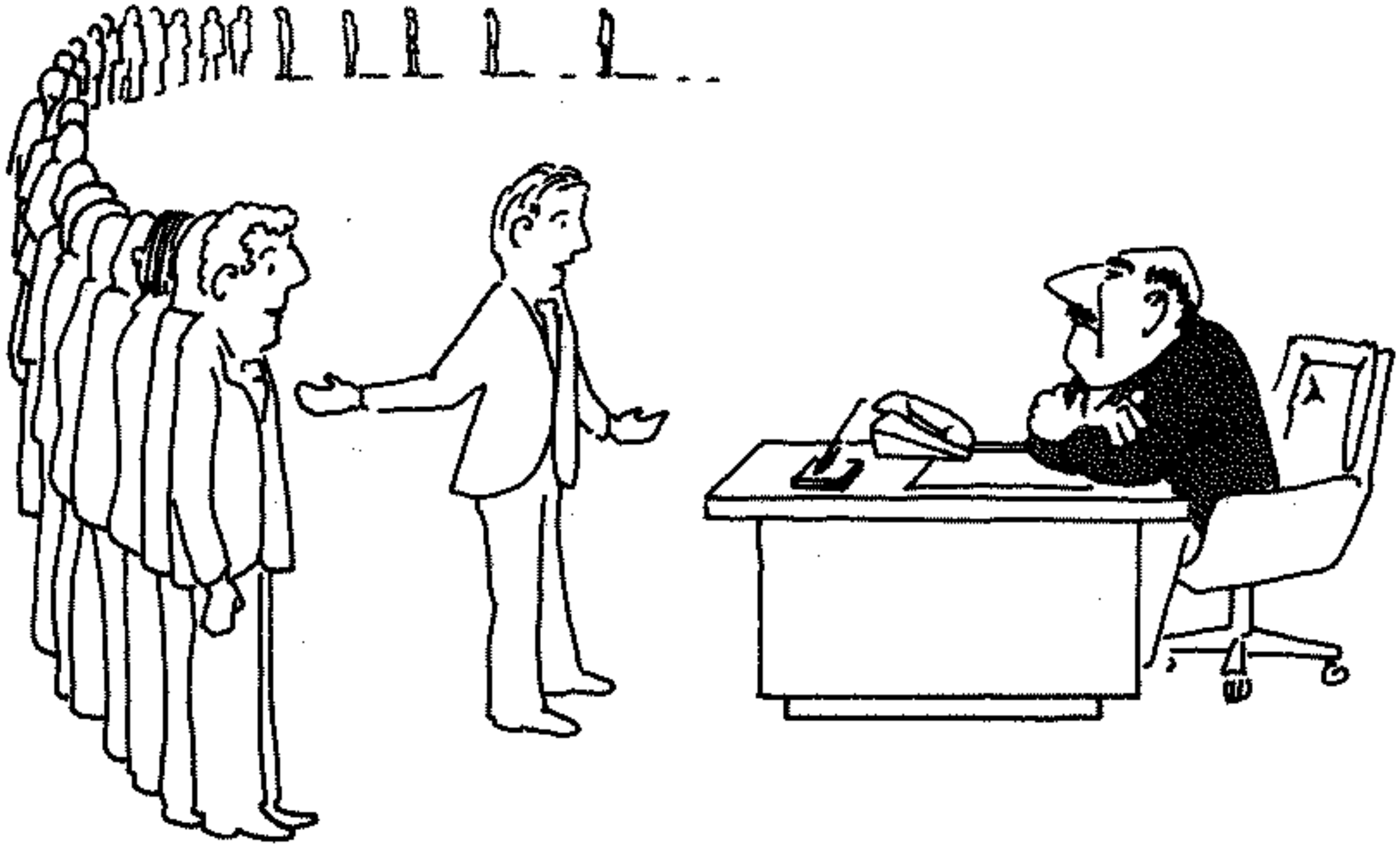


COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Michael R. Garey / David S. Johnson

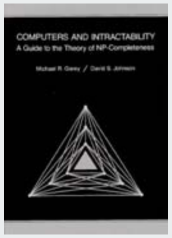


"I can't find an efficient algorithm, because no such algorithm is possible!"

Poly-Time Reductions: Implications



"I can't find an efficient algorithm, but neither can all these famous people."


COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Michael R. Garey / David S. Johnson

Summary

Reductions are important in theory to:

- Establish tractability.
- Establish intractability.
- Classify problems according to their computational requirements.

Reductions are important in practice to:

- Design algorithms.
- Design reusable software modules.
  stack, queue, sorting, priority queue, symbol table, set, graph
  shortest path, regular expressions, linear programming
- Determine difficulty of your problem and choose the right tool.
  use exact algorithm for tractable problems
  use heuristics for intractable problems

Combinatorial Search

- permutations
- backtracking
- counting
- subsets
- paths in a graph

Overview

Exhaustive search. Iterate through all elements of a search space.

Backtracking. Systematic method for examining feasible solutions to a problem, by systematically eliminating infeasible solutions.

Applicability. Huge range of problems (include NP-hard ones).

Caveat. Search space is typically exponential in size ⇒ effectiveness may be limited to relatively small instances.

Caveat to the caveat. Backtracking may prune search space to reasonable size, even for relatively large instances

Warmup: enumerate N-bit strings

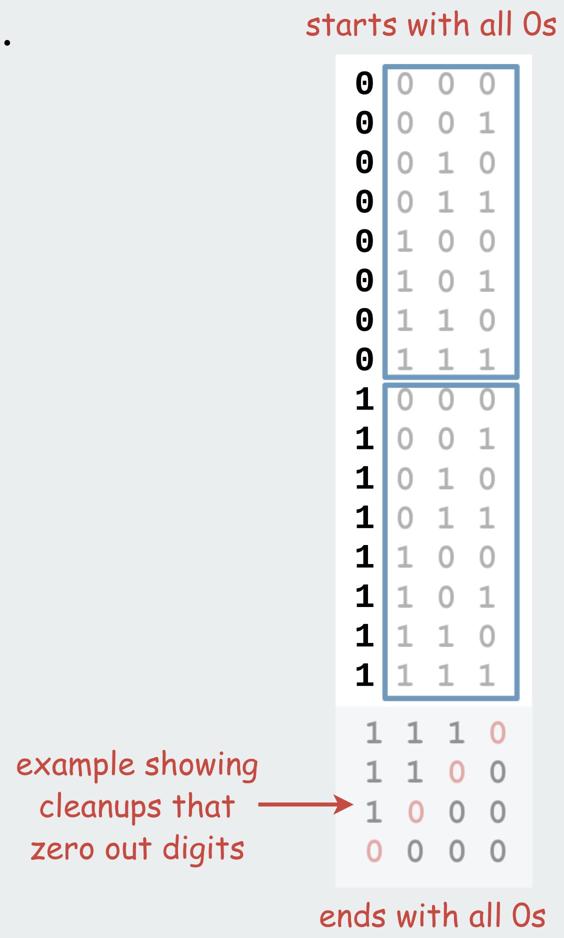
Problem: process all 2^N N-bit strings (stay tuned for applications).

Equivalent to counting in binary from 0 to 2^N - 1.

- maintain `a[i]` where `a[i]` represents bit `i`
- initialize all bits to `0`
- simple recursive method does the job (call `enumerate(0)`)

```
private void enumerate(int k)
{
  if (k == N)
  {  process(); return;  }
  enumerate(k+1);
  a[k] = 1;
  enumerate(k+1);
  a[k] = 0;
}
```

clean up



starts with all 0s

```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
 1 1 1 0
 1 1 0 0
 1 0 0 0
 0 0 0 0
```

example showing cleanups that zero out digits

ends with all 0s

Invariant (prove by induction);

Enumerates all (N-k)-bit strings and cleans up after itself.

Warmup: enumerate N-bit strings (full implementation)

Equivalent to counting in binary from 0 to $2^N - 1$.

```
public class Counter
{
   private int N;   // number of bits
   private int[] a; // bits (0 or 1)

   public Counter(int N)
   {
      this.N = N;
      a = new int[N];
      for (int i = 0; i < N; i++)
          a[i] = 0;
      enumerate(0);
   }

   private void enumerate(int k)
   {
     if (k == N)
     {  process(); return;  }
     enumerate(k+1);
     a[k] = 1;
     enumerate(k+1);
     a[k] = 0;
   }

   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Counter c = new Counter(N);
   }
}
```

optional (in this case)

all the programs in this lecture are variations on this theme

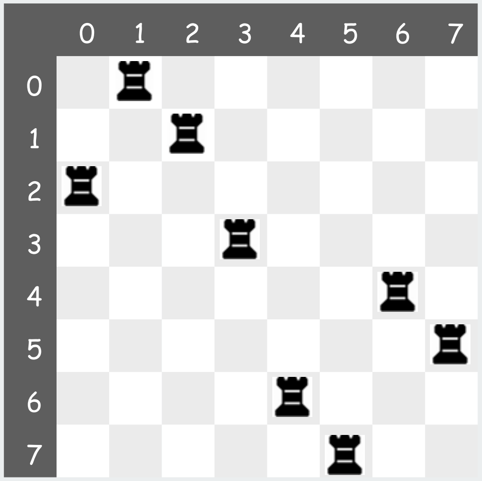
```
private void process()
{
   for (int i = 0; i < N; i++)
      StdOut.print(a[i]);
   StdOut.println();
}
```

```
% java Counter 4
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

- **permutations**
- backtracking
- counting
- subsets
- paths in a graph

N-rooks Problem

How many ways are there to place
N rooks on an N-by-N board so that no rook can attack any other?



original problem: N = 8

```
int[] a = { 1, 2, 0, 3, 6, 7, 4, 5 };
```

No two in the same row, so represent solution with an array

`a[i]` = column of rook in row `i`.

No two in the same column, so array entries are all different

`a[]` is a permutation (rearrangement of 0, 1, ... N-1)

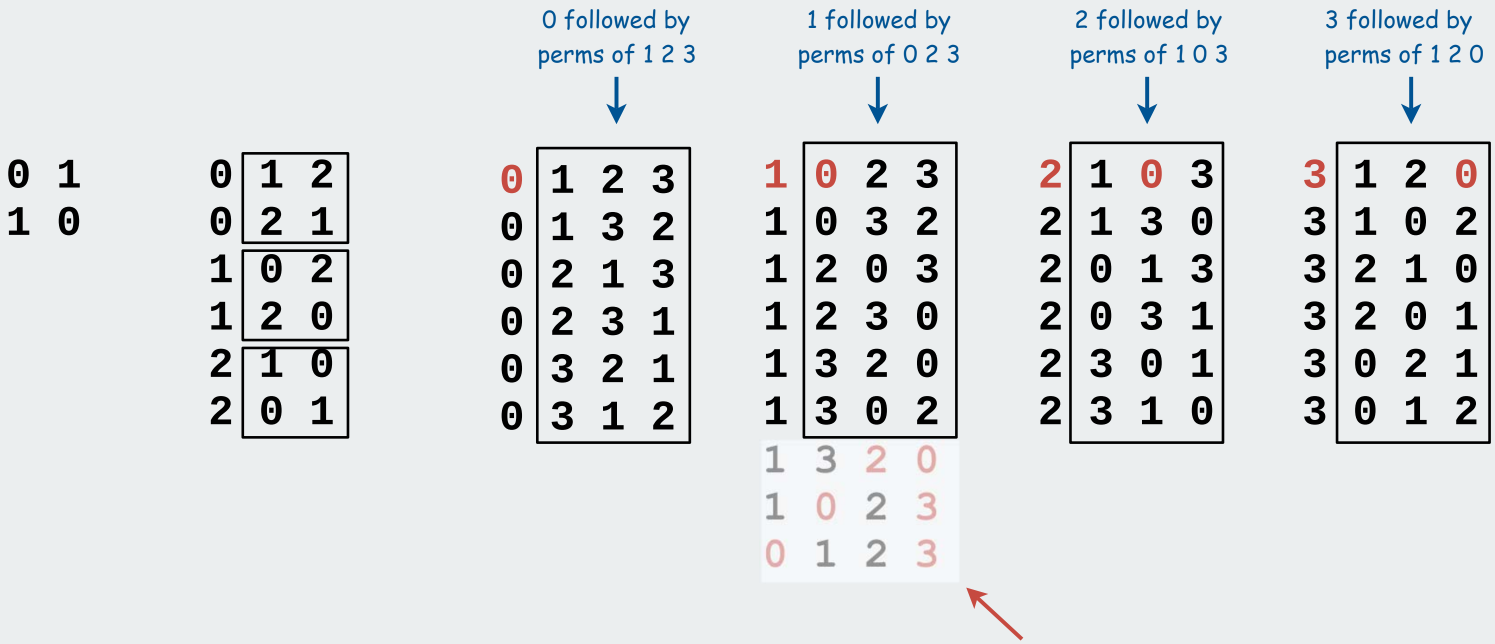
Answer: There are N! non mutually-attacking placements.

Challenge: Enumerate them all.

Enumerating permutations

Recursive algorithm to enumerate all N! permutations of size N:

- Start with `0 1 2 ... N-1`.
- For each value of `i`
  - swap `i` into position `0`
  - enumerate all (N-1)! arrangements of `a[1..N-1]`
  - clean up (swap `i` and `0` back into position)



0 followed by perms of 1 2 3

1 followed by perms of 0 2 3

2 followed by perms of 1 0 3

3 followed by perms of 1 2 0

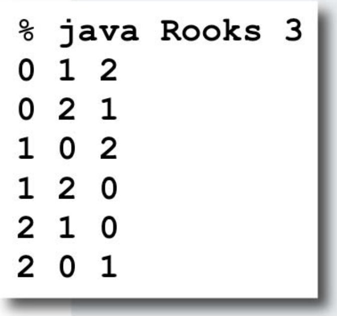
```
0 1
1 0

0 1 2
0 2 1
1 0 2
1 2 0
2 1 0
2 0 1

0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 2 1
0 3 1 2

1 0 2 3
1 0 3 2
1 2 0 3
1 2 3 0
1 3 2 0
1 3 0 2
1 3 2 0
1 0 2 3
0 1 2 3

2 1 0 3
2 1 3 0
2 0 1 3
2 0 3 1
2 3 0 1
2 3 1 0

3 1 2 0
3 1 0 2
3 2 1 0
3 2 0 1
3 0 2 1
3 0 1 2
```

example showing cleanup swaps
that bring perm back to original

N-rooks problem (enumerating all permutations): *scaffolding*

```
public class Rooks
{
   private int N;
   private int[] a;

   public Rooks(int N)
   {
      this.N = N;
      a = new int[N];
      for (int i = 0; i < N; i++)
         a[i] = i;
      enumerate(0);
   }

   private void enumerate(int k)
   { /* See next slide. */  }

   private void exch(int i, int j)
   {  int t = a[i]; a[i] = a[j]; a[j] = t;  }

   private void process()
   {
      for (int i = 0; i < N; i++)
          StdOut.print(a[i] + " ");
      StdOut.println();
   }

   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Rooks t = new Rooks(N);
      t.enumerate(0);
   }
}
```

initialize **`a[0..N-1]`** to **`0..N-1`**



```
% java Rooks 3
0 1 2
0 2 1
1 0 2
1 2 0
2 1 0
2 0 1
```

N-rooks problem (enumerating all permutations): recursive enumeration

Recursive algorithm to enumerate all N! permutations of size N:

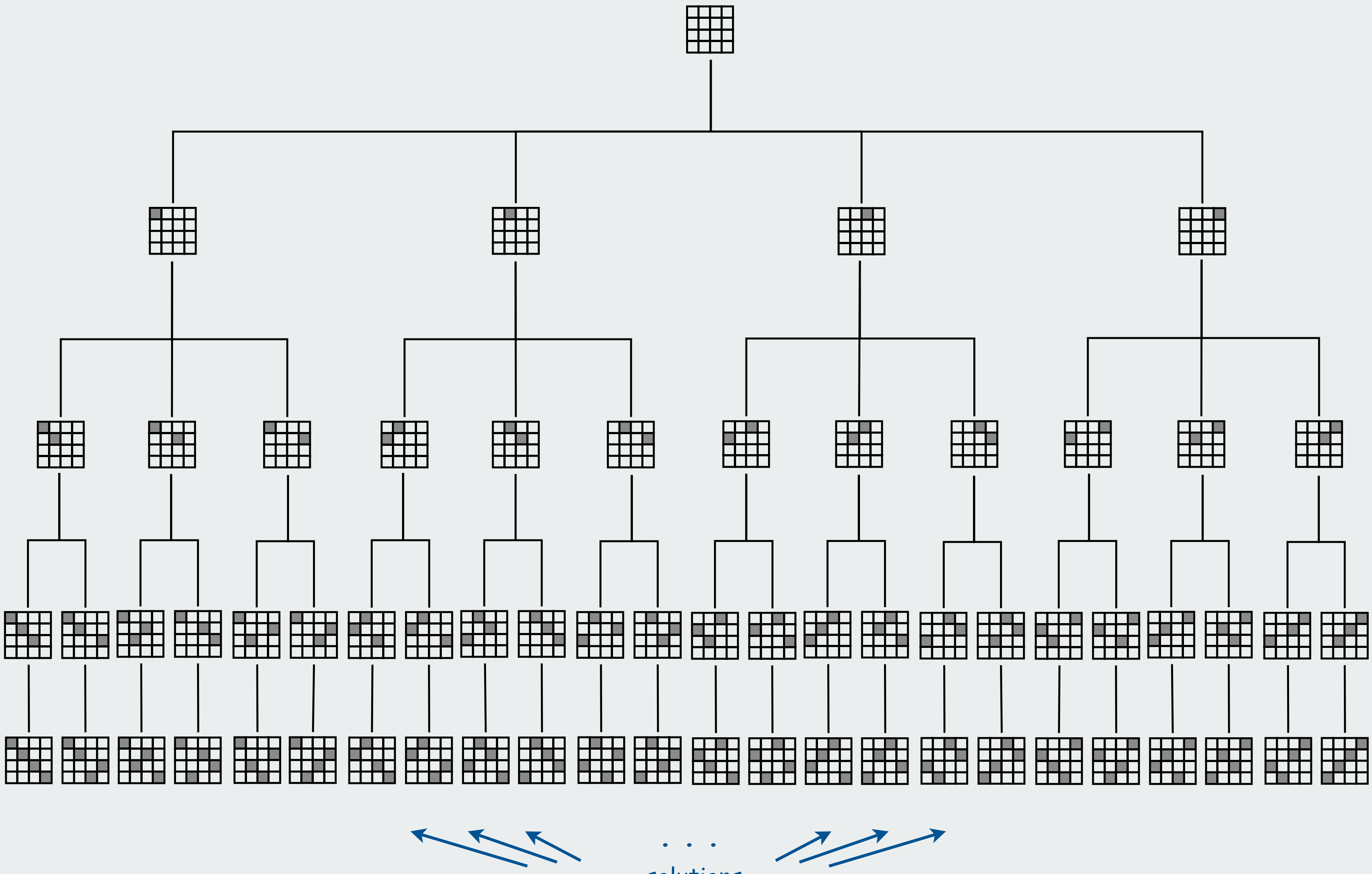
- Start with `0 1 2 ... N-1.`
- For each value of `i`
  - swap `i` into position `0`
  - enumerate all (N-1)! arrangements of `a[1..N-1]`
  - clean up (swap `i` and `0` back into position)

```
private void enumerate(int k)
{
   if (k == N)
   {
      process();
      return;
   }
   for (int i = k; i < N; i++)
   {
      exch(a, k, i);
      enumerate(k+1);
      exch(a, k, i);
   }
}
```

clean up

```
% java Rooks 4
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 2 1
0 3 1 2
1 0 2 3
1 0 3 2
1 2 0 3
1 2 3 0
1 3 2 0
1 3 0 2
2 1 0 3
2 1 3 0
2 0 1 3
2 0 3 1
2 3 0 1
2 3 1 0
3 1 2 0
3 1 0 2
3 2 1 0
3 2 0 1
3 0 2 1
3 0 1 2
```

4-Rooks search tree



. . .
solutions

N-rooks problem: back-of-envelope running time estimate

[Studying slow way to compute N! but good warmup for calculations.]

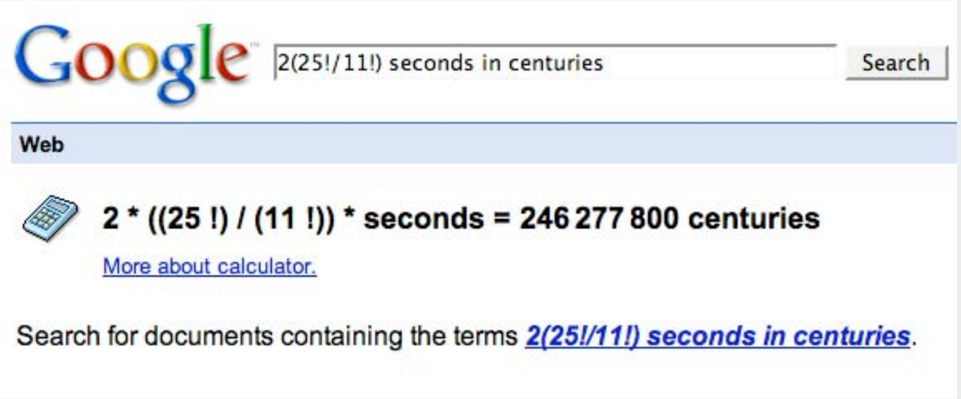
```
% java Rooks 10
3628800 solutions

% java Rooks 11
39916800 solutions

% java Rooks 12
479001600 solutions
```

← instant

← about 2 seconds

← about 24 seconds (checks with N! hypothesis)

Hypothesis: Running time is about 2(N! / 11!) seconds.



2(25!/11!) seconds in centuries

2 * ((25 !) / (11 !)) * seconds = 246 277 800 centuries

More about calculator.

Search for documents containing the terms 2(25!/11!) seconds in centuries.

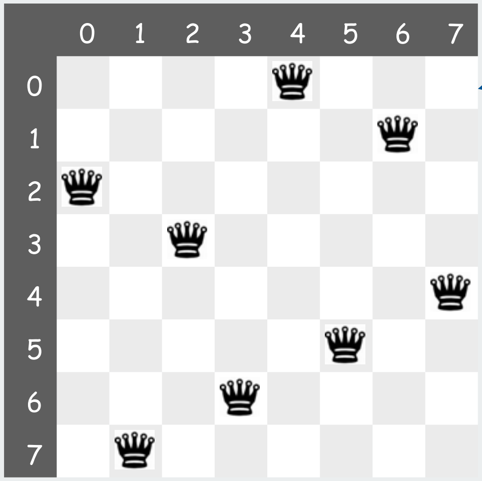
```
% java Rooks 25
```

← millions of centuries

- permutations
- backtracking
- counting
- subsets
- paths in a graph

N-Queens problem

How many ways are there to place
N queens on an N-by-N board so that no queen can attack any other?

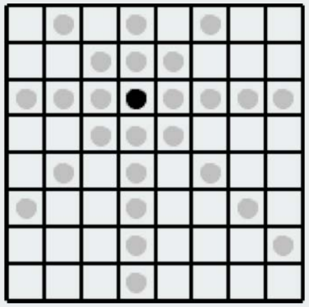


original problem: N = 8

```
int[] a = { 4, 6, 0, 2, 7, 5, 3, 1 };
```

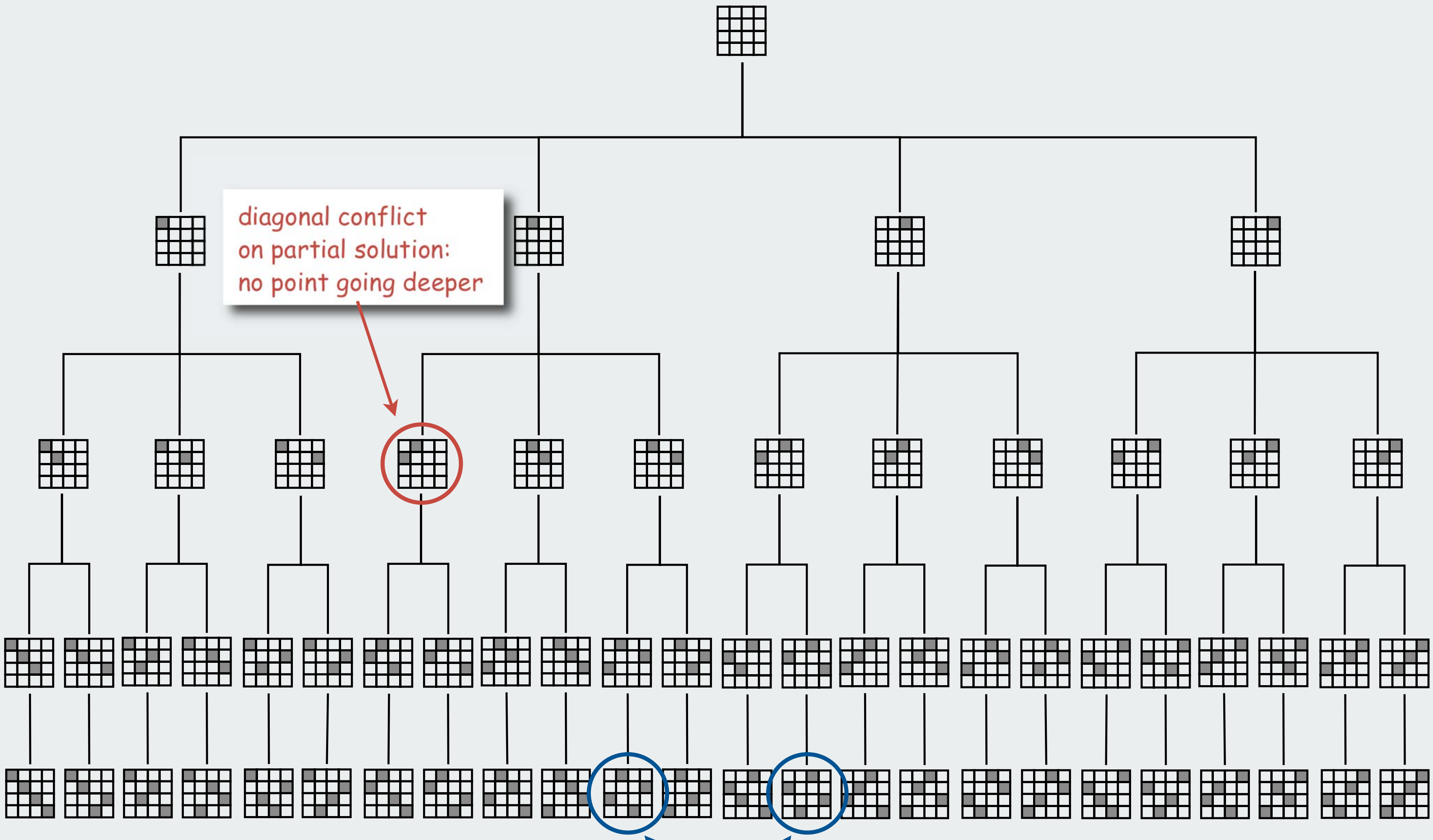
Representation. Same as for rooks:
represent solution as a permutation: `a[i]` = column of queen in row `i`.

Additional constraint: no diagonal attack is possible



Challenge: Enumerate (or even count) the solutions

4-Queens search tree



diagonal conflict on partial solution: no point going deeper

solutions

N Queens: Backtracking solution

Iterate through elements of search space.

- when there are N possible choices, make one choice and recur.
- if the choice is a dead end, backtrack to previous choice, and make next available choice.

Identifying dead ends allows us to prune the search tree

For N queens:

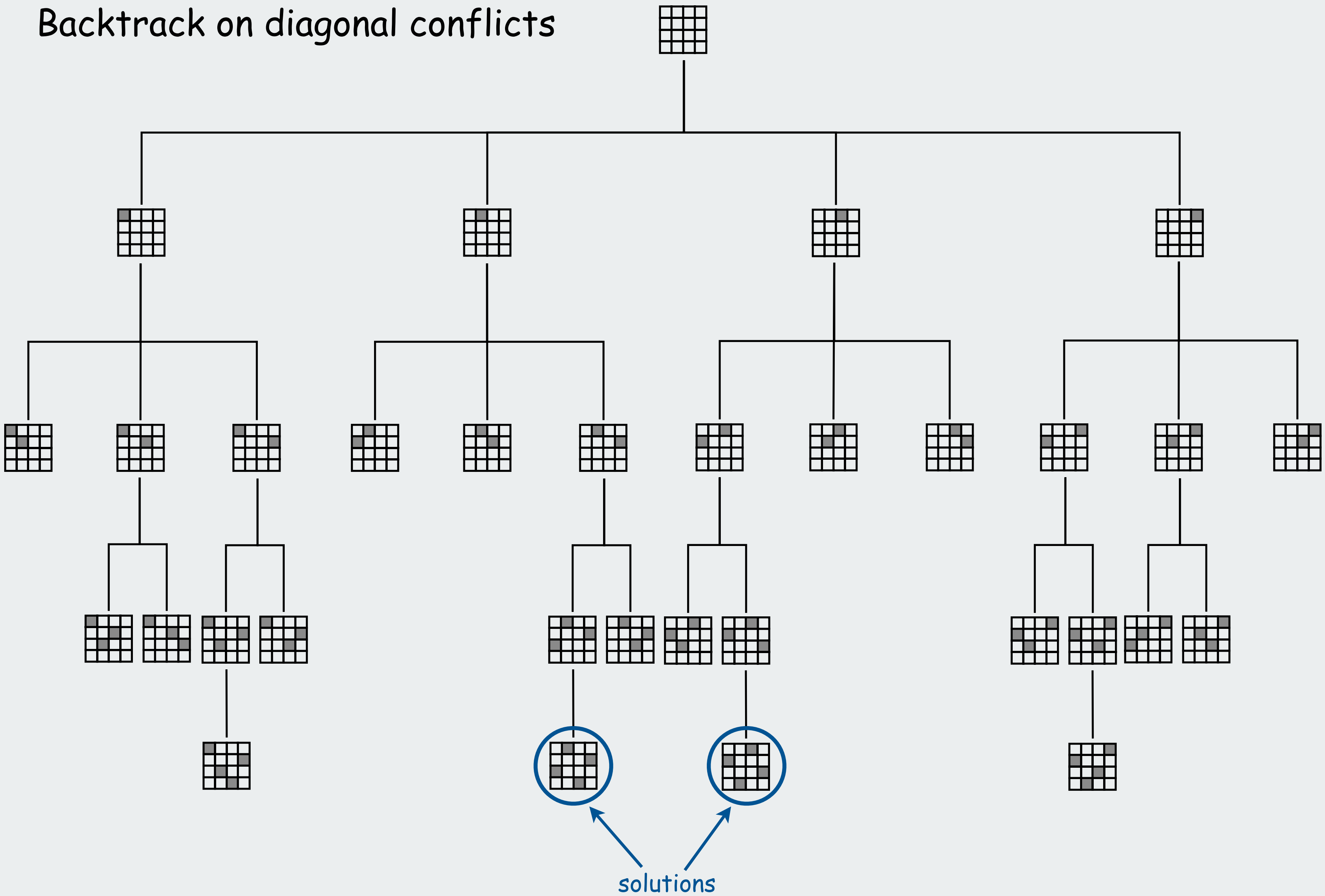
- dead end: a diagonal conflict
- pruning: backtrack and try next row when diagonal conflict found

In general, improvements are possible:

- try to make an "intelligent" choice
- try to reduce cost of choosing/backtracking

4-Queens Search Tree (pruned)

Backtrack on diagonal conflicts



N-Queens: Backtracking solution

```
private boolean backtrack(int k)
{
   for (int i = 0; i < k; i++)
   {
      if ((a[i] - a[k]) == (k - i)) return true;
      if ((a[k] - a[i]) == (k - i)) return true;
   }
   return false;
}

private void enumerate(int k)
{
   if (k == N)
   {
      process();
      return;
   }
   for (int i = k; i < N; i++)
   {
      exch(a, k, i);
      if (! backtrack(k)) enumerate(k+1);
      exch(a, k, i);
   }
}
```

stop enumerating if adding the nth queen leads to a diagonal violation

```
% java Queens 4
1 3 0 2
2 0 3 1

% java Queens 5
0 2 4 1 3
0 3 1 4 2
1 3 0 2 4
1 4 2 0 3
2 0 3 1 4
2 4 1 3 0
3 1 4 2 0
3 0 2 4 1
4 1 3 0 2
4 2 0 3 1

% java Queens 6
1 3 5 0 2 4
2 5 1 4 0 3
3 0 4 1 5 2
4 2 0 5 3 1
```

N-Queens: Effectiveness of backtracking

Pruning the search tree leads to enormous time savings

N	2	3	4	5	6	7	8	9	10	11	12
Q(N)	0	0	2	10	4	40	92	352	724	2,680	14,200
N!	2	6	24	120	720	5,040	40,320	362,880	3,628,800	39,916,800	479,001,600

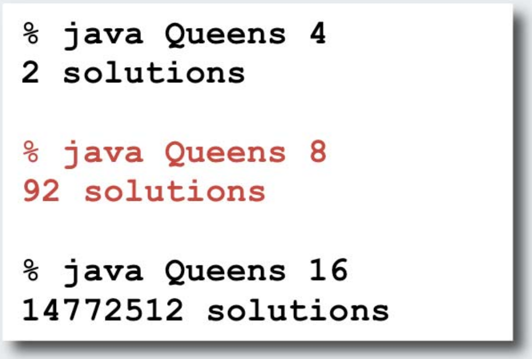
N	13	14	15	16
Q(N)	73,712	365,596	2,279,184	14,772,512
N!	6,227,020,800	87,178,291,200	1 ,307,674,368,000	20, 922,789,888,000

↑ savings: factor of more than 1-million

N-Queens: How many solutions?

Answer to original question easy to obtain:

- add an instance variable to count solutions (initialized to 0)
- change `process()` to increment the counter
- add a method to return its value



```
% java Queens 4
2 solutions

% java Queens 8
92 solutions

% java Queens 16
14772512 solutions
```

Source: On-line encyclopedia of integer sequences, N. J. Sloane [sequence A000170]

N	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Q(N)	0	0	2	10	4	40	92	352	724	2,680	14,200	73,712	365,596	2,279,184

N	16	17	18	19		25
Q(N)	14,772,512	95,815,104	666,090,624	4,968,057,848	. . .	2, 207,893,435,808,350

took 53 years of CPU time (2005)

N-queens problem: back-of-envelope running time estimate

Hypothesis ??

```
% java Queens 13
73712 solutions
```

about a second

```
% java Queens 14
365596 solutions
```

about 7 seconds

```
% java Queens 15
2279184 solutions
```

about 49 seconds

```
% java Queens 16
14772512 solutions
```

about 360 seconds

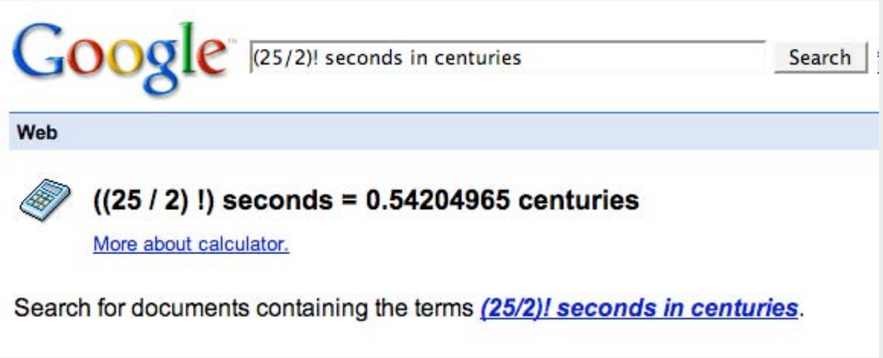
ratio ↓

6.32

6.73

7.38

Hypothesis: Running time is about (N/2) ! seconds.



(25/2)! seconds in centuries

((25 / 2) !) seconds = 0.54204965 centuries

More about calculator.

Search for documents containing the terms ***(25/2)! seconds in centuries***.

```
% java Queens 25
```

about 54 years ✓

- permutations
- backtracking
- **counting**
- subsets
- paths in a graph

Counting: Java Implementation

Problem: enumerate all N-digit base-R numbers

Solution: generalize binary counter in lecture warmup

enumerate N-digit base-R numbers

```
private static void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }

   for (int n = 0; n < R; n++)
   {
      a[k] = n;
      enumerate(k + 1);
   }
   a[k] = 0;
}
```

clean up not needed: Why?

enumerate binary numbers (from warmup)

```
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }
   enumerate(k+1);
   a[k] = 1;
   enumerate(k+1);
   a[k] = 0;
}
```

clean up

```
0 0 0    1 0 0    2 0 0
0 0 1    1 0 1    2 0 1
0 0 2    1 0 2    2 0 2
0 1 0    1 1 0    2 1 0
0 1 1    1 1 1    2 1 1
0 1 2    1 1 2    2 1 2
0 2 0    1 2 0    2 2 0
0 2 1    1 2 1    2 2 1
0 2 2    1 2 2    2 2 2
0 2 0
0 0 0
```

example showing cleanups that zero out digits

Counting application: Sudoku

Problem:

Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

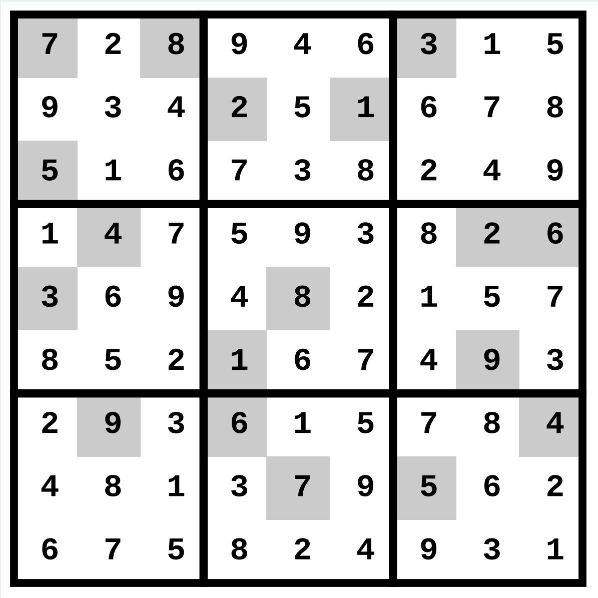
7		8				3		
			2		1			
5								
	4						2	6
3				8				
			1				9	
	9		6					4
				7		5		

Remark: Natural generalization is NP-hard.

Counting application: Sudoku

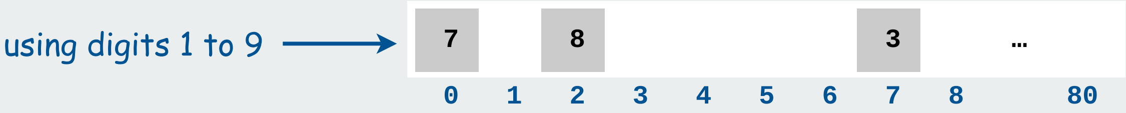
Problem:

Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.



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

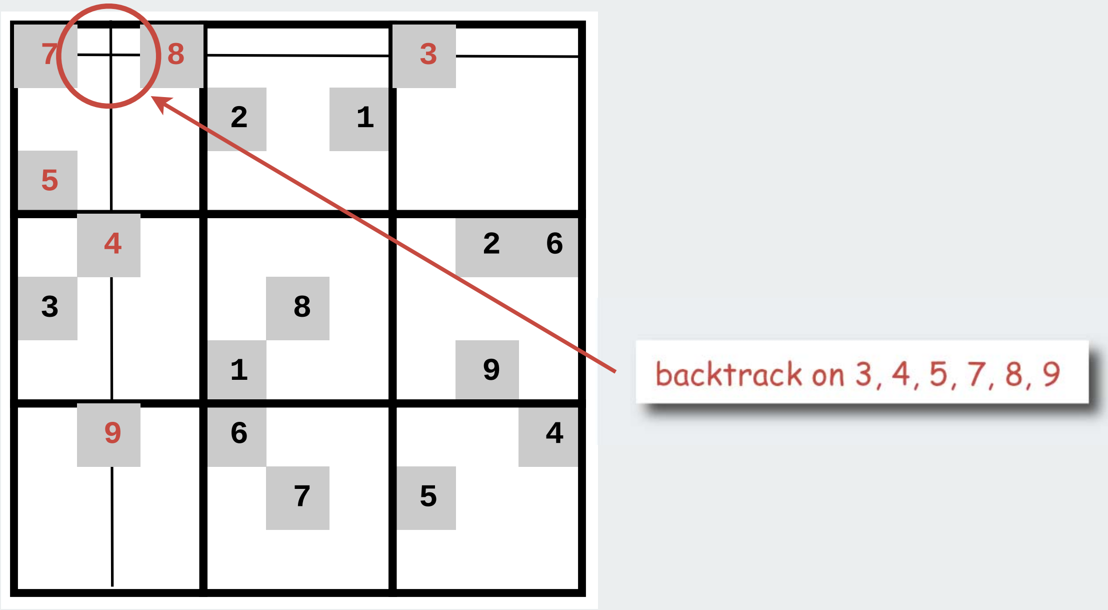
Solution: Enumerate all 81-digit base-9 numbers (with backtracking).



Sudoku: Backtracking solution

Iterate through elements of search space.

- For each empty cell, there are 9 possible choices.
- Make one choice and recur.
- If you find a conflict in row, column, or box, then backtrack.



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

backtrack on 3, 4, 5, 7, 8, 9

Improvements are possible.

- try to make an "intelligent" choice
- try to reduce cost of choosing/backtracking

Sudoku: Java implementation

```
private static void solve(int cell)
{
   if (cell == 81)
   {  show(board); return;  }

   if (board[cell] != 0)
   {  solve(cell + 1); return;  }

   for (int n = 1; n <= 9; n++)
   {
      if (! backtrack(cell, n))
      {
         board[cell] = n;
         solve(cell + 1);
      }
   }

   board[cell] = 0;
}
```

already filled in

try all 9 possibilities

unless a Sudoku constraint is violated (see booksite for code)

clean up

```
...
int[81] board;
for (int i = 0; i < 81; i++)
   board[i] = StdOut.readInt();
Solver s = new Solver(board);
s.solve();
...
```

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

% java Solver
7 2 8 9 4 6 3 1 5
9 3 4 2 5 1 6 7 8
5 1 6 7 3 8 2 4 9
1 4 7 5 9 3 8 2 6
3 6 9 4 8 2 1 5 7
8 5 2 1 6 7 4 9 3
2 9 3 6 1 5 7 8 4
4 8 1 3 7 9 5 6 2
6 7 5 8 2 4 9 3 1
```

Works remarkably well (plenty of constraints). Try it!

- permutations
- backtracking
- counting
- **subsets**
- paths in a graph

Enumerating subsets: natural binary encoding

Given n items, enumerate all 2^n subsets.

- count in binary from 0 to 2^n - 1.
- bit i represents item i
- if 0, in subset; if 1, not in subset

i	binary	subset	complement
0	0 0 0 0	empty	4 3 2 1
1	0 0 0 1	1	4 3 2
2	0 0 1 0	2	4 3 1
3	0 0 1 1	2 1	4 3
4	0 1 0 0	3	4 2 1
5	0 1 0 1	3 1	4 2
6	0 1 1 0	3 2	4 1
7	0 1 1 1	3 2 1	4
8	1 0 0 0	4	3 2 1
9	1 0 0 1	4 1	3 2
10	1 0 1 0	4 2	3 1
11	1 0 1 1	4 2 1	3
12	1 1 0 0	4 3	2 1
13	1 1 0 1	4 3 1	2
14	1 1 1 0	4 3 2	1
15	1 1 1 1	4 3 2 1	empty

Enumerating subsets: natural binary encoding

Given N items, enumerate all 2^N subsets.

- count in binary from 0 to 2^N - 1.
- maintain `a[i]` where `a[i]` represents item `i`
- if `0`, `a[i]` in subset; if `1`, `a[i]` not in subset

Binary counter from warmup does the job

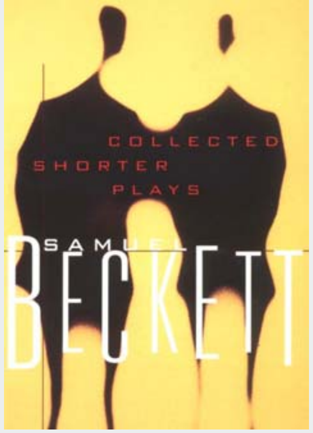
```
private void enumerate(int k)
{
  if (k == N)
  {  process(); return;  }
  enumerate(k+1);
  a[k] = 1;
  enumerate(k+1);
  a[k] = 0;
}
```

Digression: Samuel Beckett play

Quad. Starting with empty stage, 4 characters enter and exit one at a time, such that each subset of actors appears exactly once.

code	*subset*	*move*
0 0 0 0	*empty*	
0 0 0 1	1	enter 1
0 0 1 1	2 1	enter 2
0 0 1 0	2	exit 1
0 1 1 0	3 2	enter 3
0 1 1 1	3 2 1	enter 1
0 1 0 1	3 1	exit 2
0 1 0 0	3	exit 1
1 1 0 0	4 3	enter 4
1 1 0 1	4 3 1	enter 1
1 1 1 1	4 3 2 1	enter 2
1 1 1 0	4 3 2	exit 1
1 0 1 0	4 2	exit 3
1 0 1 1	4 2 1	enter 1
1 0 0 1	4 1	exit 2
1 0 0 0	4	exit 1

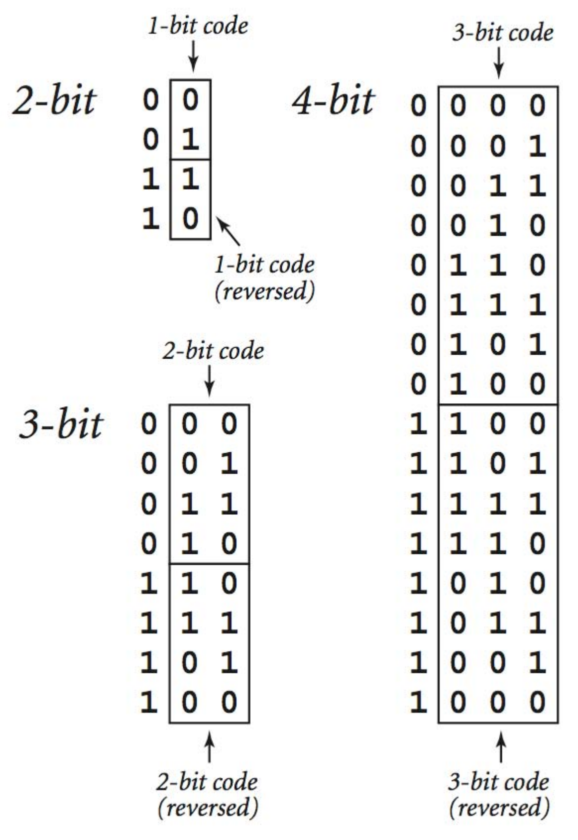
ruler function



Binary reflected gray code

The n-bit binary reflected Gray code is:

- the (n-1) bit code with a 0 prepended to each word, followed by
- the (n-1) bit code in reverse order, with a 1 prepended to each word.



2-bit

```
0 0
0 1
1 1
1 0
```

3-bit

```
0 0 0
0 0 1
0 1 1
0 1 0
1 1 0
1 1 1
1 0 1
1 0 0
```

4-bit

```
0 0 0 0
0 0 0 1
0 0 1 1
0 0 1 0
0 1 1 0
0 1 1 1
0 1 0 1
0 1 0 0
1 1 0 0
1 1 0 1
1 1 1 1
1 1 1 0
1 0 1 0
1 0 1 1
1 0 0 1
1 0 0 0
```

Beckett: Java implementation

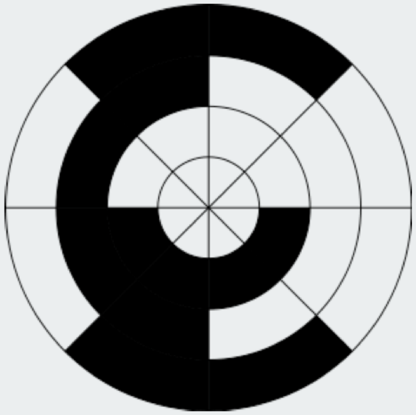
```
public static void moves(int n, boolean enter)
{
   if (n == 0) return;
   moves(n-1, true);
   if (enter) System.out.println("enter " + n);
   else       System.out.println("exit  " + n);
   moves(n-1, false);
}
```

```
% java Beckett 4
enter 1
enter 2
exit  1
enter 3
enter 1
exit  2
exit  1
enter 4
enter 1
enter 2
exit  1
exit  3
enter 1
exit  2
exit  1
```

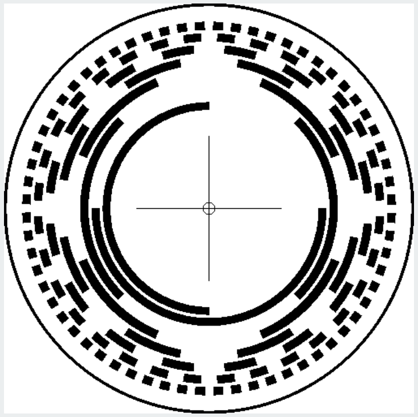
stage directions
for 3-actor play
`moves(3, true)`

reverse stage directions
for 3-actor play
`moves(3, false)`

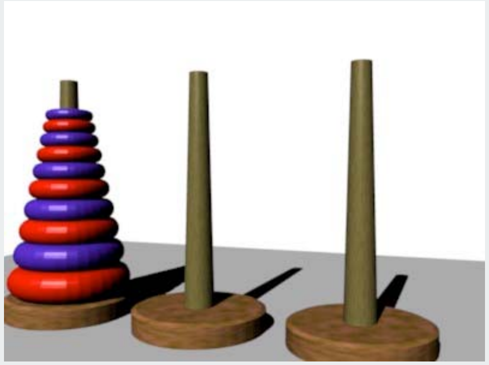
More Applications of Gray Codes



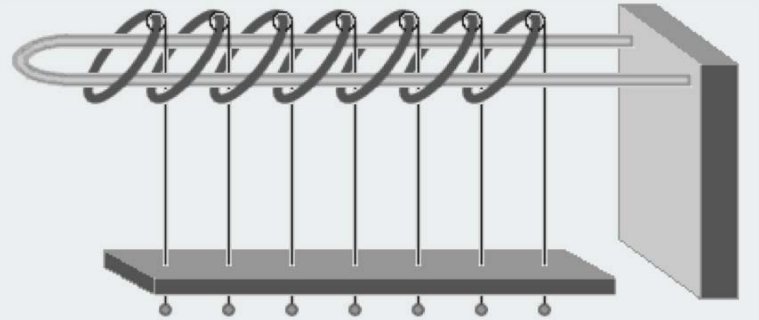
3-bit rotary encoder



8-bit rotary encoder



Towers of Hanoi



Chinese ring puzzle

Enumerating subsets using Gray code

Two simple changes to binary counter from warmup:

- flip `a[k]` instead of setting it to `1`
- eliminate cleanup

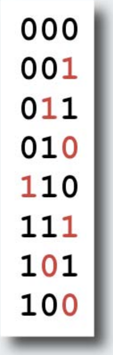
Gray code enumeration

```
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }
   enumerate(k+1);
   a[k] = 1 - a[k];
   enumerate(k+1);
}
```

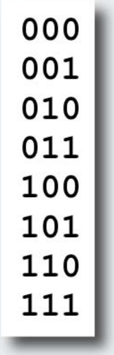
standard binary (from warmup)

```
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }
   enumerate(k+1);
   a[k] = 1;
   enumerate(k+1);
   a[k] = 0;
}
```

clean up



```
000
001
011
010
110
111
101
100
```



```
000
001
010
011
100
101
110
111
```

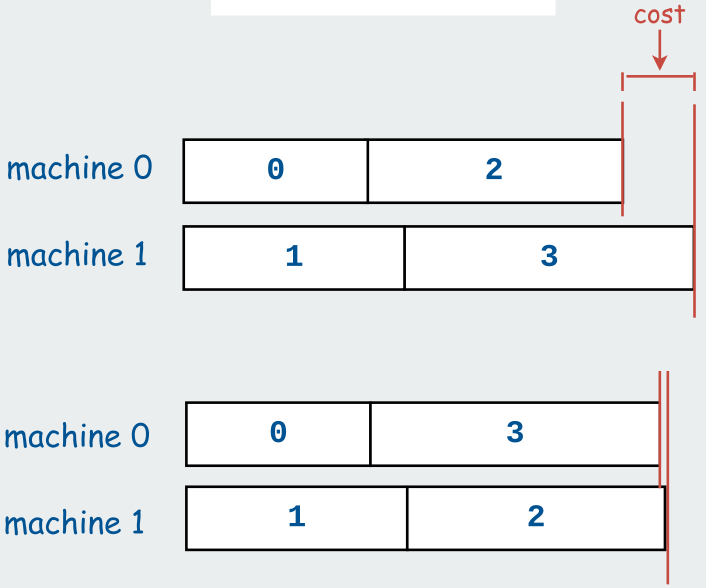
Advantage (same as Beckett): only one item changes subsets

Scheduling

Scheduling (set partitioning). Given n jobs of varying length, divide among two machines to minimize the time the last job finishes.

or, equivalently, difference between finish times

job	length
0	1.41
1	1.73
2	2.00
3	2.23



```
public double[] finish(int[] a)
{
    double[] time = new double[2];
    time[0] = 0.0; time[1] = 0.0;
    for (int i = 0; i < N; i++)
        time[a[i]] += jobs[i];
    return time;
}

private double cost(int[] a)
{
    double[] time = finish(a);
    return Math.abs(time[0] - time[1]);
}
```

```
i        a[]          time[0]  time[1]

       0 1 1 0         1.41       0
0      0 1 1 0         1.41       0
1      0 1 1 0         1.41     1.73
2      0 1 1 0         1.41     3.73
3      0 1 1 0         3.64     3.73
                       3.64     3.73

                   cost: .09
```

Remark: NP-hard.

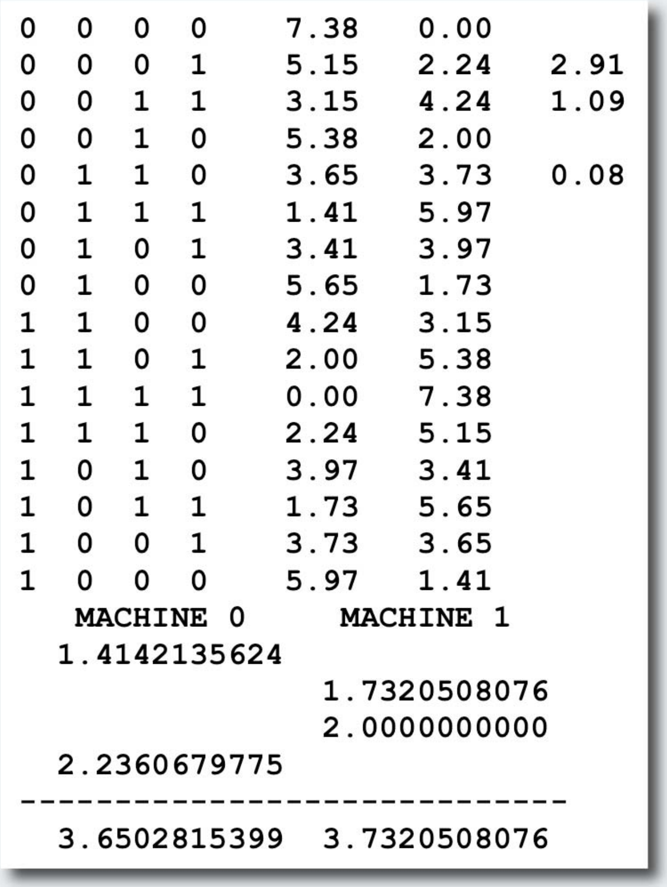
Scheduling (full implementation)

```
public class Scheduler
{
   int N;           // Number of jobs.
   int[] a;         // Subset assignments.
   int[] b;         // Best assignment.
   double[] jobs;   // Job lengths.

   public Scheduler(double[] jobs)
   {
      this.N = jobs.length;;
      this.jobs = jobs;
      a = new int[N];
      b = new int[N];
      for (int i = 0; i < N; i++)
          a[i] = 0;
      for (int i = 0; i < N; i++)
          b[i] = a[i];
      enumerate(0);
   }

   public int[] best()
   {  return b;  }


   private void enumerate(int k)
   { /* Gray code enumeration. */   }

   private void process()
   {
     if (cost(a) < cost(b))
       for (int i = 0; i < N; i++)
         b[i] = a[i];
   }

   public static void main(String[] args)
   { /* Create Scheduler, print result. */  }
}
```

trace of

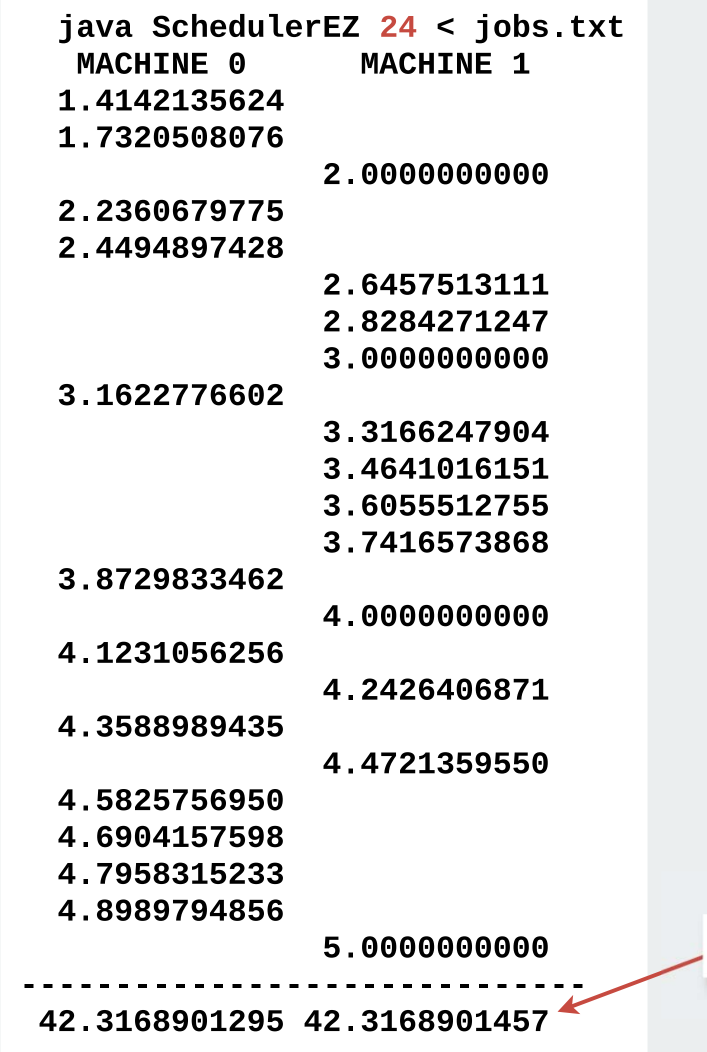
`% java Scheduler 4 < jobs.txt`

a[]   finish times   cost



```
0  0  0  0    7.38   0.00
0  0  0  1    5.15   2.24   2.91
0  0  1  1    3.15   4.24   1.09
0  0  1  0    5.38   2.00
0  1  1  0    3.65   3.73   0.08
0  1  1  1    1.41   5.97
0  1  0  1    3.41   3.97
0  1  0  0    5.65   1.73
1  1  0  0    4.24   3.15
1  1  0  1    2.00   5.38
1  1  1  1    0.00   7.38
1  1  1  0    2.24   5.15
1  0  1  0    3.97   3.41
1  0  1  1    1.73   5.65
1  0  0  1    3.73   3.65
1  0  0  0    5.97   1.41
   MACHINE 0      MACHINE 1
  1.4142135624
                 1.7320508076
                 2.0000000000
  2.2360679775
-----------------------------
  3.6502815399   3.7320508076
```

Scheduling (larger example)



```
  java SchedulerEZ 24 < jobs.txt
   MACHINE 0       MACHINE 1
  1.4142135624
  1.7320508076
                 2.0000000000
  2.2360679775
  2.4494897428
                 2.6457513111
                 2.8284271247
                 3.0000000000
  3.1622776602
                 3.3166247904
                 3.4641016151
                 3.6055512755
                 3.7416573868
  3.8729833462
                 4.0000000000
  4.1231056256
                 4.2426406871
  4.3588989435
                 4.4721359550
  4.5825756950
  4.6904157598
  4.7958315233
  4.8989794856
                 5.0000000000
------------------------------
 42.3168901295 42.3168901457
```

cost < 10^{-8}

Large number of subsets leads to remarkably low cost

Scheduling: improvements

Many opportunities (details omitted)

- fix last job on machine 0 (quick factor-of-two improvement)
- backtrack when partial schedule cannot beat best known (check total against goal: half of total job times)

```
private void enumerate(int k)
{
  if (k == N-1)
  {  process(); return;  }
  if (backtrack(k)) return;
  enumerate(k+1);
  a[k] = 1 - a[k];
  enumerate(k+1);
}
```

- process all 2^k subsets of last k jobs, keep results in memory, (reduces time to 2^{N-k} when 2^k memory available).

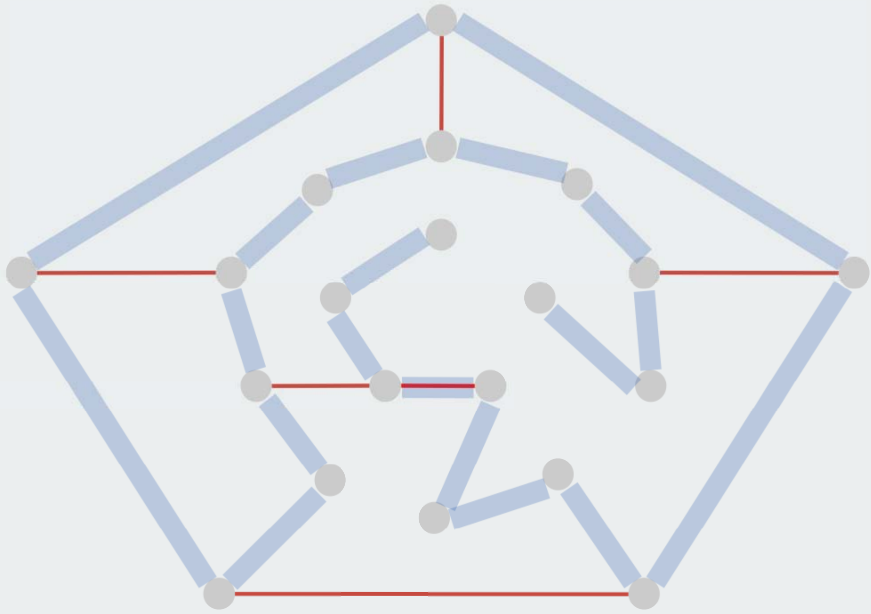
Backtracking summary

N-Queens : permutations with backtracking
Soduko : counting with backtracking
Scheduling: subsets with backtracking

- permutations
- backtracking
- counting
- subsets
- **paths in a graph**

Hamilton Path

Hamilton path. Find a simple path that visits every vertex exactly once.

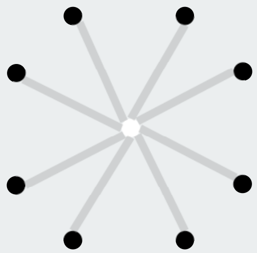


Remark. Euler path easy, but Hamilton path is NP-complete.

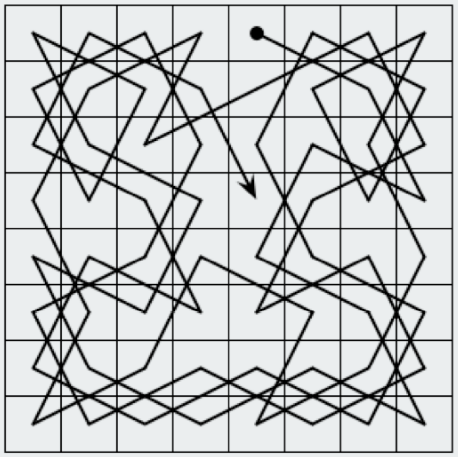
visit every edge
exactly once

Knight's Tour

Knight's tour. Find a sequence of moves for a knight so that, starting from any square, it visits every square on a chessboard exactly once.



legal knight moves



a knight's tour

Solution. Find a Hamilton path in knight's graph.

Hamilton Path: Backtracking Solution

Backtracking solution. To find Hamilton path starting at v:

- Add v to current path.
- For each vertex w adjacent to v
  find a simple path starting at w using all remaining vertices
- Remove v from current path.

How to implement?
Add cleanup to DFS (!!)

Hamilton Path: Java implementation

```
public class HamiltonPath
{
   private boolean[] marked;
   private int count;

   public HamiltonPath(Graph G)
   {
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         dfs(G, v, 1);
      count = 0;
   }

   private void dfs(Graph G, int v, int depth)
   {
      marked[v] = true;

      if (depth == G.V()) count++;

      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w, depth+1);

      marked[v] = false;
   }
}
```

also need code to count solutions (path length = V)

clean up

Easy exercise: Modify this code to find and print the longest path

The Longest Path

Recorded by Dan Barrett in 1988 while a student at Johns Hopkins during a difficult algorithms final.

Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path!

If you said P is NP tonight,
There would still be papers left to write,
I have a weakness,
I'm addicted to completeness,
And I keep searching for the longest path.

The algorithm I would like to see
Is of polynomial degree,
But it's elusive:
Nobody has found conclusive
Evidence that we can find a longest path.

I have been hard working for so long.
I swear it's right, and he marks it wrong.
Some how I'll feel sorry when it's done: GPA 2.1
Is more than I hope for.

Garey, Johnson, Karp and other men (and women)
Tried to make it order N log N.
Am I a mad fool
If I spend my life in grad school,
Forever following the longest path?

Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path.