Efficient Sorting Algorithms I

Mark Matthews PhD

Summary

- Sorting so far…
- Efficient sorting algorithms
- Divide & Conquer
- MergeSort
- MergeSort improvements
- Bottom Up MergeSort

Two classic sorting algorithms: mergesort and quicksort

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.

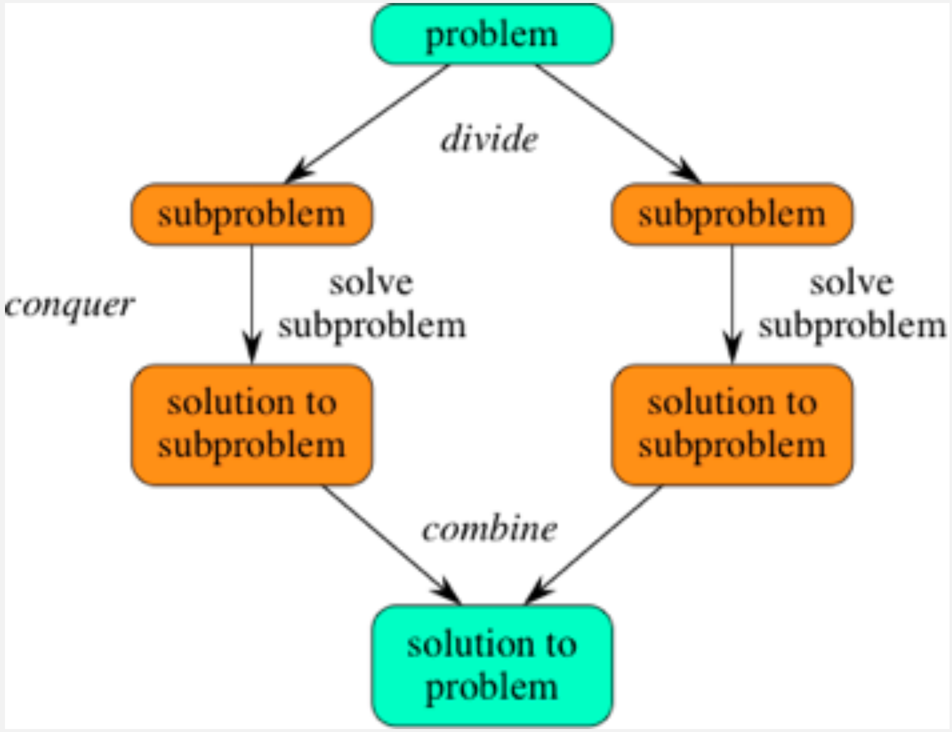
Mergesort.

JS …

Quicksort.

UNIX JS …

Divide & Conquer Paradigm



Both mergesort & quick sort use Divide and Conquer which is based on recursion

3 main parts

1. **Divide** the problem into smaller sub-problems
2. **Conquer** the sub-problems recursively (don't forget your base case)
3. **Combine** the solutions to the sub-problems into the solution for the original problem

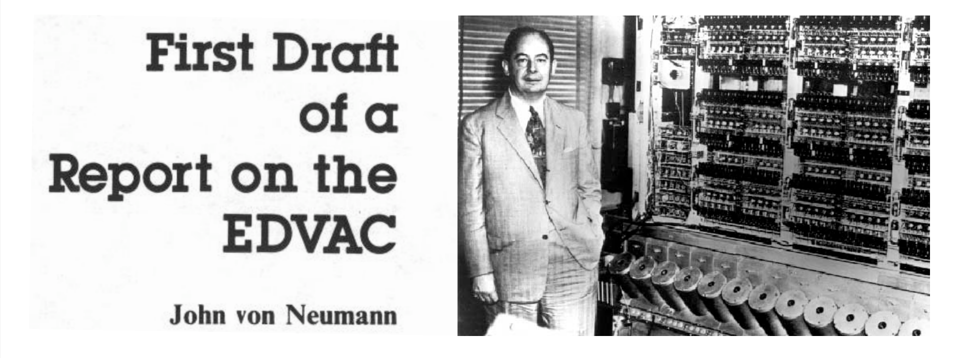
Mergesort

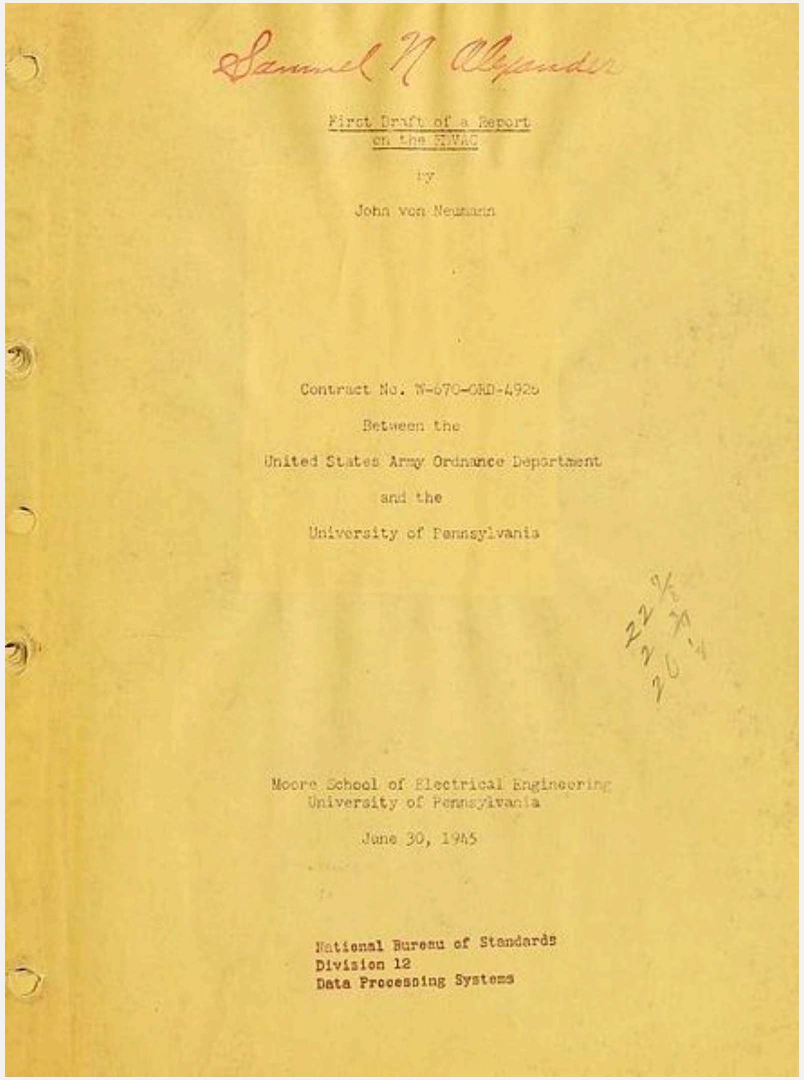
Basic plan.

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

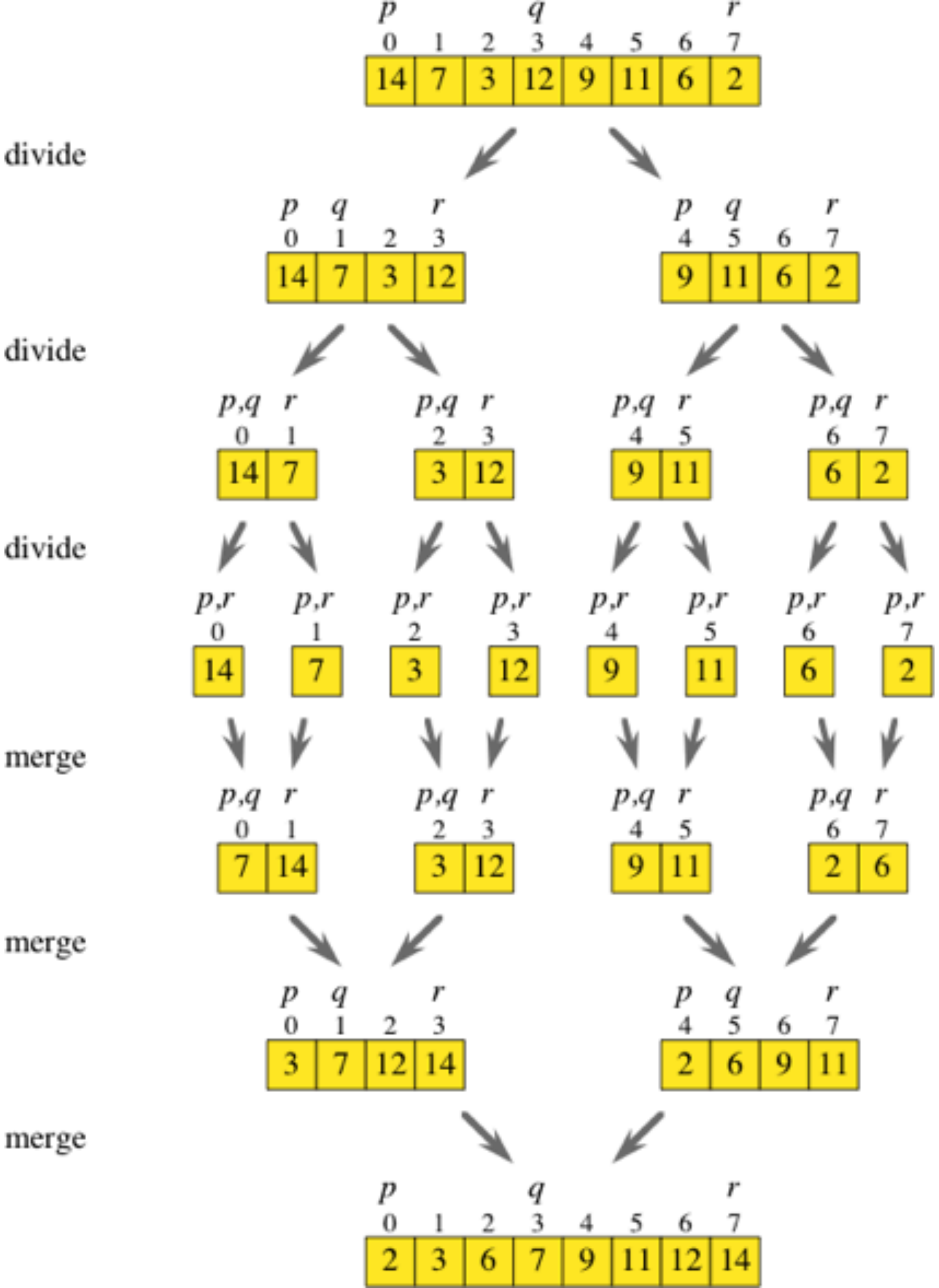
input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview





https://fermatslibrary.com/s/von-neumanns-first-computer-program



p q r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 2
divide
divide
divide
merge
merge
merge
2 3 6 7 9 11 12 14

Merge Sort Demo

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

	lo				mid	mid+1				hi
a[]	E	E	G	M	R	A	C	E	R	T
			sorted					sorted		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

	lo				mid	mid+1				hi
a[]	E	E	G	M	R	A	C	E	R	T

copy to auxiliary array

aux[]										

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	E	E	G	M	R	A	C	E	R	T

aux[]	E	E	G	M	R	A	C	E	R	T

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	E	E	G	M	R	A	C	E	R	T
	k									

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i					j				

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	E	G	M	R	A	C	E	R	T
	k									

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i					j				

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	E	G	M	R	A	C	E	R	T
		k								

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i						j			

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

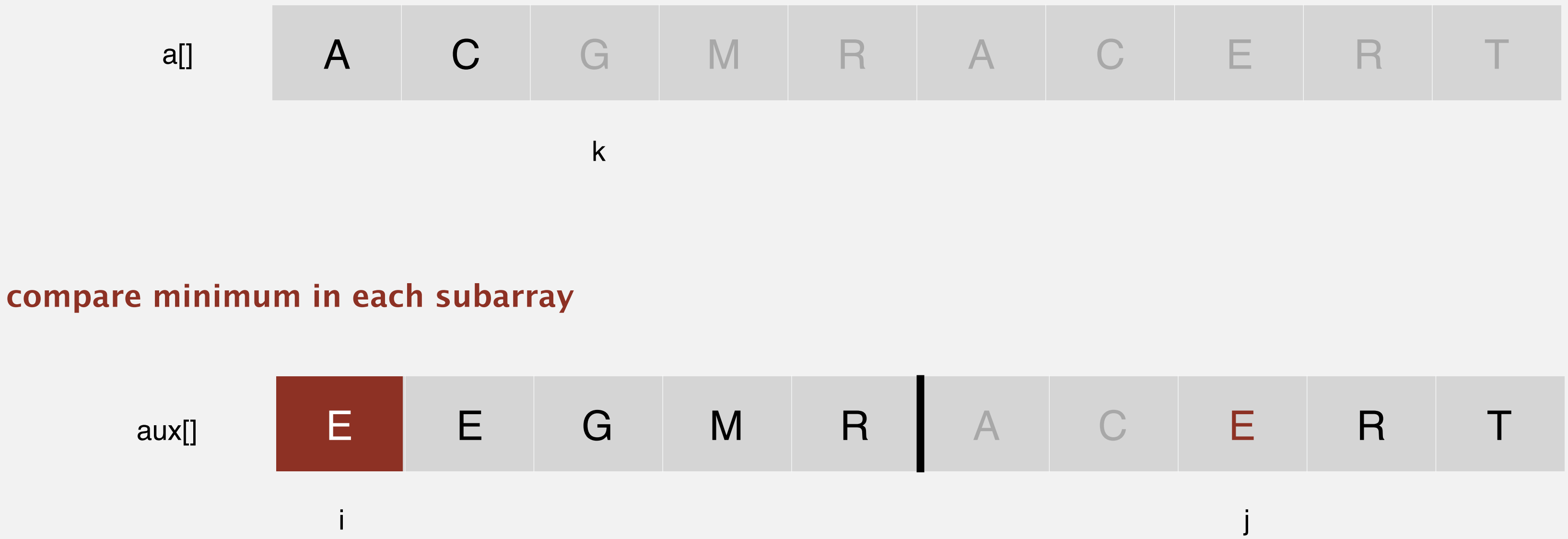
a[]	A	C	G	M	R	A	C	E	R	T
		k								

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i						j			

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].



Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	M	R	A	C	E	R	T
			k							

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i							j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	M	R	A	C	E	R	T
				k						

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
		i						j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	R	A	C	E	R	T
				k						

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
		i						j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	R	A	C	E	R	T
					k					

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i					j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	A	C	E	R	T
					k					

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i					j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	A	C	E	R	T
						k				

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i						j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	C	E	R	T
						k				

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i						j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	C	E	R	T
							k			

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
				i					j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

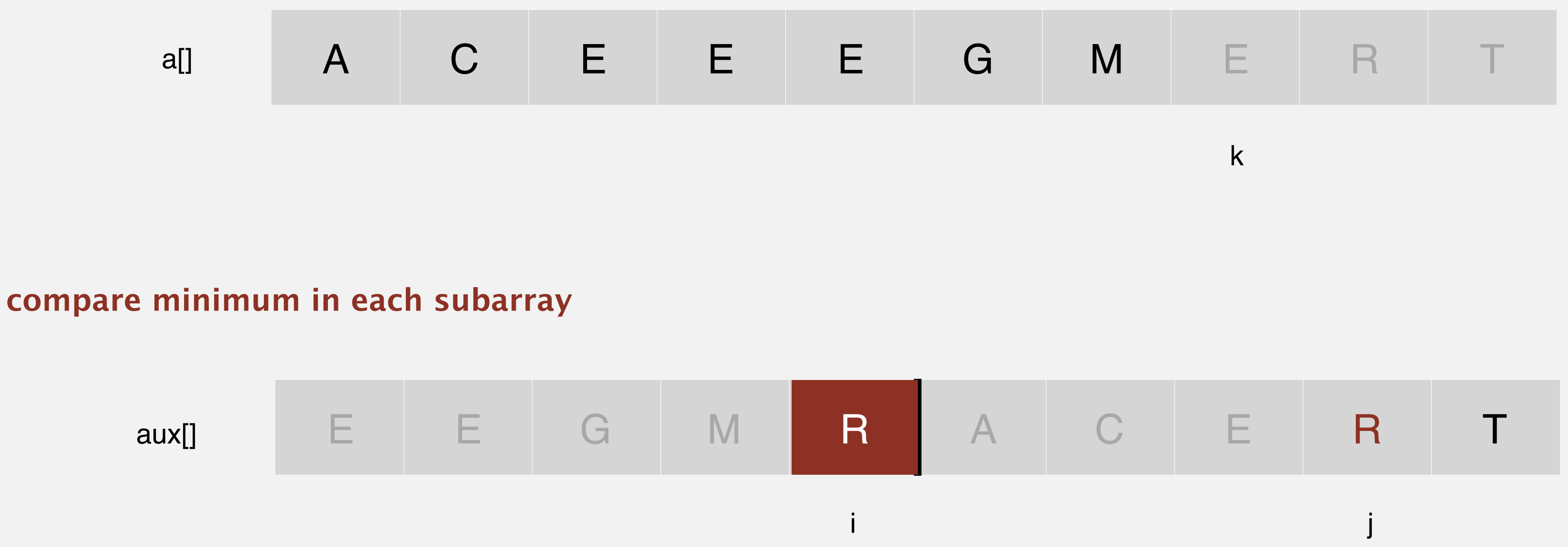
a[]	A	C	E	E	E	G	M	E	R	T
							k			

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
				i					j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].



Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T
								k		

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
					i				j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T
									k	

one subarray exhausted, take from other

aux[]	E	E	G	M	R	A	C	E	R	T
						i			j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T
									k	

one subarray exhausted, take from other

aux[]	E	E	G	M	R	A	C	E	R	T
						i			j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T
										k

one subarray exhausted, take from other

aux[]	E	E	G	M	R	A	C	E	R	T
						i				j

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T
										k

one subarray exhausted, take from other

aux[]	E	E	G	M	R	A	C	E	R	T
						i				j

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T

k

both subarrays exhausted, done

aux[]	E	E	G	M	R	A	C	E	R	T

i j

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

	lo									hi
a[]	A	C	E	E	E	G	M	R	R	T

sorted

Merging: Pseudocode

Input: unsorted array
Output: sorted array

```
function mergeSort (int[] a){

N = array.length;

//base case
if (n == 1){
return array;
}

//create left and right sub-arrays
left = mergeSort(left);
right = mergeSort(right);

mergeArray = merge(left, right);

return mergedArray;
}
```

Input = 2 sorted arrays, a, b
Output = 1 sorted array, S

```
function merge (int[] a, int[] b){

//repeat while both arrays have elements in them
while (a.notEmpty() && b.notEmpty()){

//if element in 1st array is <= 1st element in 2nd array
if (a.firstElement <= b.firstElement){
S.insertLast(a.removeFirst());
} else if (b.firstElement <= a.firstElement){
S.insertLast(b.removeFirst());
}

//when while loop ends
If (a.notEmpty()){
//add remaining elements in a to S
} else if (b.notEmpty()){
//add remaining elements in b to S
}

return S;
```

Merging: Java implementation

```java
public static void mergeSort(int[] a, int n) {
        if (n < 2)
            return;
        int mid = n / 2;
        int[] l = new int[mid];
        int[] r = new int[n - mid];

        for (int i = 0; i < mid; i++) {
            l[i] = a[i];
        }
        for (int i = mid; i < n; i++) {
            r[i - mid] = a[i];
        }
        mergeSort(l, mid);
        mergeSort(r, n - mid);

        merge(a, l, r, mid, n - mid);
    }
```

```java
public static void merge(int[] a, int[] l, int[] r, int left, int right) {

        int i = 0, j = 0, k = 0;

        while (i < left && j < right) {

            if (l[i] <= r[j])
                a[k++] = l[i++];
            else
                a[k++] = r[j++];

        }

        while (i < left)
            a[k++] = l[i++];

        while (j < right)
            a[k++] = r[j++];
    }
}
```

Mergesort: trace

Basic plan:

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

lo hi

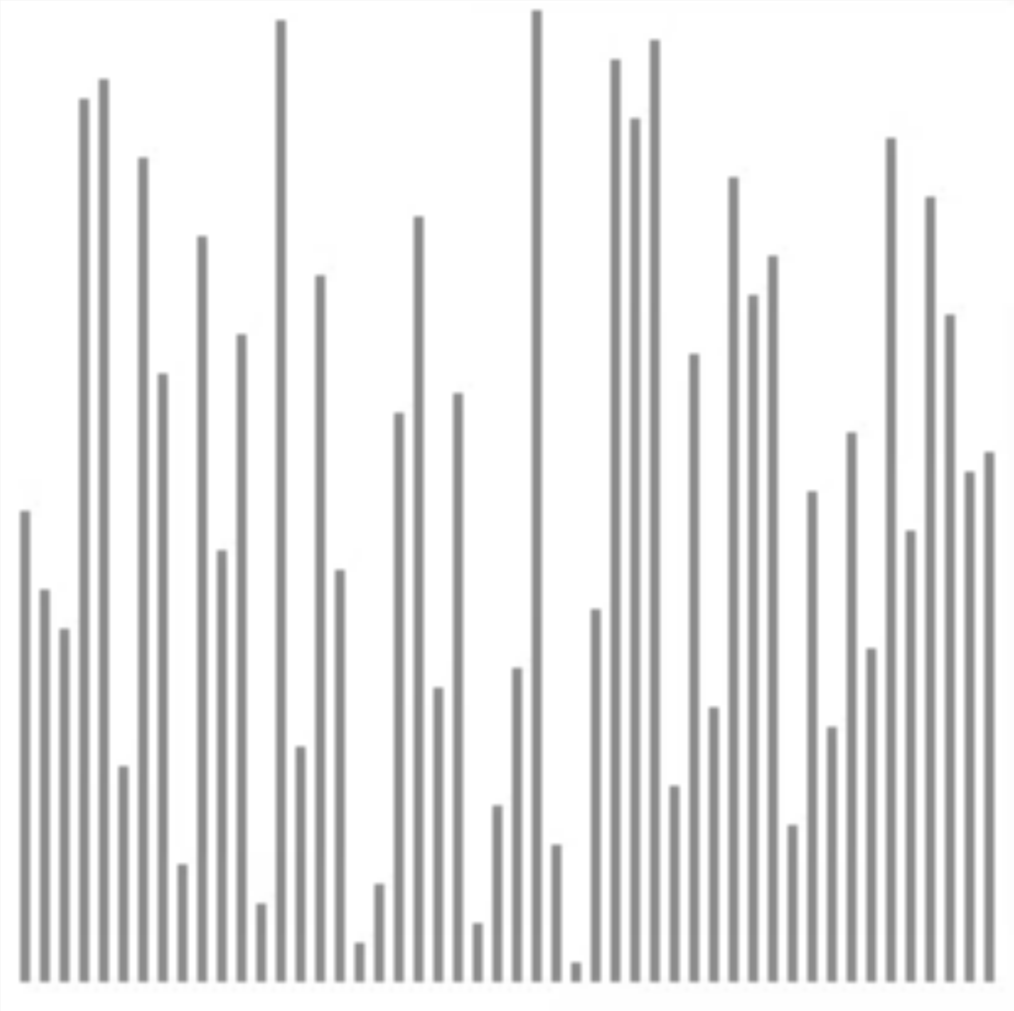
a[]

	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
`merge(a, aux, 0, 0, 1)`	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 2, 2, 3)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 0, 1, 3)`	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 4, 4, 5)`	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 6, 6, 7)`	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 4, 5, 7)`	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
`merge(a, aux, 0, 3, 7)`	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
`merge(a, aux, 8, 8, 9)`	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
`merge(a, aux, 10, 10, 11)`	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
`merge(a, aux, 8, 9, 11)`	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
`merge(a, aux, 12, 12, 13)`	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
`merge(a, aux, 14, 14, 15)`	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
`merge(a, aux, 12, 13, 15)`	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
`merge(a, aux, 8, 11, 15)`	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
`merge(a, aux, 0, 7, 15)`	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

result after recursive call

Mergesort: animation

50 random items



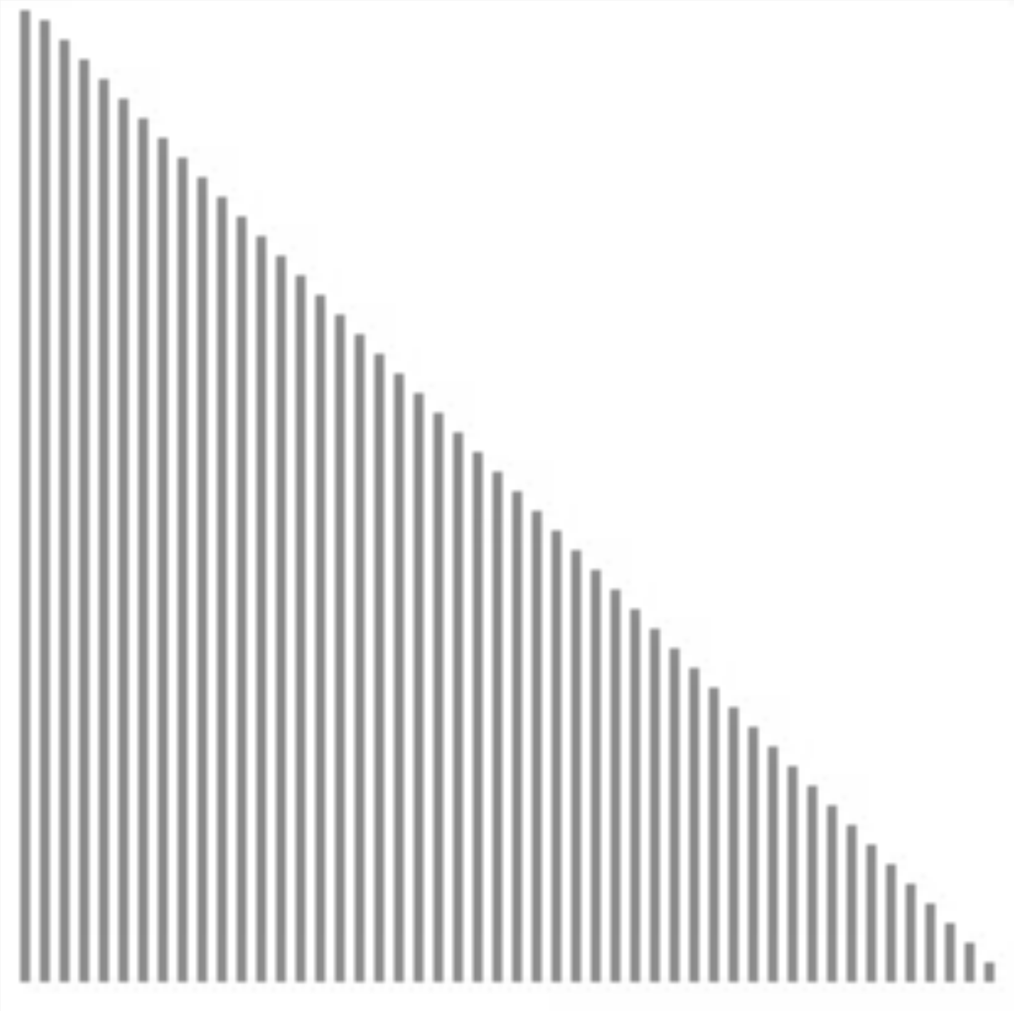
▲ algorithm position
in order
current subarray
not in order

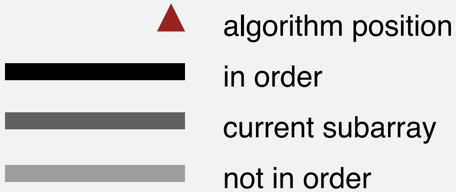
Reason it is slow: excessive data movement.

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

Mergesort: animation

50 reverse-sorted items





Reason it is slow: excessive data movement.

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

Merge Sort Complexity

Mergesort: empirical analysis

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

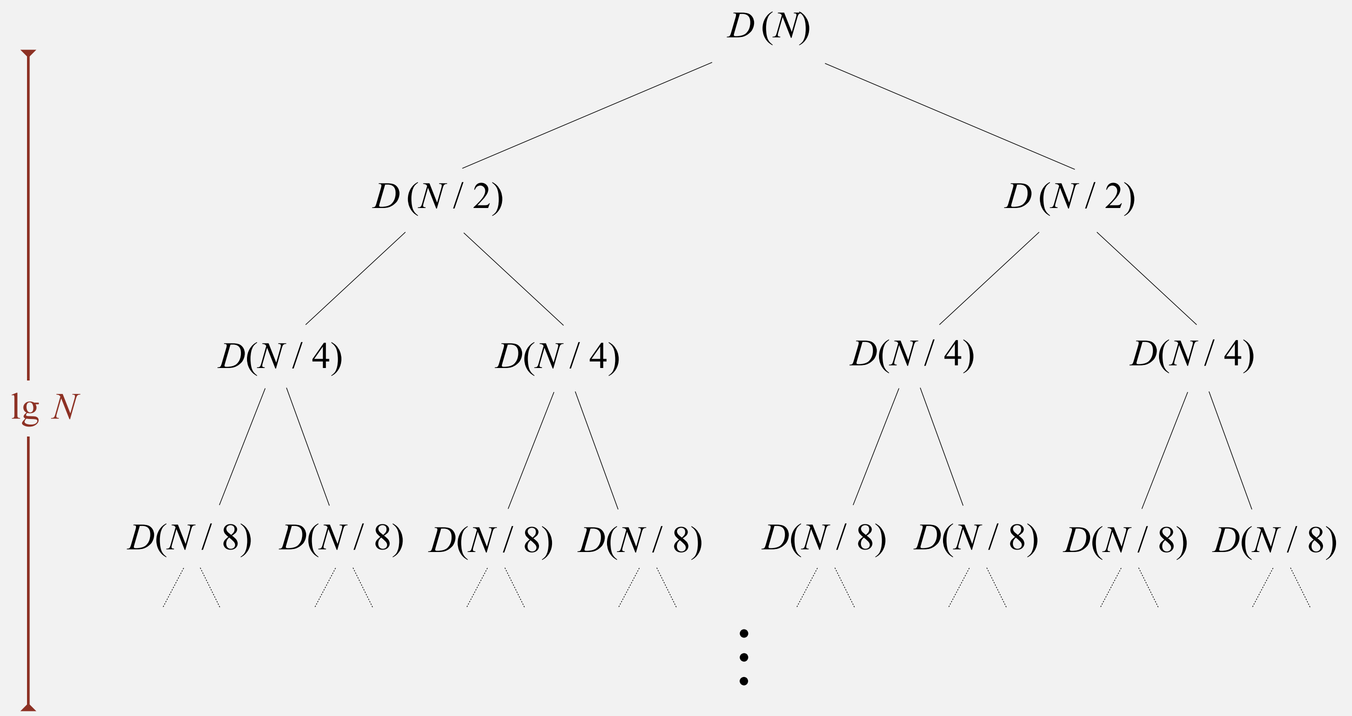
	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Bottom line. Good algorithms are better than supercomputers.

MergeSort performance

Mergesort uses ≤ N lg N compares to sort an array of length N.

Pf 1. [assuming N is a power of 2]



$$N = N$$

$$2\,(N/2) = N$$

$$4\,(N/4) = N$$

$$8\,(N/8) = N$$

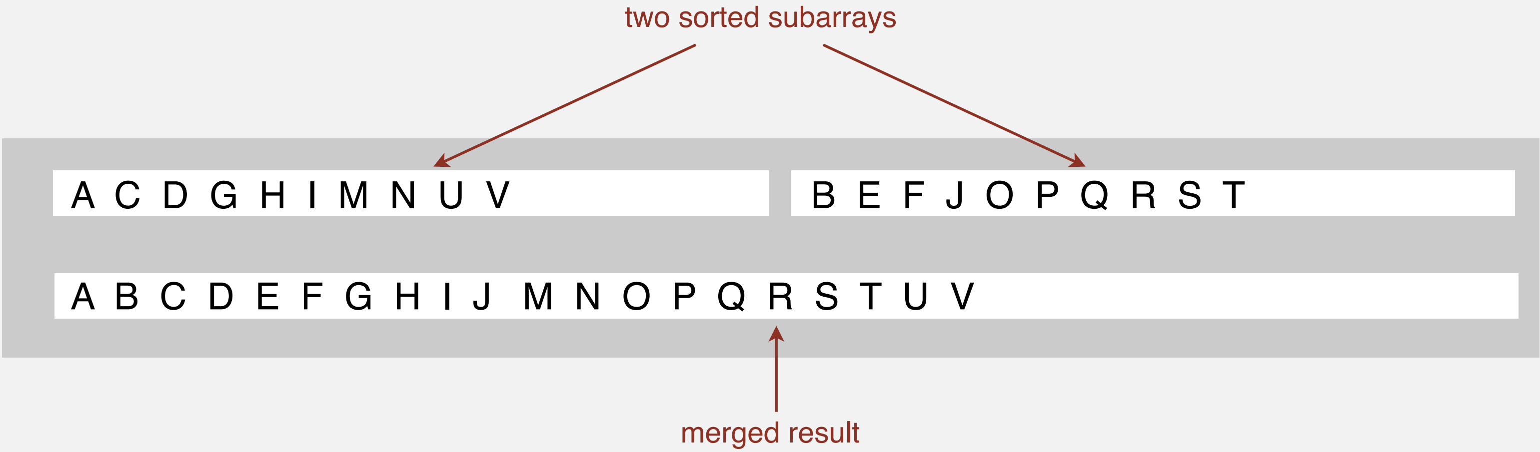
$$\vdots$$

$$T(N) = N \lg N$$

Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to N.

Pf. The array aux[] needs to be of length N for the last merge.



Challenge 1 (not hard). Use aux[] array of length $\sim \frac{1}{2} N$ instead of N.

Challenge 2 (very hard). In-place merge. [Kronrod 1969]

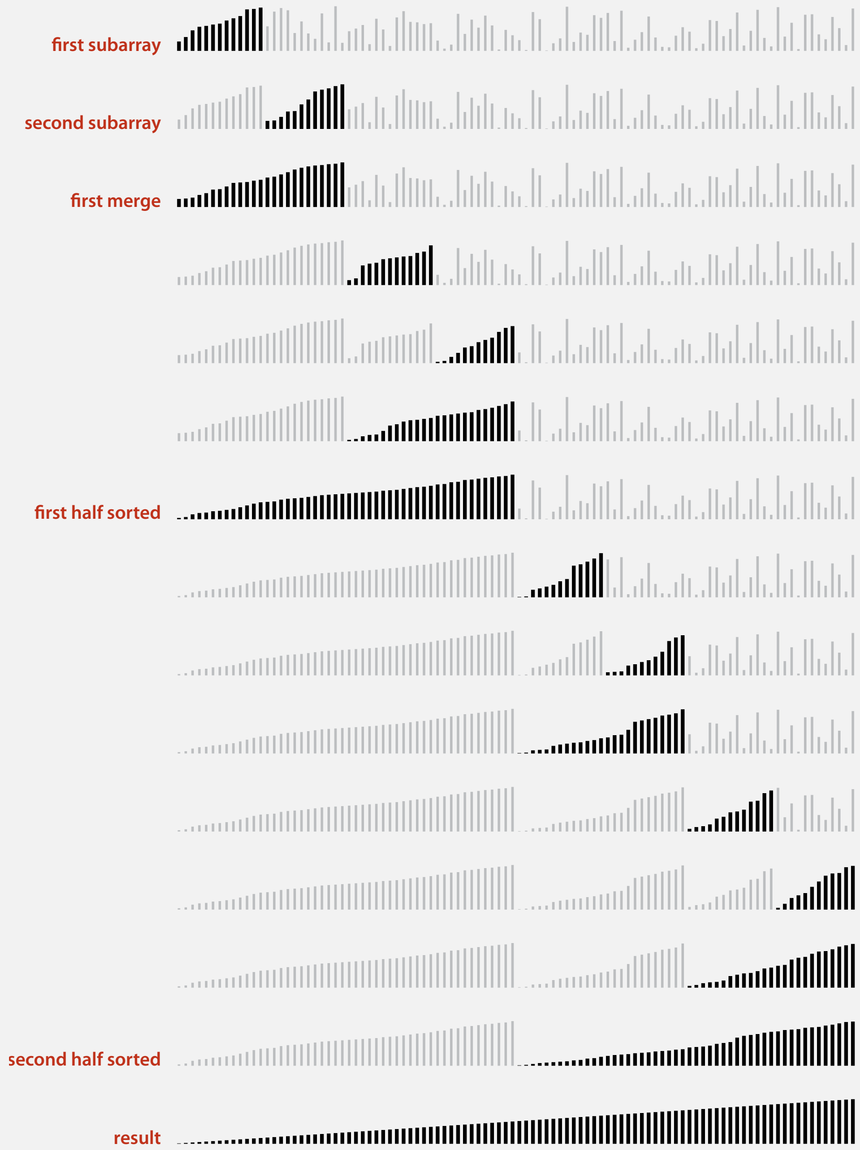
Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
  if (hi <= lo + CUTOFF - 1)
  {
    Insertion.sort(a, lo, hi);
    return;
  }
  int mid = lo + (hi - lo) / 2;
  sort (a, aux, lo, mid);
  sort (a, aux, mid+1, hi);
  merge(a, aux, lo, mid, hi);
}
```

Mergesort with cutoff to insertion sort: visualization



Mergesort: practical improvements

Stop if already sorted.

- Is largest item in first half ≤ smallest item in second half?
- Helps for partially-ordered arrays.

A B C D E F G H I J	M N O P Q R S T U V

A B C D E F G H I J M N O P Q R S T U V

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   if (!less(a[mid+1], a[mid])) return;
   merge(a, aux, lo, mid, hi);
}
```

Java 6 system sort

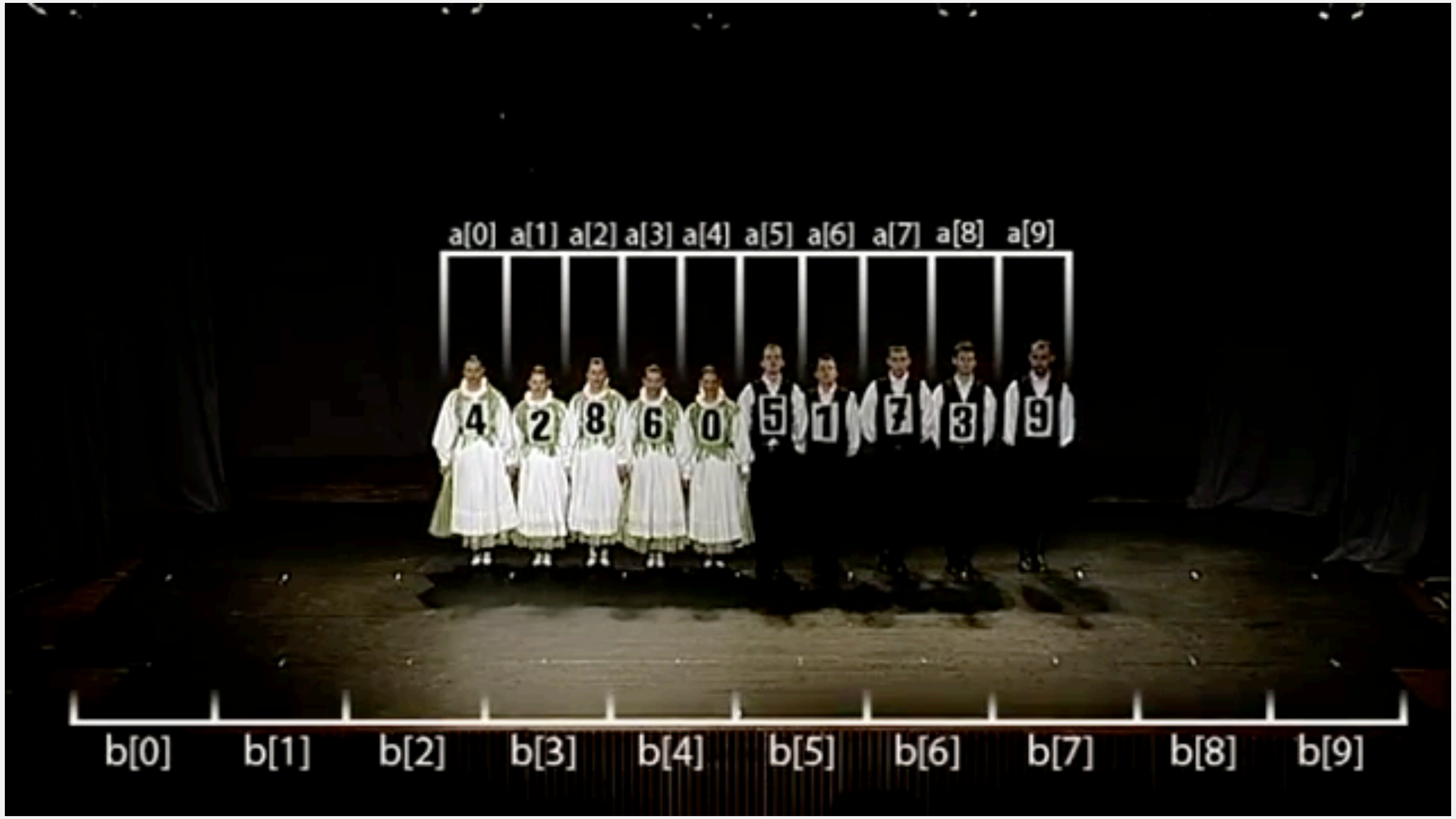
Basic algorithm for sorting objects = mergesort.

- Cutoff to insertion sort = 7.
- Stop-if-already-sorted test.
- Eliminate-the-copy-to-the-auxiliary-array trick.

Arrays.sort(a)

http://www.java2s.com/Open-Source/Java/6.0-JDK-Modules/j2me/java/util/Arrays.java.html

Mergesort: Transylvanian-Saxon folk dance



Reason it is slow: excessive data movement.

https://www.youtube.com/watch?v=XaqR3G_NVoo

Bottom-up MergeSort
(non-recursive)

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, ...

	a[i]															
	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
sz = 1																
`merge(a, aux, 0, 0, 1)`	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 2, 2, 3)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 4, 4, 5)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 6, 6, 7)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 8, 8, 9)`	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
`merge(a, aux, 10, 10, 11)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 12, 12, 13)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 14, 14, 15)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																
`merge(a, aux, 0, 1, 3)`	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
`merge(a, aux, 4, 5, 7)`	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
`merge(a, aux, 8, 9, 11)`	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
`merge(a, aux, 12, 13, 15)`	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																
`merge(a, aux, 0, 3, 7)`	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
`merge(a, aux, 8, 11, 15)`	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 8																
`merge(a, aux, 0, 7, 15)`	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Bottom-up mergesort: Java implementation

```
public class MergeBU
{
   private static void merge(...)
   {  /* as before */  }

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      Comparable[] aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

but about 10% slower than recursive, top-down mergesort on typical systems

Bottom line. Simple and non-recursive version of mergesort.

Timsort

Idea. Exploit pre-existing order by identifying naturally-occurring runs.

- Natural mergesort.
- Use binary insertion sort to make initial runs (if needed).
- A few more clever optimizations.

Tim Peters

Intro

This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than lg(N!) comparisons needed, and as few as N-1), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.

In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs "intelligently". Everything else is complication for speed, and some hard-won measure of memory efficiency.

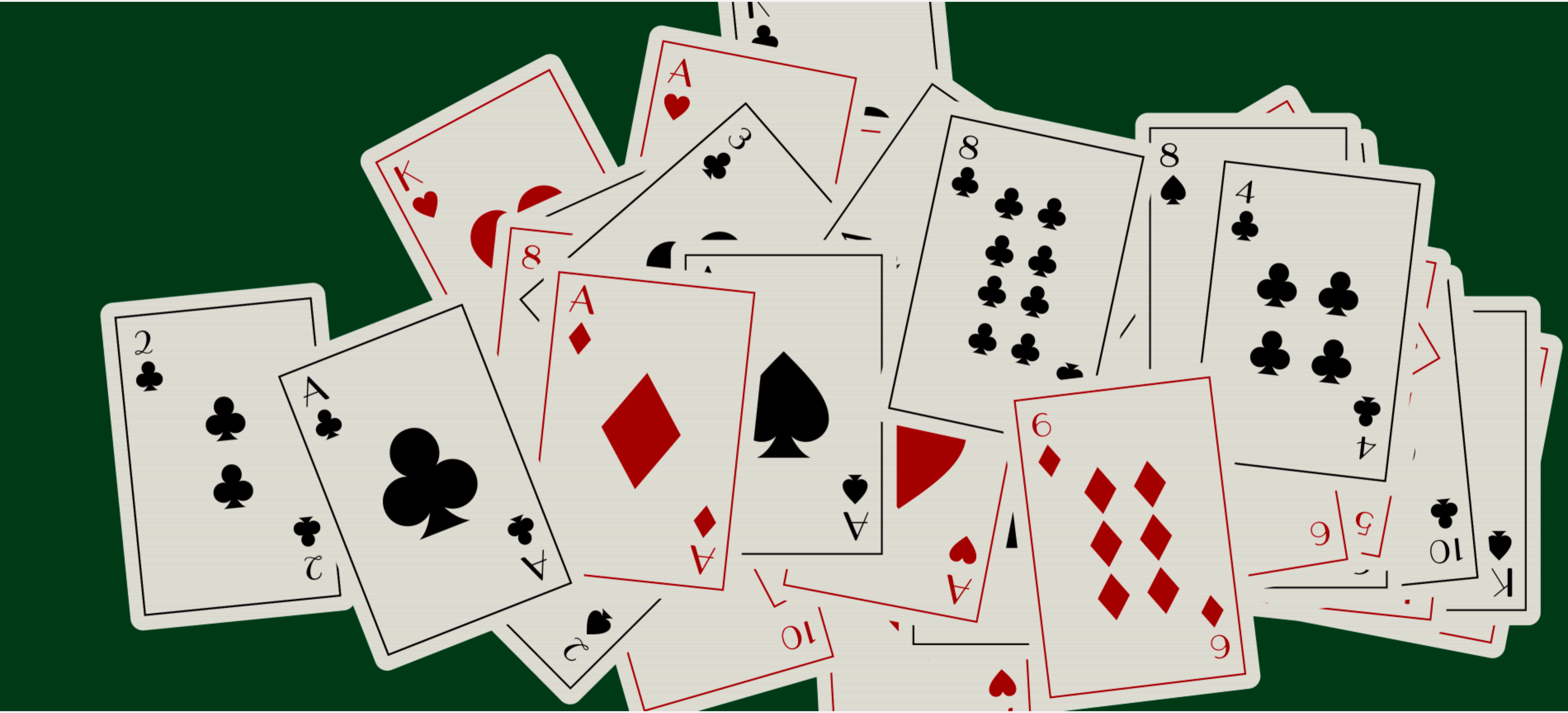
...

Consequence. Linear time on many arrays with pre-existing order.

Now widely used. Python, Java 7, GNU Octave, Android, ….

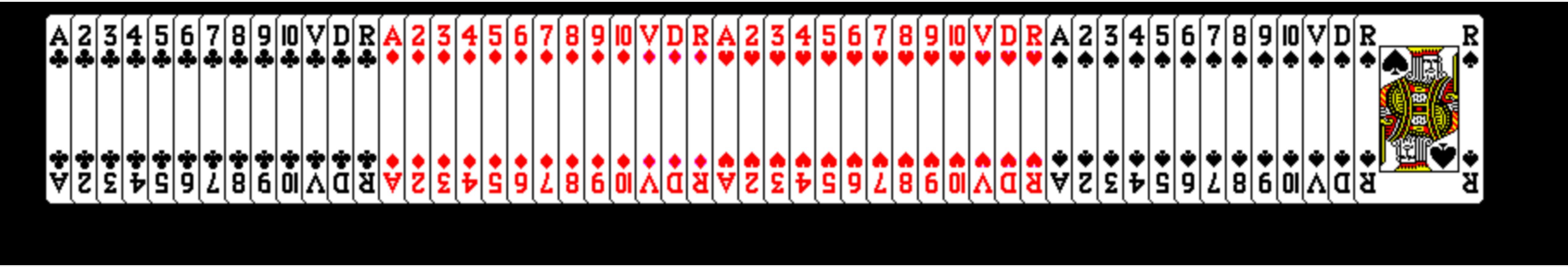
MergeSort Recap

Human Merge Sort



A Perfect Shuffle Aside

If you start with an ordered deck, how many "perfect shuffles" do you need to perform to arrive back with an ordered deck?



https://blogs.sas.com/content/iml/2018/09/24/perfect-riffle-shuffles.html

Animation: https://jdgmiles.github.io/Riffle-Shuffles/

Sorting summary

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	N exchanges
insertion	✔	✔	N	$\frac{1}{4} N^2$	$\frac{1}{2} N^2$	use for small N or partially ordered
shell	✔		$N \log_3 N$	?	$c\, N^{3/2}$	tight code; subquadratic
merge		✔	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
?	✔	✔	N	$N \lg N$	$N \lg N$	holy sorting grail

MergeSort: summary

Comparison sort?	Comparison
Time Complexity	O(N log N)
Space Complexity	Out of place
Internal or External?	External
Recursive / Non-recursive?	Recursive
Stable	Yes, stable