Quicksort

- quicksort
- selection
- duplicate keys
- system sorts

Reference: `http://www.cs.princeton.edu/algs4`

Algorithms in Java, 4th Edition · Robert Sedgewick and Kevin Wayne · Copyright © 2008 · September 26, 2008 11:14:59 AM

Two 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.

Mergesort.

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

Quicksort.

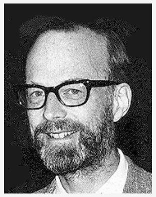
- Java sort for primitive types.
- C qsort, Unix, g++, Visual C++, Python.

- quicksort
- selection
- duplicate keys
- system sorts

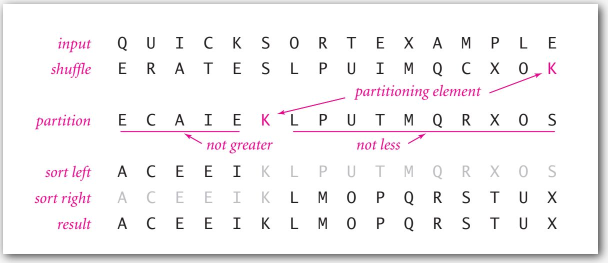
Quicksort

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
shuffle     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   A C E E I K L P U T M Q R X O S
sort right  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 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.

	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	-1	15	E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
scan left, scan right	1	12	E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
exchange	1	12	E	C	A	T	E	S	L	P	U	I	M	Q	R	X	O	K
scan left, scan right	3	9	E	C	A	T	E	S	L	P	U	I	M	Q	R	X	O	K
exchange	3	9	E	C	A	I	E	S	L	P	U	T	M	Q	R	X	O	K
scan left, scan right	5	5	E	C	A	I	E	S	L	P	U	T	M	Q	R	X	O	K
final exchange	5	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result			E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

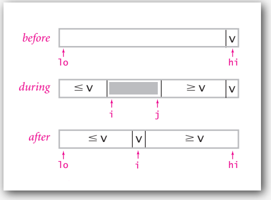
a[i]

v

Partitioning trace (array contents before and after each exchange)

Quicksort: Java code for partitioning

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

      while (less(a[++i], a[hi]))                  find item on left to swap
         if (i == hi) break;

      while (less(a[hi], a[--j]))                  find item on right to swap
         if (j == lo) break;


      if (i >= j) break;                               check if pointers cross

      exch(a, i, j);                                                      swap
   }

   exch(a, i, hi);                                swap with partitioning item
   return i;                      return index of item now known to be in place
}
```



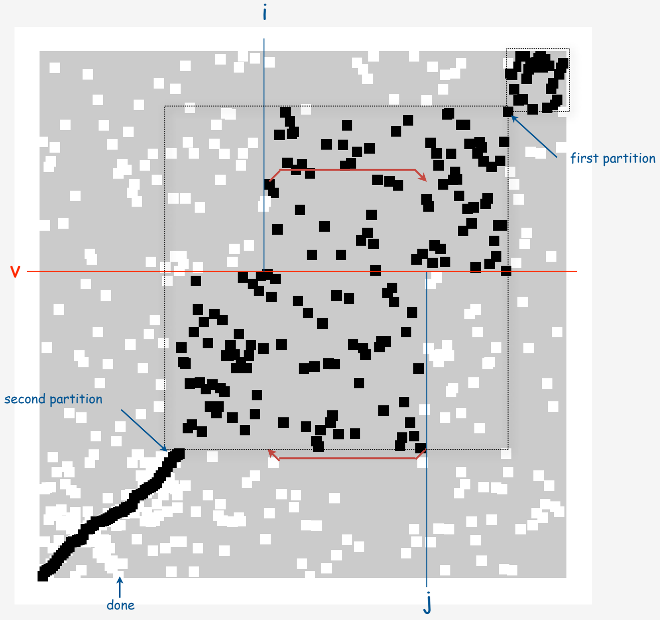
Quicksort: Java implementation

```
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 lo, int hi)
   {
      if (hi <= lo) return;
      int i = partition(a, lo, hi);
      sort(a, lo, i-1);
      sort(a, i+1, hi);
   }
}
```

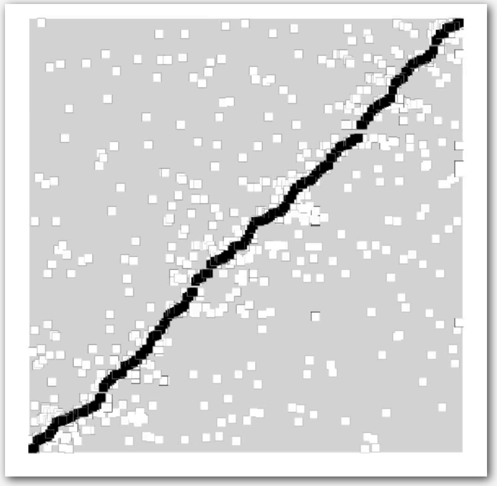
Quicksort trace

	lo	i	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	2	4	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	3	4	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	12	15	A	C	E	E	I	K	L	P	O	R	M	Q	S	X	U	T
no partition for subarrays of size 1	6	10	11	A	C	E	E	I	K	L	P	O	M	Q	R	S	X	U	T
	6	7	9	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	13	15	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
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort animation



Quicksort animation



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 == hi)` test is redundant, but the `(j == lo)` 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 the partitioning element.

Quicksort: empirical analysis

Running time estimates:

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

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.3 sec	6 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.
Lesson 2. Great algorithms are better than good ones.

Quicksort: average-case analysis

Proposition I. The average number of compares C_N to quicksort an array of N elements is ~ 2N ln N (and the number of exchanges is ~ ⅓ N ln N).

Pf. C_N satisfies the recurrence $C_0 = C_1 = 0$ and for $N \geq 2$:

$$C_N = \underset{\substack{\uparrow \\ \text{partitioning}}}{(N + 1)} + \underset{\substack{\uparrow \\ \text{left}}}{(C_0 + C_1 + \ldots + C_{N-1})} / N + \underset{\substack{\uparrow \\ \text{right}}}{(C_{N-1} + C_{N-2} + \ldots + C_0)} \underset{\substack{\uparrow \\ \text{partitioning} \\ \text{probability}}}{/ N}$$

- Multiply both sides by N and collect terms:

$$NC_N = N(N + 1) + 2 (C_0 + C_1 + \ldots + C_{N-1})$$

- Subtract this from the same equation for N-1:

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

- Rearrange terms and divide by N(N+1):

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

Quicksort: average-case analysis

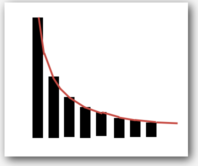
- From before:

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

- Repeatedly apply above equation:

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

- Approximate by an integral:



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

- 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 compares is quadratic.

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

Average case. Number of compares is ~ 1.39 N lg N.

- 39% more compares than mergesort.
- But faster than mergesort in practice because of less data movement.

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

Quicksort: practical improvements

Median of sample.

- Best choice of pivot element = 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 lg N comparisons

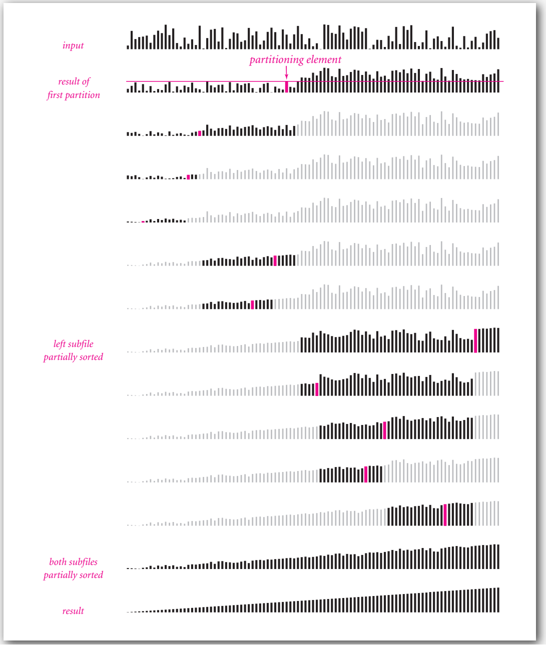
- Median-of-3 random elements.
- Cutoff to insertion sort for ≈ 10 elements.

Non-recursive version.

guarantees O(log N) stack size

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

Quicksort with cutoff to insertion sort: visualization



- quicksort
- selection
- duplicate keys
- system sorts

Selection

Goal. Find the k^{th} largest element.
Ex. Min (k = 0), max (k = N-1), median (k = N/2).

Applications.

- Order statistics.
- Find the "top k."

Use theory as a guide.

- Easy O(N log N) upper bound.
- Easy O(N) upper bound for k = 1, 2, 3.
- Easy Ω(N) lower bound.

Which is true?

- Ω(N log N) lower bound? ← is selection as hard as sorting?
- O(N) upper bound? ← is there a linear-time algorithm for all k?

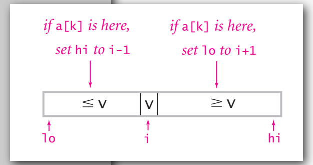
Quick-select

Partition array so that:

- Element `a[i]` is in place.
- No larger element to the left of `i`.
- No smaller element to the right of `i`.

Repeat in one subarray, depending on `i`; finished when `i` equals `k`.

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



Quick-select: mathematical analysis

Proposition. Quick-select takes linear time on average.

Pf sketch.

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

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

Ex. $(2 + 2 \ln 2)\, N$ compares to find the median.

Remark. Quick-select might use $\sim N^2/2$ compares, but as with quicksort, the random shuffle provides a probabilistic guarantee.

Theoretical context for selection

Challenge. Design algorithm whose worst-case running time is linear.

Proposition. [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] There exists a compare-based selection algorithm whose worst-case running time is linear.

Remark. But, algorithm is 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.

Generic methods

In our `select()` implementation, client needs a cast.

```
Double[] a = new Double[N];
for (int i = 0; i < N; i++)
   a[i] = StdRandom.uniform();
Double median = (Double) Quick.select(a, N/2);
```

hazardous cast required

The compiler is also unhappy.

```
% javac Quick.java
Note: Quick.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Q. How to fix?

Generic methods

Safe version. Compiles cleanly, no cast needed in client.

```
public class Quick
{
    public  static <Key extends Comparable<Key>> Key select(Key[] a, int k)
    {  /* as before */  }

    public  static <Key extends Comparable<Key>> void sort(Key[] a)
    {  /* as before */  }

    private static <Key extends Comparable<Key>> int partition(Key[] a, int lo, int hi)
    {  /* as before */  }

    private static <Key extends Comparable<Key>> boolean less(Key v, Key w)
    {  /* as before */  }

    private static <Key extends Comparable<Key>> void exch(Key[] a, int i, int j)
    {  Key swap = a[i]; a[i] = a[j]; a[j] = swap;  }

}
```

generic type variable (value inferred from argument `a[]`)

return type matches array type

can declare variables of generic type

Remark. Obnoxious code needed in system sort; not in this course (for brevity).

- quicksort
- selection
- **duplicate keys**
- system sorts

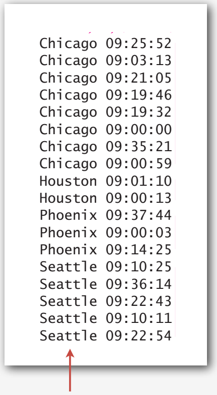
Duplicate keys

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

- Sort population by age.
- Find collinear points. ← see Assignment 3
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge file.
- Small number of key values.



```
Chicago 09:25:52
Chicago 09:03:13
Chicago 09:21:05
Chicago 09:19:46
Chicago 09:19:32
Chicago 09:00:00
Chicago 09:35:21
Chicago 09:00:59
Houston 09:01:10
Houston 09:00:13
Phoenix 09:37:44
Phoenix 09:00:03
Phoenix 09:14:25
Seattle 09:10:25
Seattle 09:36:14
Seattle 09:22:43
Seattle 09:10:11
Seattle 09:22:54
```

key

Duplicate keys

Mergesort with duplicate keys. Always ~ N lg N compares.

Quicksort with duplicate keys.

- Algorithm goes quadratic unless partitioning stops on equal keys!
- 1990s C user found this defect in `qsort()`.

several textbook and system implementations also have this defect

S T O P O N E Q U A L K E Y S

swap

swap

Duplicate keys: the problem

Assume all keys are equal. Recursive code guarantees this case predominates!

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

Consequence. ~ $N^2 / 2$ compares when all keys equal.

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

Recommended. Stop scans on keys equal to the partitioning element.

Consequence. ~ 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

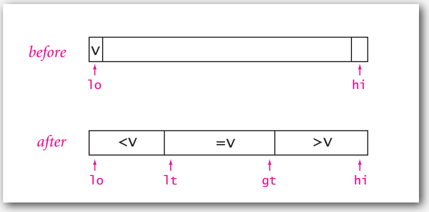
Desirable. Put all keys equal to the partitioning element in place.

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

3-way partitioning

Goal. Partition array into 3 parts so that:

- Elements between `lt` and `gt` equal to partition element `v`.
- No larger elements to left of `lt`.
- No smaller elements to right of `gt`.



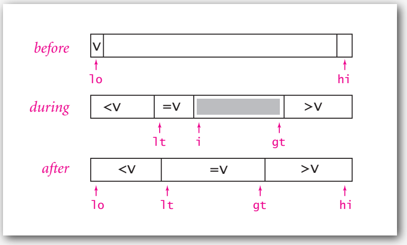
Dutch national flag problem. [Edsger Dijkstra]

- Convention wisdom until mid 1990s: not worth doing.
- New approach discovered when fixing mistake in C library `qsort()`.
- Now incorporated into `qsort()` and Java system sort.

3-way partitioning: Dijkstra's solution

3-way partitioning.

- Let `v` be partitioning element `a[lo]`.
- Scan `i` from left to right.
  - `a[i]` less than `v` : exchange `a[lt]` with `a[i]` and increment both `lt` and `i`
  - `a[i]` greater than `v` : exchange `a[gt]` with `a[i]` and decrement `gt`
  - `a[i]` equal to `v` : increment `i`



All the right properties.

- In-place.
- Not much code.
- Small overhead if no equal keys.

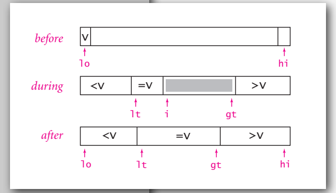
3-way partitioning: trace

			v					a[]						
lt	i	gt	0	1	2	3	4	5	6	7	8	9	10	11
0	0	11	R	B	W	W	R	W	B	R	R	W	B	R
0	1	11	R	B	W	W	R	W	B	R	R	W	B	R
1	2	11	B	R	W	W	R	W	B	R	R	W	B	R
1	2	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	9	B	R	R	B	R	W	B	R	R	W	W	W
2	4	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	8	B	B	R	R	R	W	B	R	R	W	W	W
2	5	7	B	B	R	R	R	R	B	R	W	W	W	W
2	6	7	B	B	R	R	R	R	B	R	W	W	W	W
3	7	7	B	B	B	R	R	R	R	R	W	W	W	W
3	8	7	B	B	B	R	R	R	R	R	W	W	W	W

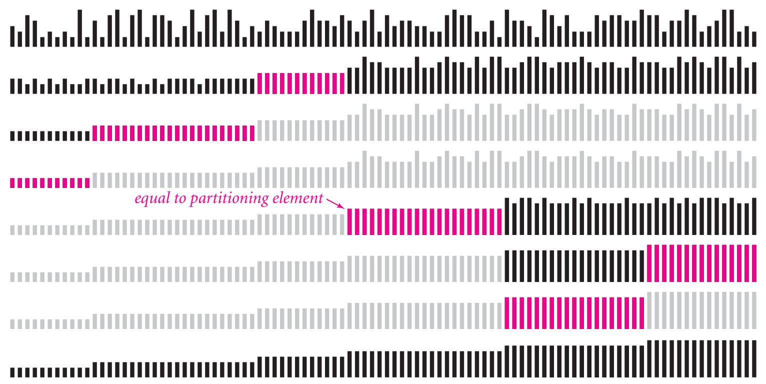
3-way partitioning trace (array contents after each loop iteration)

3-way quicksort: Java implementation

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int lt = lo, gt = hi;
   Comparable v = a[lo];
   int i = lo;
   while (i <= gt)
   {
      int cmp = a[i].compareTo(v);
      if      (cmp < 0) exch(a, lt++, i++);
      else if (cmp > 0) exch(a, i, gt--);
      else              i++;
   }

   sort(a, lo, lt - 1);
   sort(a, gt + 1, hi);
}
```



3-way quicksort: visual trace



Visual trace of quicksort with 3-way partitioning

Duplicate keys: lower bound

Proposition. [Sedgewick-Bentley, 1997] Quicksort with 3-way partitioning is entropy-optimal.

Pf. [beyond scope of course]

- Generalize decision tree.
- Tie cost to Shannon entropy.

Ex. Linear-time when only a constant number of distinct keys.

Bottom line. Randomized quicksort with 3-way partitioning reduces running time from linearithmic to linear in broad class of applications.

- selection
- duplicate keys
- comparators
- system sorts

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!

Java system sorts

Java uses both mergesort and quicksort.

- `Arrays.sort()` sorts array of `Comparable` or any primitive type.
- Uses quicksort for primitive types; mergesort for objects.

```
import java.util.Arrays;

public class StringSort
{
   public static void main(String[] args)
   {
      String[] a = StdIn.readAll().split("\\s+");
      Arrays.sort(a);
      for (int i = 0; i < N; i++)
         StdOut.println(a[i]);
   }
}
```

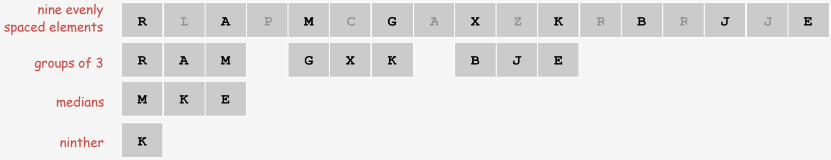
Q. Why use different algorithms, depending on type?

Java system sort for primitive types

Engineering a sort function. [Bentley-McIlroy, 1993]

- Original motivation: improve `qsort()`.
- Basic algorithm = 3-way quicksort with cutoff to insertion sort.
- Partition on Tukey's ninther: median of the medians of 3 samples, each of 3 elements.

approximate median-of-9


nine evenly spaced elements
R L A P M C G A X Z K R B R J J E
groups of 3
R A M G X K B J E
medians
M K E
ninther
K

Why use Tukey's ninther?

- Better partitioning than sampling.
- Less costly than random.

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

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

A killer input.

- Blows function call stack in Java and crashes program.
- Would take quadratic time if it didn't crash first.

more disastrous consequences in C

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

```
% 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

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

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

- Construct malicious input while running system quicksort, in response to elements compared.
- If `v` is partitioning element, commit to `(v < a[i])` and `(v < a[j])`, but don't commit to `(a[i] < a[j])` or `(a[j] > a[i])` until `a[i]` and `a[j]` are compared.

Consequences.

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

Remark. Attack is not effective if file is randomly ordered before sort.

Q. Why do you think system sort is deterministic?

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. Usually.

Sorting summary

	inplace?	stable?	worst	average	best	remarks
selection	×		$N^2/2$	$N^2/2$	$N^2/2$	N exchanges
insertion	×	×	$N^2/2$	$N^2/4$	N	use for small N or partially ordered
shell	×		?	?	N	tight code, subquadratic
quick	×		$N^2/2$	$2N\ln N$	$N\lg N$	$N\log N$ probabilistic guarantee fastest in practice
3-way quick	×		$N^2/2$	$2N\ln N$	N	improves quicksort in presence of duplicate keys
merge		×	$N\lg N$	$N\lg N$	$N\lg N$	$N\log N$ guarantee, stable
???	×	×	$N\lg N$	$N\lg N$	$N\lg N$	holy sorting grail

Which sorting algorithm?

data	data	data	data	data	data	data	data
type	fifo	find	find	exch	hash	exch	exch
hash	hash	hash	hash	fifo	heap	fifo	fifo
heap	heap	heap	heap	find	type	heap	find
sort	exch	leaf	leaf	hash	link	find	hash
link	less	link	link	heap	list	link	heap
list	left	list	list	leaf	push	hash	leaf
push	leaf	push	push	left	sort	left	left
find	find	root	root	less	find	less	less
root	lifo	sort	sort	lifo	leaf	path	lifo
leaf	push	tree	tree	link	root	leaf	link
tree	tree	type	type	list	tree	lifo	list
null	null	exch	null	null	left	next	next
path	path	fifo	path	path	node	root	node
node	node	left	node	node	null	list	null
left	list	less	left	push	path	push	path
less	link	lifo	less	tree	exch	null	push
exch	sort	next	exch	type	less	swap	root
sink	sink	node	sink	sink	sink	node	sink
swim	swim	null	swim	swim	swim	swim	sort
next	next	path	next	next	fifo	sort	swap
swap	swap	sink	swap	swap	lifo	type	swim
fifo	type	swap	fifo	sort	next	sink	tree
lifo	root	swim	lifo	root	swap	tree	type
original	?	?	?	?	?	?	sorted

Which sorting algorithm?

data	data	data	data	data	data	data	data
type	fifo	find	find	exch	hash	exch	exch
hash	hash	hash	hash	fifo	heap	fifo	fifo
heap	heap	heap	heap	find	type	heap	find
sort	exch	leaf	leaf	hash	link	find	hash
link	less	link	link	heap	list	link	heap
list	left	list	list	leaf	push	hash	leaf
push	leaf	push	push	left	sort	left	left
find	find	root	root	less	find	less	less
root	lifo	sort	sort	lifo	leaf	path	lifo
leaf	push	tree	tree	link	root	leaf	link
tree	tree	type	type	list	tree	lifo	list
null	null	exch	null	null	left	next	next
path	path	fifo	path	path	node	root	node
node	node	left	node	node	null	list	null
left	list	less	left	push	path	push	path
less	link	lifo	less	tree	exch	null	push
exch	sort	next	exch	type	less	swap	root
sink	sink	node	sink	sink	sink	node	sink
swim	swim	null	swim	swim	swim	swim	sort
next	next	path	next	next	fifo	sort	swap
swap	swap	sink	swap	swap	lifo	type	swim
fifo	type	swap	fifo	sort	next	sink	tree
lifo	root	swim	lifo	root	swap	tree	type
original	quicksort	mergesort	insertion	selection	merge BU	shellsort	sorted