Advanced Topics in Sorting

- selection
- duplicate keys
- system sorts
- comparators

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

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 $\Omega(N)$ lower bound.

Which is true?

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

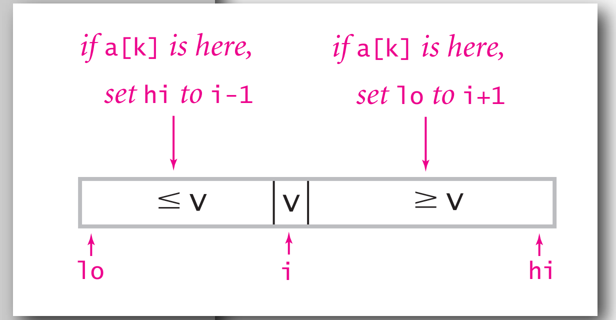
Quick-select

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 a selection algorithm whose running time is linear in the worst-case.

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

Remark. 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.

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

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

return type matches array type

```
    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;  }

}
```

can declare variables of generic type

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

- selection
- **duplicate keys**
- comparators
- applications

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

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
```

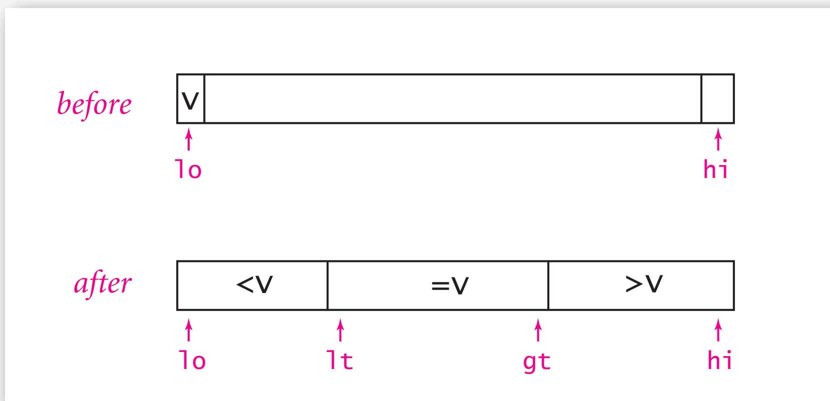
Desirable. Put all keys equal to the partitioning element in place.

```
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.

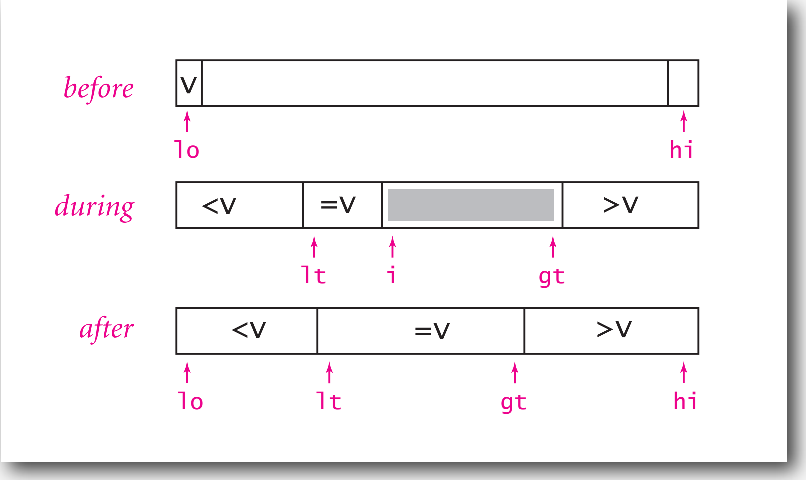
3-way partitioning: Dijkstra's solution

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[]

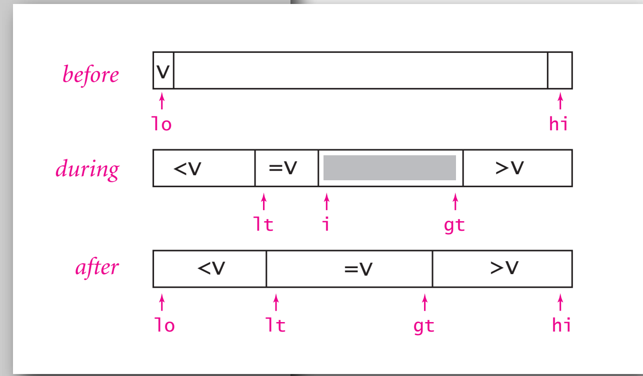
lt	i	gt	0	1	2	3	4	5	6	7	8	9	10	11
0	0	11	R	B	W	W	R	W	B	R	R	W	B	R
0	1	11	R	B	W	W	R	W	B	R	R	W	B	R
1	2	11	B	R	W	W	R	W	B	R	R	W	B	R
1	2	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	9	B	R	R	B	R	W	B	R	R	W	W	W
2	4	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	8	B	B	R	R	R	W	B	R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

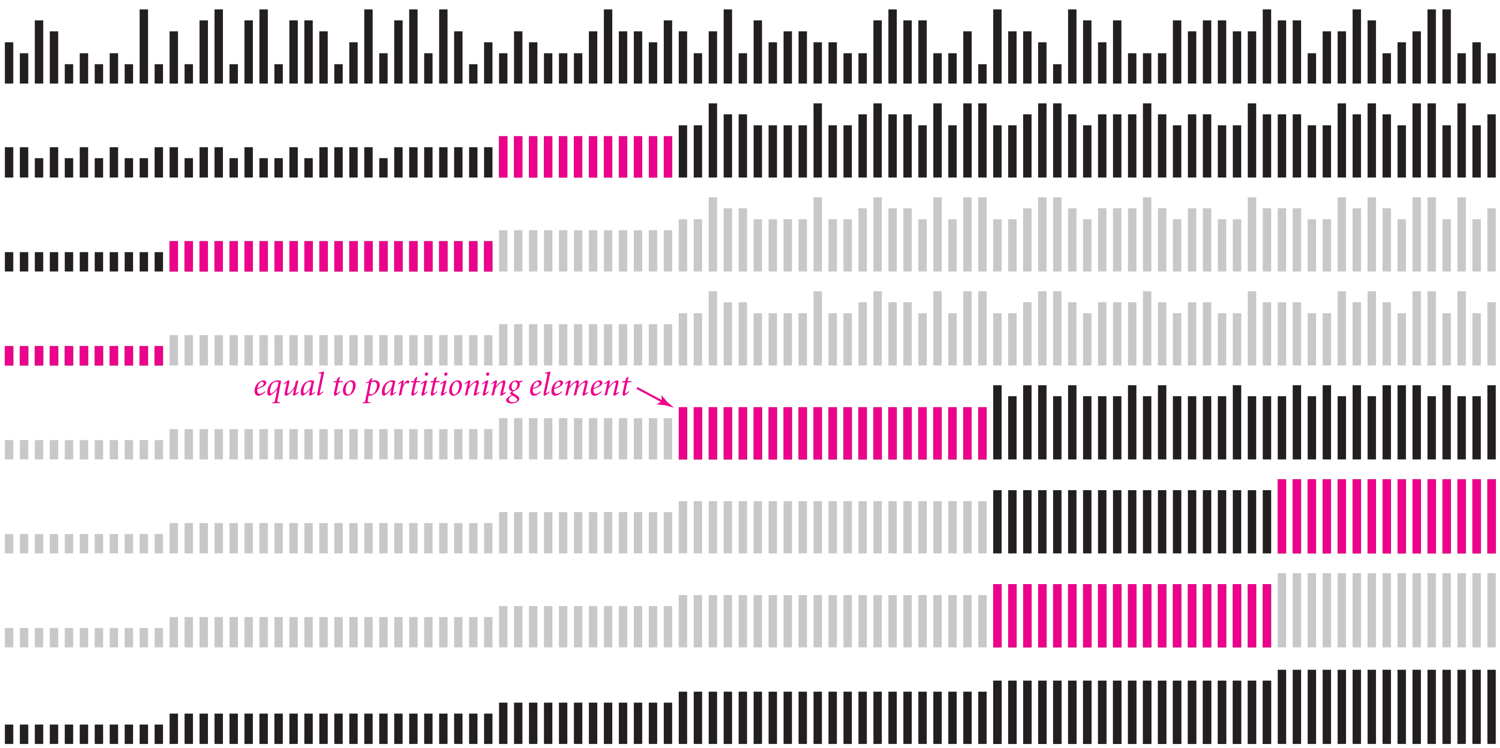
3-way quicksort: Java implementation

```
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**
- applications

Natural order

Comparable interface: sort uses type's natural order.

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

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }
   …
   public int compareTo(Date that)
   {
      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
}
```

natural order

Generalized compare

Comparable interface: sort uses type's natural order.

Problem 1. May want to use a non-natural order.

Problem 2. Desired data type may not come with a "natural" order.

Ex. Sort strings by:

- Natural order. `Now is the time`
- Case insensitive. `is Now the time`
- Spanish. `café cafetero cuarto churro nube ñoño`
- British phone book. `McKinley Mackintosh`

pre-1994 order for digraphs ch and ll and rr

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

`import java.text.Collator;`

Comparators

Solution. Use Java's `Comparator` interface.

```
public interface Comparator<Key>
{
   public int compare(Key v, Key w);
}
```

Remark. The `compare()` method implements a total order like `compareTo()`.

Advantages. Decouples the definition of the data type from the definition of what it means to compare two objects of that type.

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

Comparator example

Reverse order. Sort an array of strings in reverse order.

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

comparator implementation

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

client

Sort implementation with comparators

To support comparators in our sort implementations:

- Pass `Comparator` to `sort()` and `less()`.
- Use it in `less()`.

Ex. Insertion sort.

type variable
(not necessarily `Comparable`)

```
public static <Key> void sort(Key[] a, Comparator<Key> 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 <Key> boolean less(Comparator<Key> c, Key v, Key w)
{  return c.compare(v, w) < 0;   }

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

Generalized compare

Comparators enable multiple sorts of a single file (by different keys).

Ex. Sort 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

Ex. 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 final String name;
   private final 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.

Stability

Q. Which sorts are stable?

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

sorted by time		*sorted by city (unstable)*		*sorted by city (stable)*	
Chicago	09:00:00	Chicago	09:25:52	Chicago	09:00:00
Phoenix	09:00:03	Chicago	09:03:13	Chicago	09:00:59
Houston	09:00:13	Chicago	09:21:05	Chicago	09:03:13
Chicago	09:00:59	Chicago	09:19:46	Chicago	09:19:32
Houston	09:01:10	Chicago	09:19:32	Chicago	09:19:46
Chicago	09:03:13	Chicago	09:00:00	Chicago	09:21:05
Seattle	09:10:11	Chicago	09:35:21	Chicago	09:25:52
Seattle	09:10:25	Chicago	09:00:59	Chicago	09:35:21
Phoenix	09:14:25	Houston	09:01:10	Houston	09:00:13
Chicago	09:19:32	Houston	09:00:13	Houston	09:01:10
Chicago	09:19:46	Phoenix	09:37:44	Phoenix	09:00:03
Chicago	09:21:05	Phoenix	09:00:03	Phoenix	09:14:25
Seattle	09:22:43	Phoenix	09:14:25	Phoenix	09:37:44
Seattle	09:22:54	Seattle	09:10:25	Seattle	09:10:11
Chicago	09:25:52	Seattle	09:36:14	Seattle	09:10:25
Chicago	09:35:21	Seattle	09:22:43	Seattle	09:22:43
Seattle	09:36:14	Seattle	09:10:11	Seattle	09:22:54
Phoenix	09:37:44	Seattle	09:22:54	Seattle	09:36:14

NOT sorted

sorted

Open problem. Stable, inplace, N log N, practical sort??

- selection
- duplicate keys
- comparators
- **system sort**

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.

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
...
```

```
% 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

algorithm	1	2	3	4	.	.	.	.	M
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	×		$N^2 / 2$	$2 N \ln N$	$N \lg N$	$N \log N$ probabilistic guarantee fastest in practice
3-way quick	×		$N^2 / 2$	$2 N \ln N$	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