Mergesort and Quicksort

- mergesort
- mergesort analysis
- quicksort
- quicksort analysis
- animations

Reference: Algorithms in Java, Chapters 7 and 8

Mergesort and Quicksort

Two great sorting algorithms.

- Full scientific understanding of their properties has enabled us to hammer them into practical system sorts.
- Occupy a prominent place in world's computational infrastructure.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort.

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

Quicksort.

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

Mergesort

Basic plan:

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

input

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

sort left

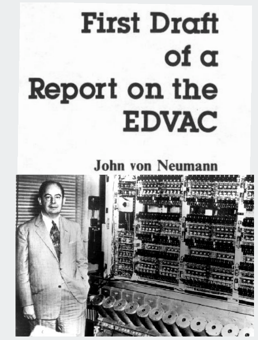
```
E E G M O R R S T E X A M P L E
```

sort right

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

merge

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



Mergesort: Example

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

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

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				

```
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++];

}
```

copy

merge

mergesort

mergesort analysis

quicksort

quicksort analysis

animations

Mergesort: Java implementation of recursive sort

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

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

l					m					r
10	11	12	13	14	15	16	17	18	19	

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: Comparison count

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

$$= \underbrace{T(N/2)}_{\text{left half}} + \underbrace{T(N/2)}_{\text{right half}} + \underbrace{N}_{\text{merge}}$$

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

- not quite right for odd N
- same recurrence holds for many algorithms
- same number of comparisons for any input of size N.

$\lg N \equiv \log_2 N$

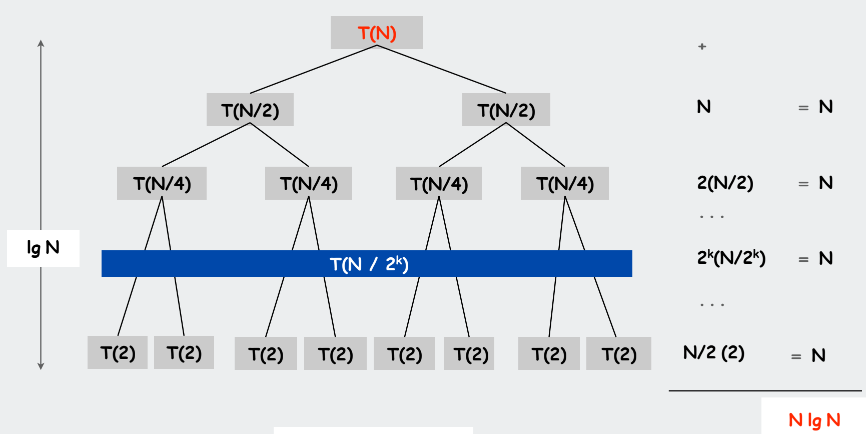
Solution of Mergesort recurrence $\quad T(N) \sim N \lg N$

- true for all N, as long as integer approx of N/2 is within a constant
- easy to prove when N is a power of 2.

can then use induction for general N (see COS 341)

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)



$T(N) = N \lg N$

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
...		
$= T(N/N)/(N/N) + 1 + 1 + \ldots + 1$		stop telescoping, $T(1) = 0$
$= \lg N$		

$T(N) = N \lg N$

Mergesort recurrence: Proof 3 (by induction)

$T(N) = 2\,T(N/2) + N$ for $N > 1$, with $T(1) = 0$ (assume that N is a power of 2)

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

Pf. [by induction on N]

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

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

Ex. (for COS 341). 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

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

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

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));
   }
}
```

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 1. Good algorithms are better than supercomputers.

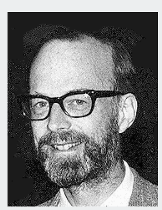
mergesort
mergesort analysis
quicksort
quicksort analysis
animations

Quicksort

Basic plan.

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

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
```



Sir Charles Antony Richard Hoare
1980 Turing Award

Quicksort Partitioning

Q. How do we partition in-place efficiently?
A. Scan from right, scan from left, exchange

input
```
E R A T E S L P U I M Q C X O K
```
scan left, scan right
```
E R A T E S L P U I M Q C X O K
```
exchange
```
E C A T E S L P U I M Q R X O K
```
scan left, scan right
```
E C A T E S L P U I M Q R X O K
```
exchange
```
E C A I E S L P U T M Q R X O K
```
scan left, scan right
```
E C A I E S L P U T M Q R X O K
```
final exchange
```
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
```


Quicksort example

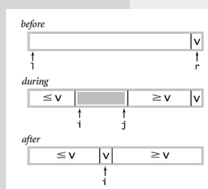
```
Q U I C K S O R T E X A M P L E

E R A T E S L P U I M Q C X O K
E C A I E K L P U T M Q R X O S
A C E I E K L P U T M Q R X O S
A C E I E K L P U T M Q R X O S
A C E I E K L P U T M Q R X O S
A C E E I K L P U T M Q R X O S
A C E E I K L P U T M Q R X O S
A C E E I K L P O R M Q S X U T
A C E E I K L P O M Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S T U X
A C E E I K L M O P Q R S T U X
A C E E I K L M O P Q R S X U T

A C E E I K L M O P Q R S T U X
```

Quicksort: Java implementation of recursive sort

```java
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: Java implementation of partitioning procedure

```java
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 element
   return i;                          // return index where crossing occurs
}
```



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.

mergesort
mergesort analysis
quicksort
quicksort analysis
animations

Quicksort: Average-case analysis

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

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

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

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

- Multiply both sides by N

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

- Subtract the same formula for N-1:

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

- Simplify:

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

Quicksort: Average Case

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

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

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

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

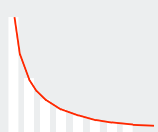
$$= C_{N-3} / (N-2) + 2/(N-1) + 2/N + 2/(N+1)$$

$$= 2\,\big(1 + 1/2 + 1/3 + \ldots + 1/N + 1/(N+1)\big)$$

- Approximate the exact answer by an integral:

$$C_N \approx 2(N+1)\big(1 + 1/2 + 1/3 + \ldots + 1/N\big)$$

$$= 2(N+1)\,H_N \approx 2(N+1)\int_1^N dx/x$$



- 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.

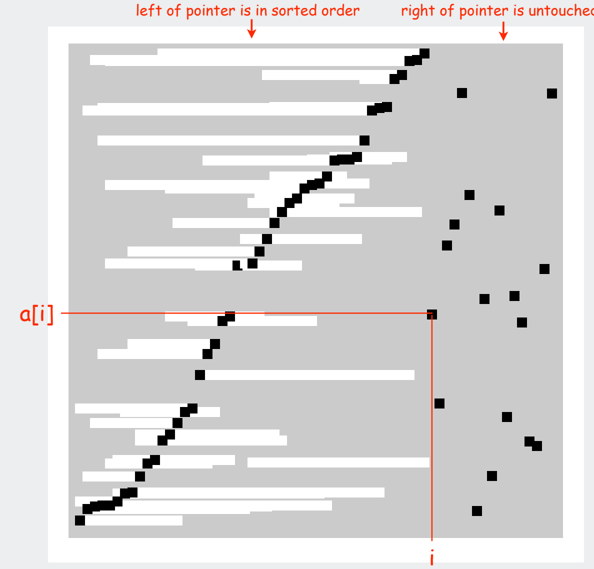
Optimize parameters.
- Median-of-3 random elements. ≈ 12/7 N log N comparisons
- Cutoff to insertion sort for ≈ 10 elements.

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

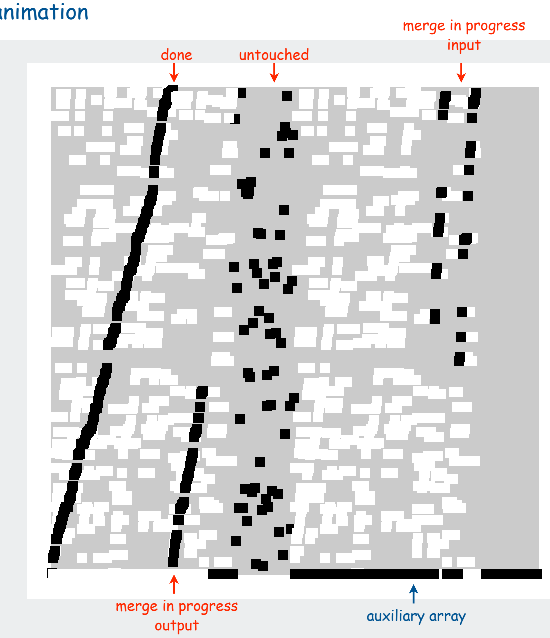
All validated with refined math models and experiments

mergesort
mergesort analysis
quicksort
quicksort analysis
animations

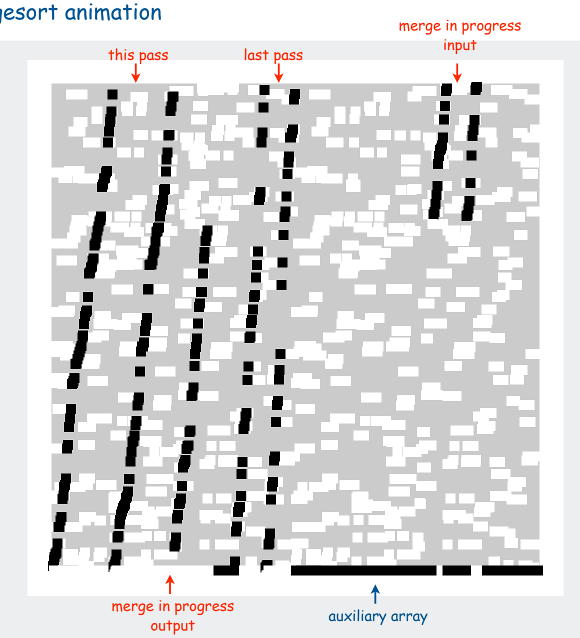
Insertion sort animation



Mergesort animation



Bottom-up mergesort animation



Quicksort animation