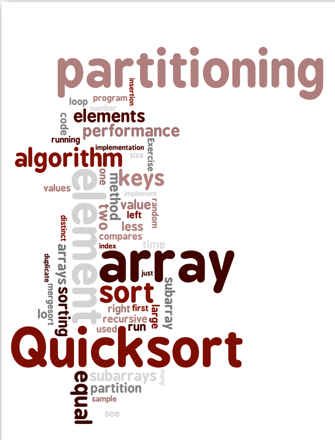
3.3 Quicksort



- quicksort
- selection
- duplicate keys
- system sorts

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 20^{th} century in science and engineering.

Mergesort. ← last lecture

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

Quicksort. ← this lecture

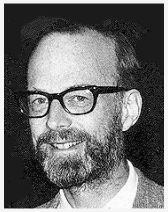
- Java sort for primitive types.
- C qsort, Unix, g++, Visual C++, Python.

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

Quicksort

Basic plan.

- Shuffle the array.
- Partition so that, for some `j`
  - element `a[j]` is in place
  - no larger element to the left of `j`
  - no smaller element to the right of `j`
- 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	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
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

partitioning element · *not greater* · *not less*

Quicksort overview

Quicksort partitioning

Basic plan.

- Scan `i` from left for an item that belongs on the right.
- Scan `j` from right for item item that belongs on the left.
- Exchange `a[i]` and `a[j]`.
- 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	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	0	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

Partitioning trace (array contents before and after each exchange)

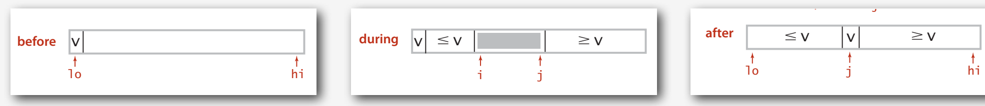
Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{
   int i = lo, j = hi+1;
   while (true)
   {
      while (less(a[++i], a[lo]))          // find item on left to swap
         if (i == hi) break;

      while (less(a[lo], 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, lo, j);                         // swap with partitioning item
   return j;                               // return index of item now known to be in place
}
```



Quicksort: Java implementation

```
public class Quick
{
   private static int partition(Comparable[] a, int lo, int hi)
   {  /* see previous slide */  }

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

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


Quicksort trace

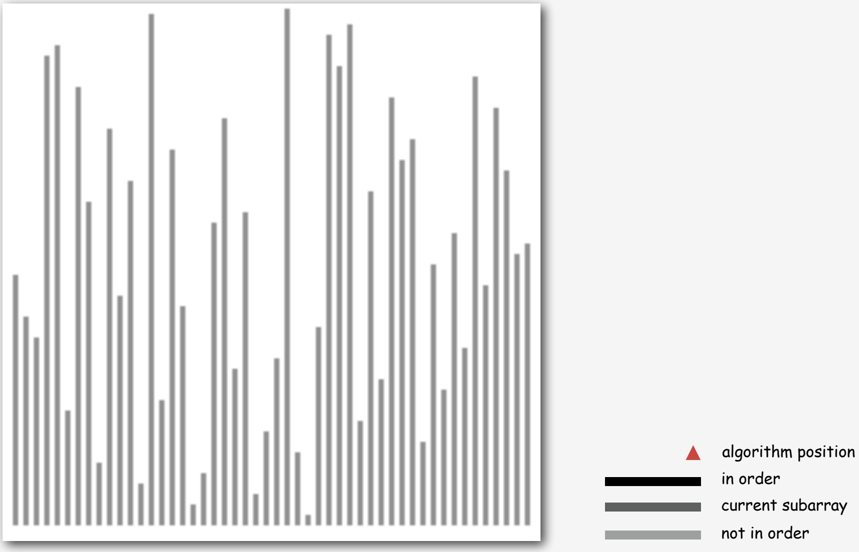
	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	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	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	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

no partition for subarrays of size 1

Quicksort trace (array contents after each partition)

Quicksort animation

50 random elements



http://www.sorting-algorithms.com/quick-sort

Quicksort: implementation details

Partitioning in-place. Using a spare array makes partitioning easier (and stable), 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 `(j == lo)` test is redundant (why?), but the `(i == hi)` test is not.

Preserving randomness. Shuffling is needed 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 compares/second.
- Supercomputer executes 10^{12} compares/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 $N \geq 2$:

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

partitioning probability

- 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:

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

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

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

Quicksort: average-case analysis

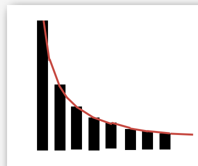
- Repeatedly apply above equation:

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

previous equation

- Approximate sum by an integral:

$$
\begin{aligned}
C_N &\sim 2(N+1)\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{N}\right) \\
&\sim 2(N+1)\int_1^N \frac{1}{x}dx
\end{aligned}
$$



- Finally, the desired result:

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

Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.

- N + (N-1) + (N-2) + … + 1 ~ 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 subarrays.

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

Optimize parameters.

~ 12/7 N ln N compares
~ 12/35 N ln N exchanges

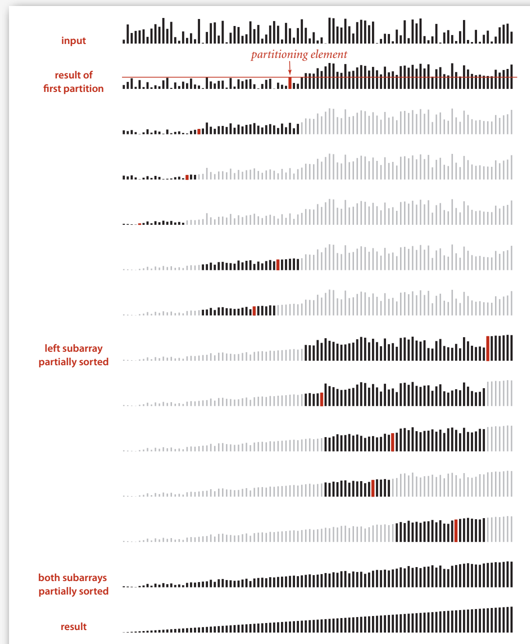
- Median-of-3 random elements.
- Cutoff to insertion sort for ≈ 10 elements.

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 with median-of-3 partitioning and cutoff for small subarrays

- 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?

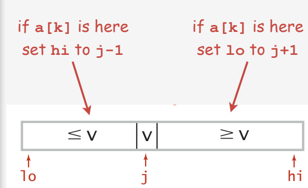
Quick-select

Partition array so that:

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

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

```
public static Comparable select(Comparable[] a, int k)
{
   StdRandom.shuffle(a);
   int lo = 0, hi = a.length - 1;
   while (hi > lo)
   {
      int j = partition(a, lo, hi);
      if      (j < k) lo = j + 1;
      else if (j > k) hi = j - 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 + … + 1 ~ 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 uses ~ $N^2/2$ compares in worst case, 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);
```

unsafe cast required

The compiler also complains.

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

Q. How to fix?

Generic methods

Pedantic (safe) version. Compiles cleanly, no cast needed in client.

```
public class QuickPedantic
{
   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

http://www.cs.princeton.edu/algs4/35applications/QuickPedantic.java.html

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 array.
- 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

if we don't stop on equal keys

if we stop on equal keys

Duplicate keys: the problem

Mistake. Put all keys equal to the partitioning element on one side.
Consequence. $\sim N^2 / 2$ 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

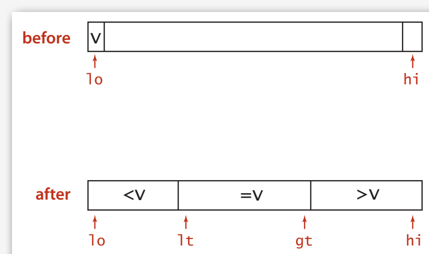
Desirable. Put all keys equal to the partitioning element in place.

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]

- Conventional 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.

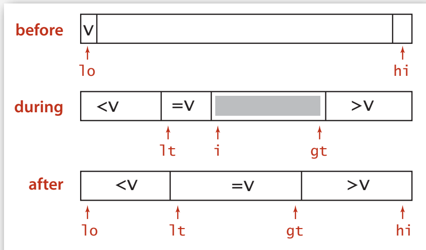
3-way partitioning: Dijkstra's solution

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

3-way partitioning: trace

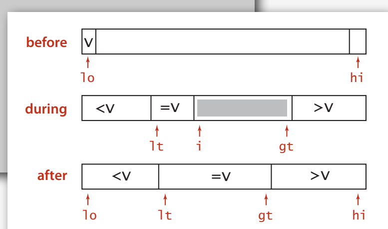
lt	i	gt	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
0	0	11	R	B	W	W	R	W	B	R	R	W	B	R
0	1	11	R	B	W	W	R	W	B	R	R	W	B	R
1	2	11	B	R	W	W	R	W	B	R	R	W	B	R
1	2	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	9	B	R	R	B	R	W	B	R	R	W	W	W
2	4	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	9	B	B	R	R	R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

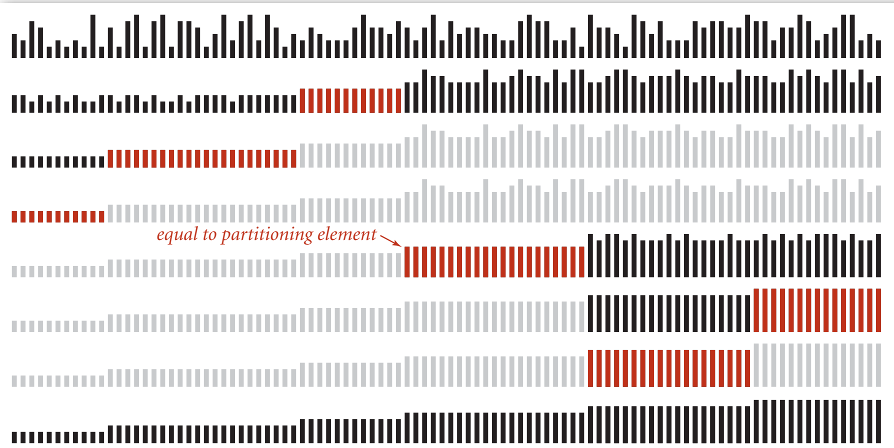
3-way quicksort: Java implementation

```
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

Sorting lower bound. If there are n distinct keys and the i^{th} one occurs x_i times, any compare-based sorting algorithm must use at least

$$\lg\left(\frac{N!}{x_1!\; x_2!\; \cdots\; x_n!}\right) \;\sim\; -\sum_{i=1}^{n} x_i \lg \frac{x_i}{N}$$

N lg N when all distinct;
linear when only a constant number of distinct keys

compares in the worst case.

proportional to lower bound

Proposition. [Sedgewick-Bentley, 1997]
Quicksort with 3-way partitioning is entropy-optimal.
Pf. [beyond scope of course]

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]);
   }
}
```

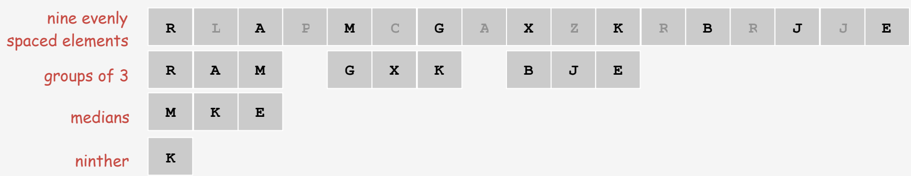
Q. Why use different algorithms, depending on type?

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



Why use Tukey's ninther?

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

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

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

A killer input.

more disastrous consequences 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
...
```

250,000 integers between 0 and 250,000

```
% 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)
   ...
```

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 array is shuffled 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.

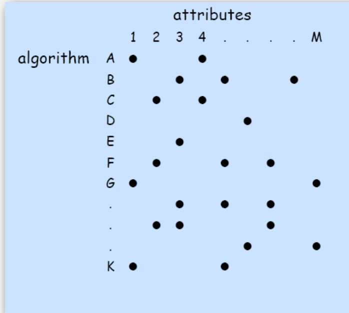
Parallel sorts.

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

System sort: Which algorithm to use?

Applications have diverse attributes.

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



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	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 partially ordered
shell	x		?	?	N	tight code, subquadratic
quick	x		$N^2/2$	$2N \ln N$	$N \lg N$	$N \log N$ probabilistic guarantee fastest in practice
3-way quick	x		$N^2/2$	$2N \ln N$	N	improves quicksort in presence of duplicate keys
merge		x	$N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee, stable
???	x	x	$N \lg N$	$N \lg N$	$N \lg N$	holy sorting grail

Which sorting algorithm?

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