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 20^{th} 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.

file →	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
record →	Andrews	3	A	874-088-1212	121 Whitman
	Rohde	3	A	232-343-5555	115 Holder
	Battle	4	C	991-878-4944	308 Blair
key →	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

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

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

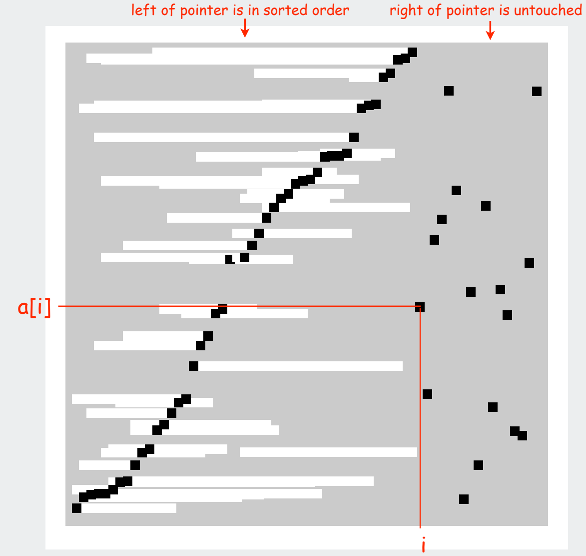
Running time: Quadratic (~c N2)

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



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

big increments:
small subfiles

small increments:
subfiles nearly in order

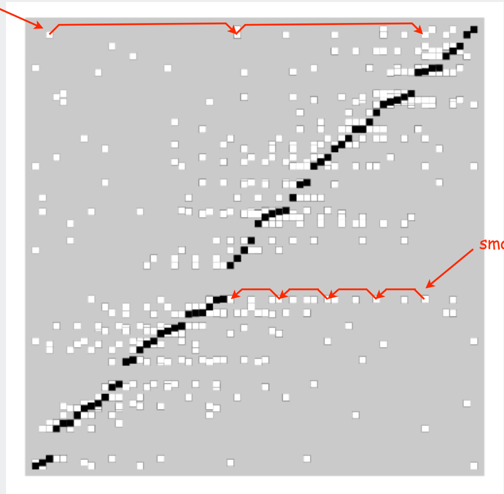
method of choice for both
small subfiles
subfiles nearly in order

insertion sort!

magic increment sequence

```
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



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:

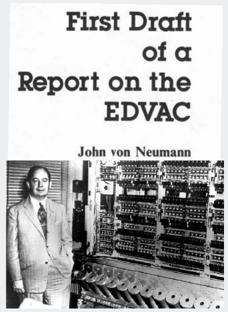
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

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**



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

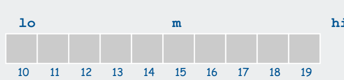
merge

```
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) ≡ number of array stores to mergesort an input of size N

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

left half — right half — merge

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

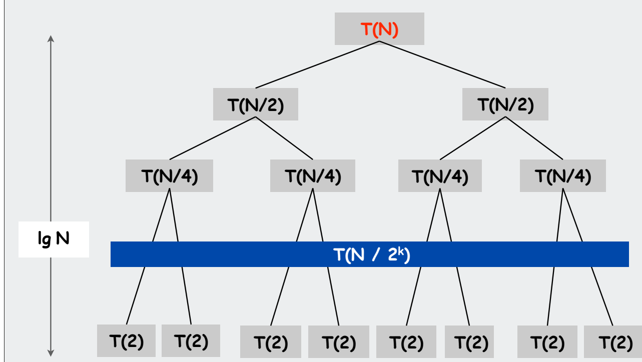
- not quite right for odd N
- same recurrence holds for many algorithms
- same for any input of size N
- comparison count slightly smaller because of array ends

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
. . .	
$= T(N/N)/(N/N) + 1 + 1 + . . . + 1$	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 $T(2N) + 2N \lg (2N)$.

$T(2N) = 2\,T(N) + 2N$	given
$= 2 N \lg N + 2 N$	inductive hypothesis
$= 2 N (\lg (2N) - 1) + 2N$	algebra
$= 2 N \lg (2N)$	QED

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)			Mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 sec	18 min
super	instant	1 second	1.6 weeks	instant	instant	instant

Lesson. Good algorithms are better than supercomputers.

Good enough?

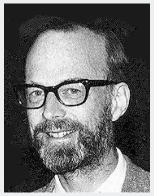
18 minutes might be too long for some applications

- 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

	l	r	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
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	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

no partition for subfiles of size 1

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

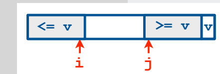
	i	j	r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	-1	15	15	E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
scans →	1	12	15	E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
exchange →	1	12	15	E	C	A	T	E	S	L	P	U	I	M	Q	R	X	O	K
	3	9	15	E	C	A	T	E	S	L	P	U	I	M	Q	R	X	O	K
	3	9	15	E	C	A	I	E	S	L	P	U	T	M	Q	R	X	O	K
	5	5	15	E	C	A	I	E	S	L	P	U	T	M	Q	R	X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

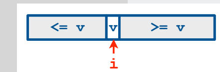
(column heading over 0–15: a[i])

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]))      find item on left to swap
         if (i == r) break;

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

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

      exch(a, i, j);                  swap
   }

   exch(a, i, r);                     swap with partitioning item
   return i;                          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 $N \geq 2$:

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

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

- 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

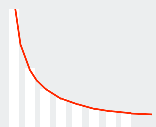
$$NC_N = (N+1)C_{N-1} + 2N$$

- 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 ~ 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.
- Median-of-3 random elements. ≈ 12/7 N log N comparisons
- Cutoff to insertion sort for ≈ 10 elements.

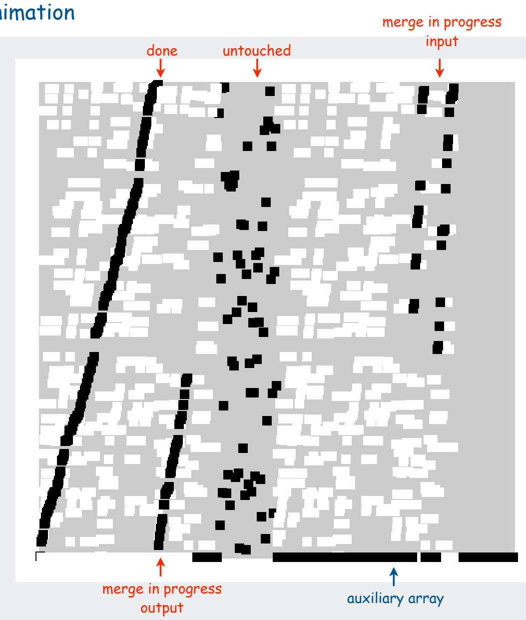
Non-recursive version.
- Use explicit stack.
- Always sort smaller half first. guarantees O(log N) stack size

All validated with refined math models and experiments

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

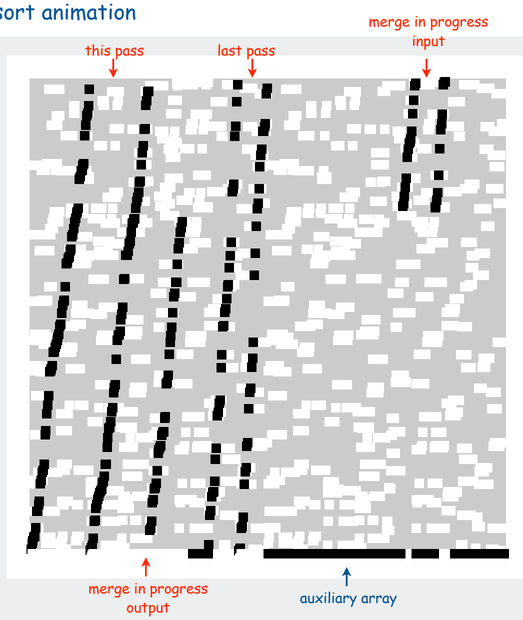
49

Mergesort animation



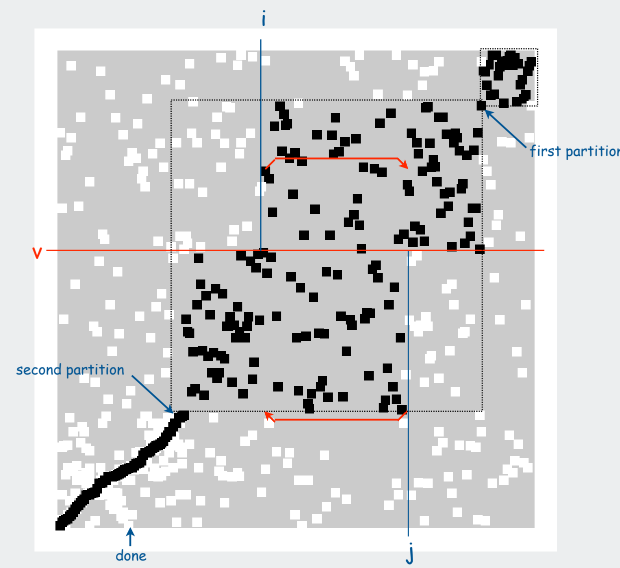
50

Bottom-up mergesort animation



51

Quicksort animation



52