Announcements

I have changed the homework assignments to be due on Thursdays rather than Tuesdays, to make better use of office hours. All HWs are still due before class at 10:30am.

HW3 will be released today. Reminder: DO NOT SHARE CODE OR DISCUSS YOUR IMPLEMENTATION. Do not publish or look online.

Grades for HW1 have been returned. Talk to your recitation TAs if you have questions about the rubric or about the re-grading policy.

Be sure to do all of the readings. They are listed on the Lectures page of the CIS 121 web site. The homework and the exams assume that you have done the assigned readings in addition to attending the lectures.

I will be traveling on Tuesday. The TAs have volunteered to teach a course focusing on reviewing HW1 and HW2, and on Big O and recurrences.

http://algs4.cs.princeton.edu

2.2 MERGESORT

- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *divide-and-conquer*

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. [today]

...

Quicksort. [next lecture]

...

http://algs4.cs.princeton.edu

2.2 Mergesort

- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *divide-and-conquer*

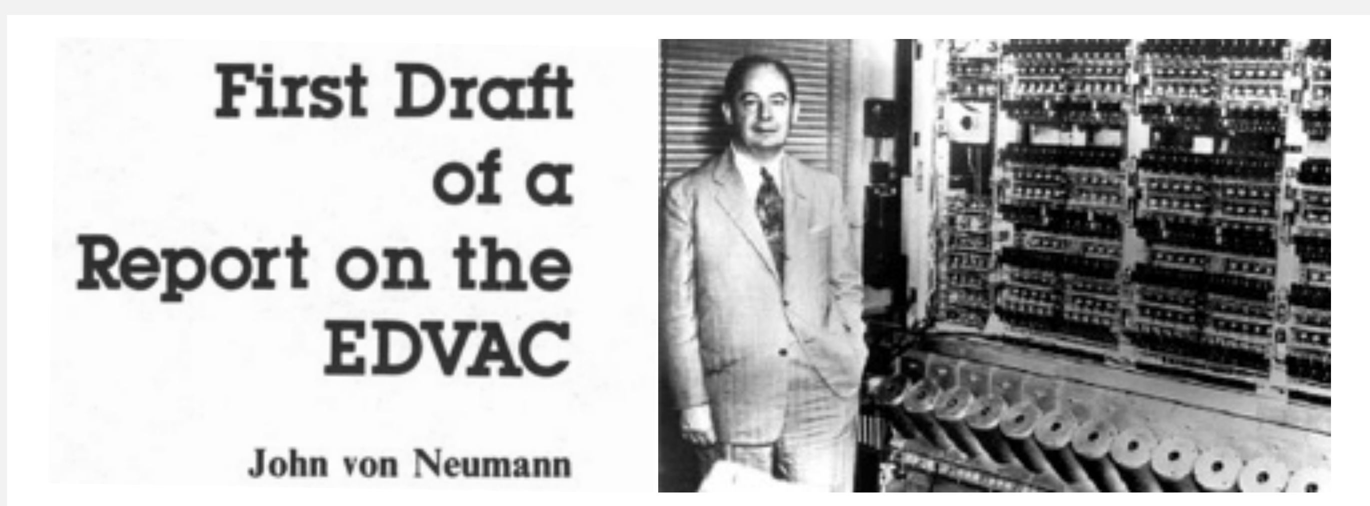
Mergesort

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



Abstract in-place merge 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]`.

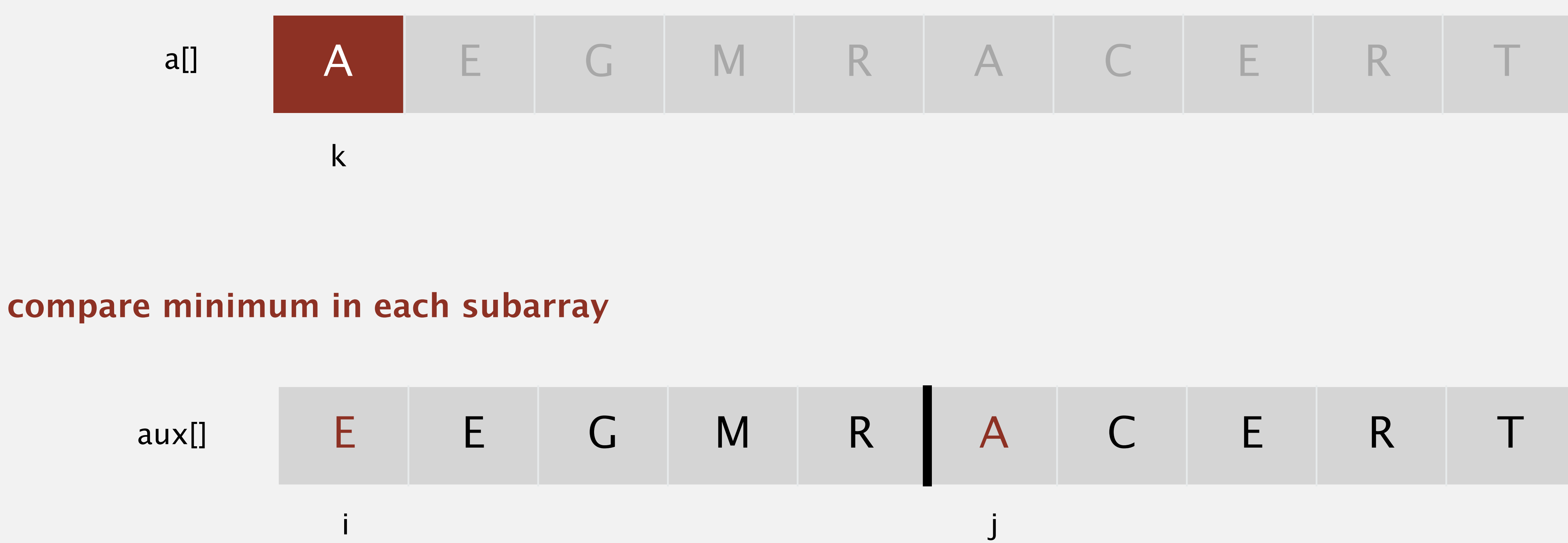
a[]	E	E	G	M	R	A	C	E	R	T
	k									

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]`.



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]`.

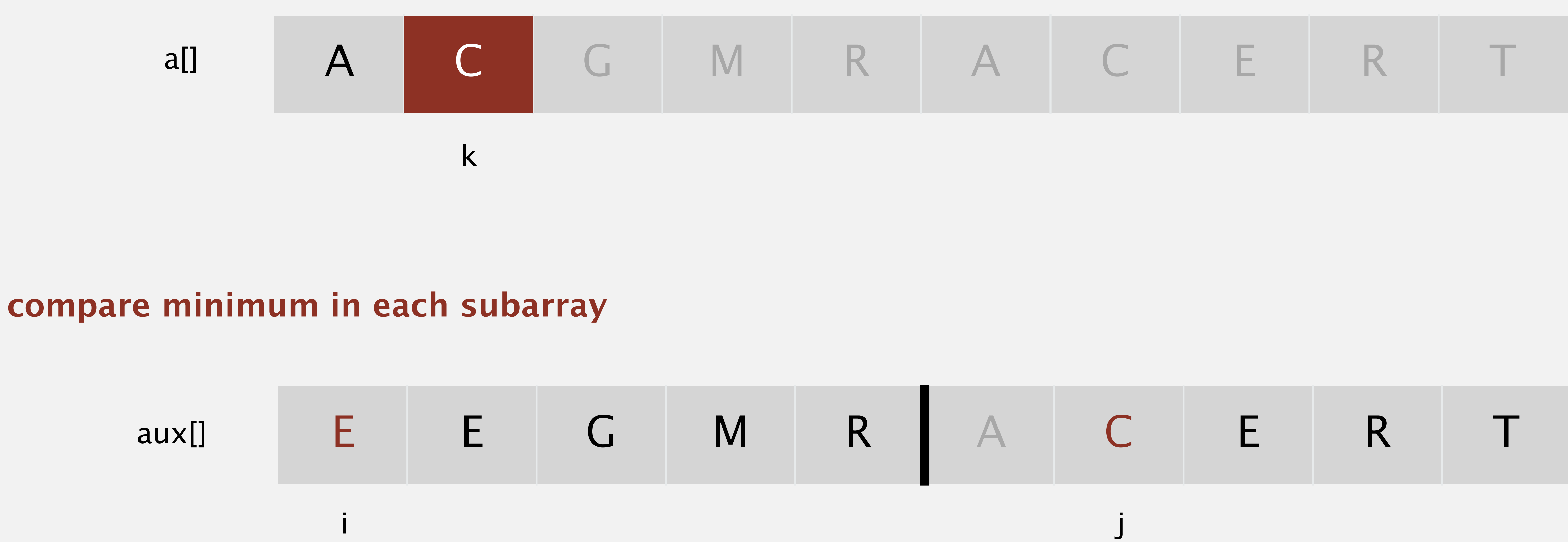
a[]	A	E	G	M	R	A	C	E	R	T
		k								

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]`.



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]`.

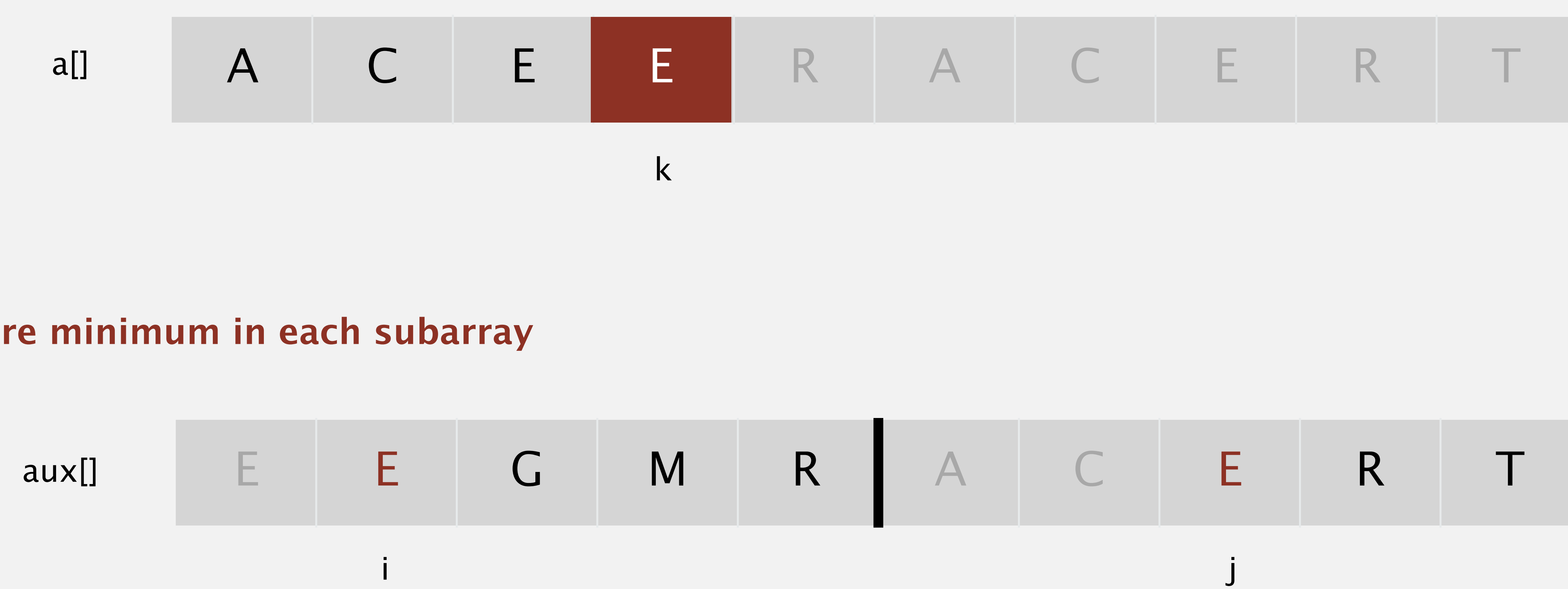
a[]	A	C	E	M	R	A	C	E	R	T
				k						

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]`.



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]`.

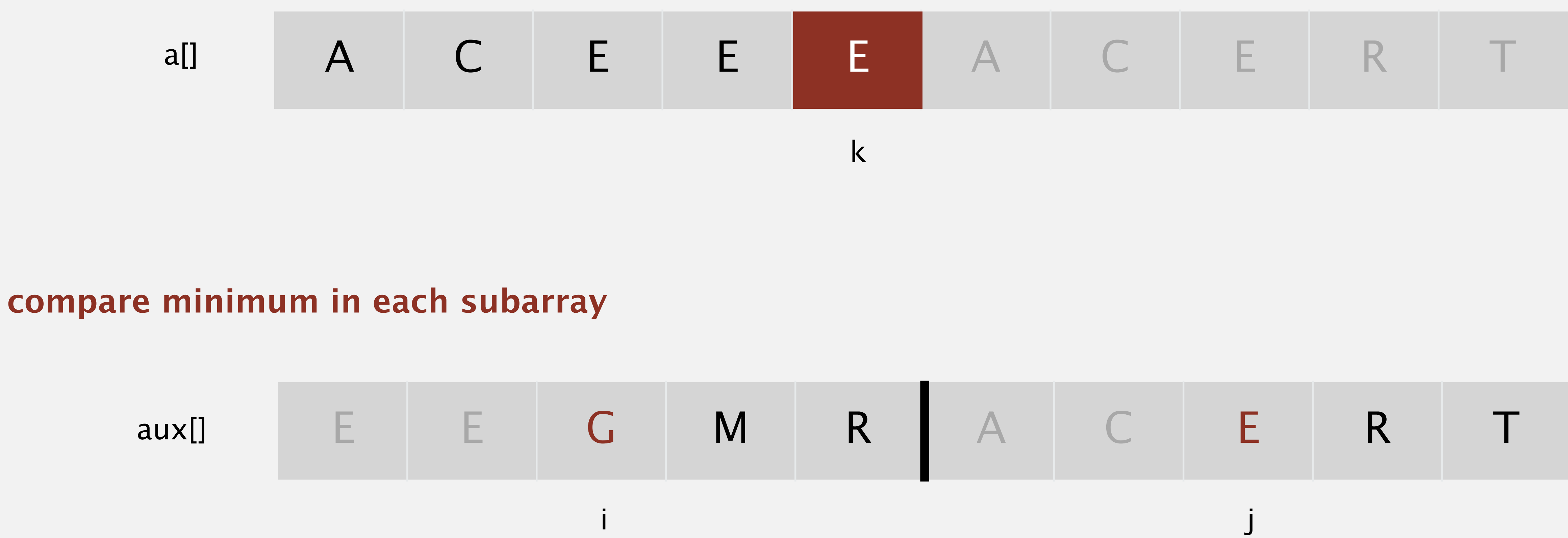
a[]	A	C	E	E	R	A	C	E	R	T
					k					

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]`.



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]`.

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]`.

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]`.

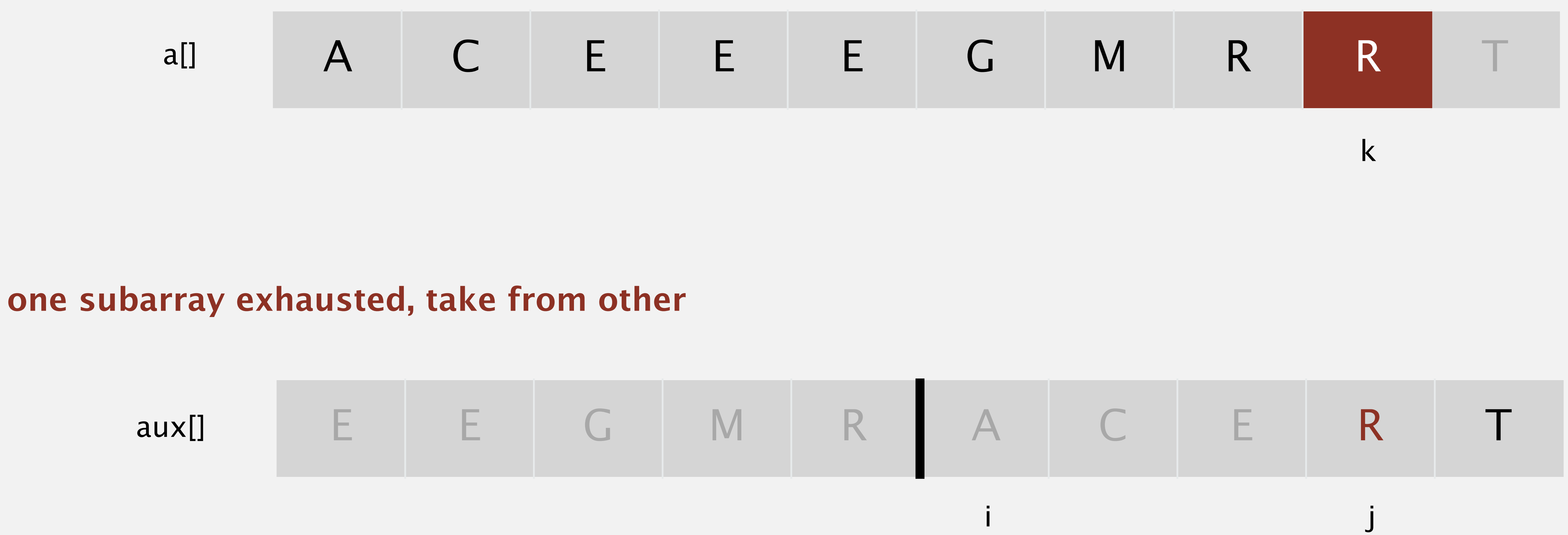
a[]	A	C	E	E	E	G	M	R	R	T
									k	

one subarray exhausted, take from other

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

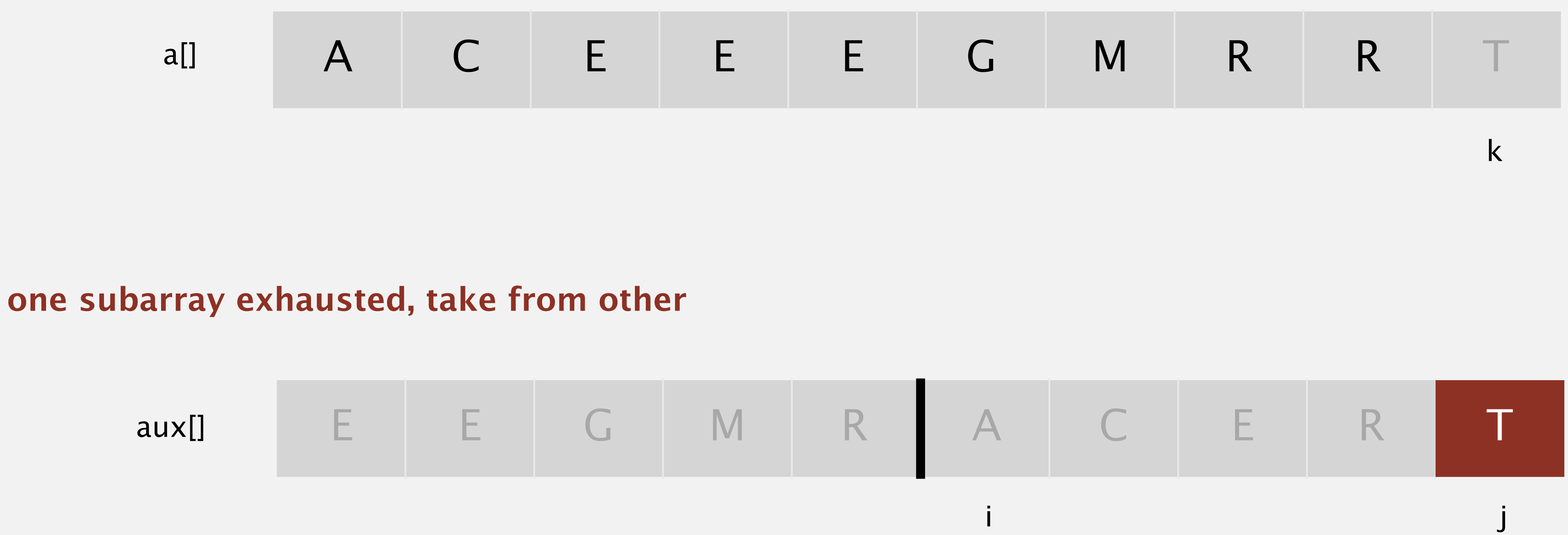
Merging demo

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]`.



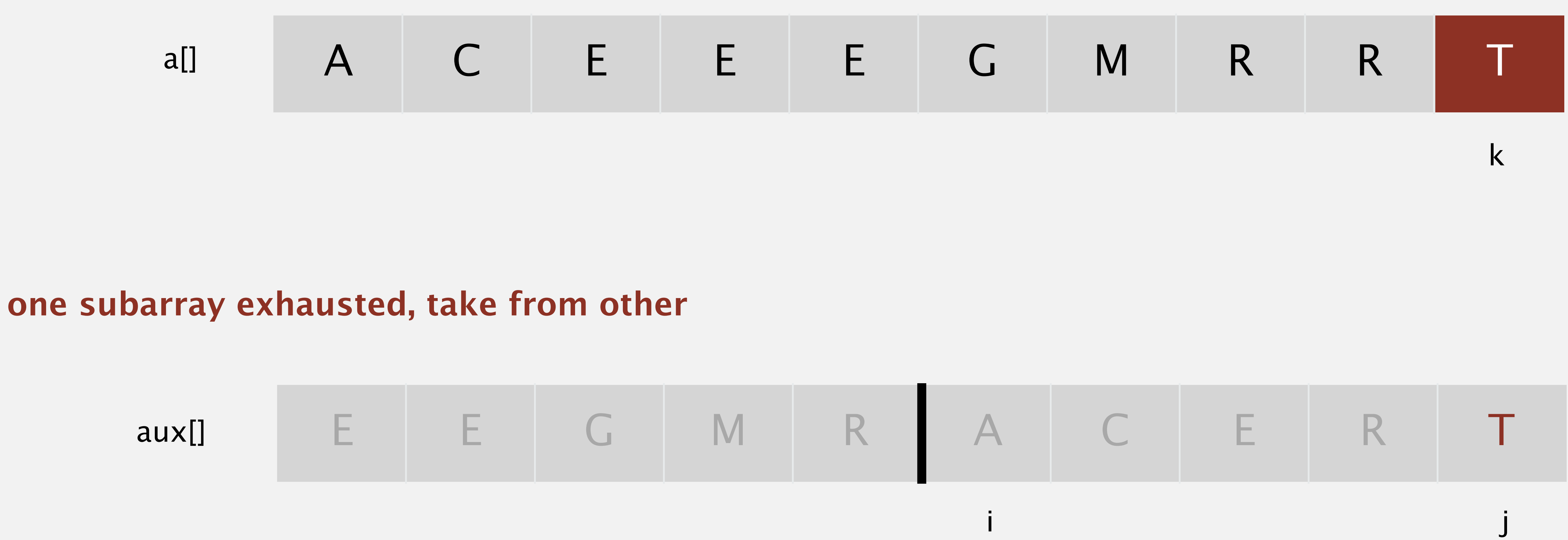
Merging demo

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]`.



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

Abstract in-place merge 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

Mergesort: Transylvanian-Saxon folk dance

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

Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{

   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }
}
```

copy

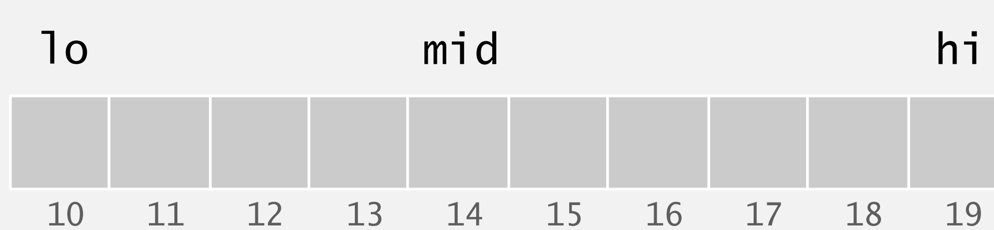
merge

	lo			i	mid			j		hi
aux[]	A	G	L	O	R	H	I	M	S	T

						k				
a[]	A	G	H	I	L	M				

Mergesort: Java implementation

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

   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);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort(Comparable[] a)
   {
      Comparable[] aux = new Comparable[a.length];
      sort(a, aux, 0, a.length - 1);
   }
}
```



Mergesort: trace

	a[]															
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
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 quiz 2

Which of the following subarray lengths will occur when running mergesort on an array of length 12?

A. { 1, 2, 3, 4, 6, 8, 12 }

B. { 1, 2, 3, 6, 12 }

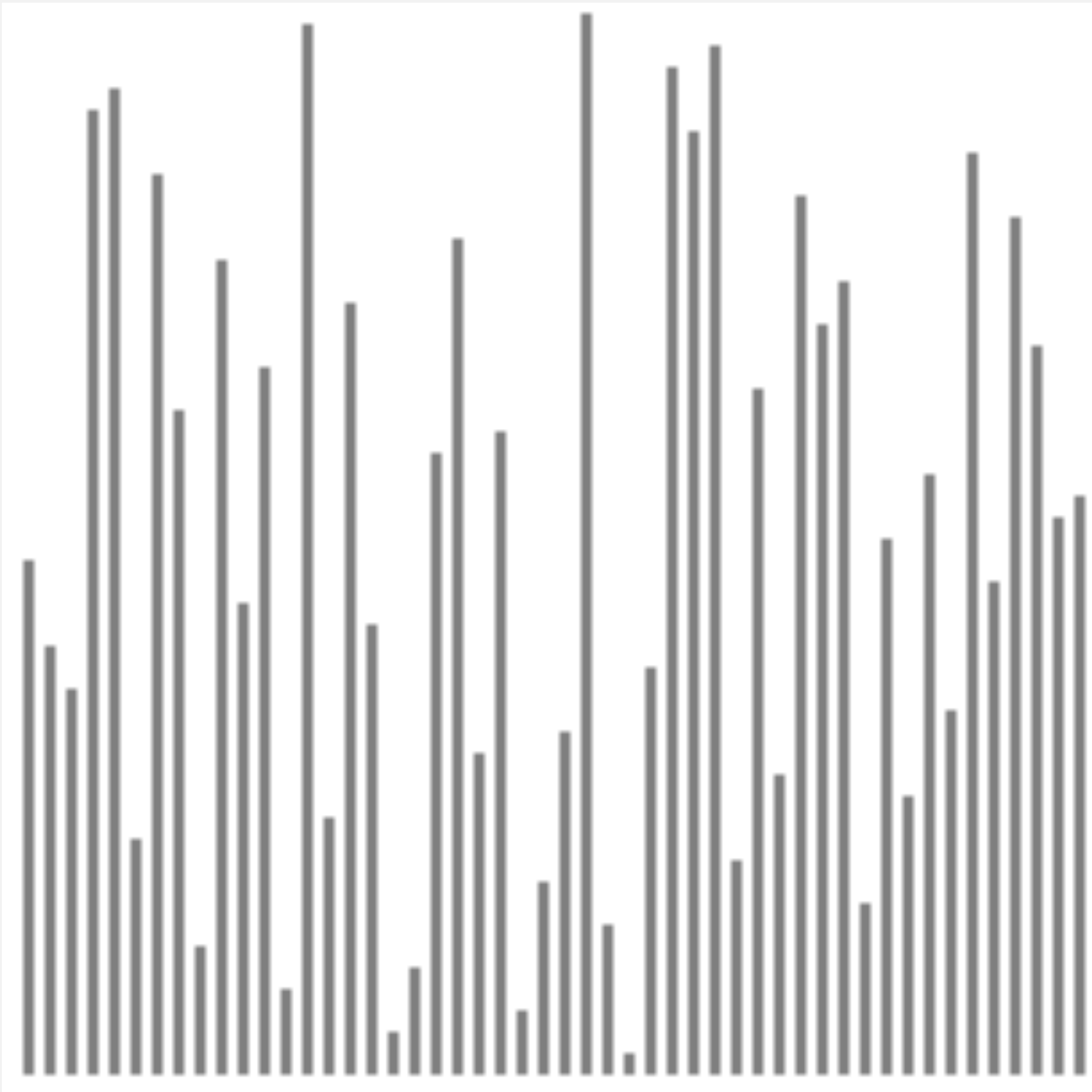
C. { 1, 2, 4, 8, 12 }

D. { 1, 3, 6, 9, 12 }

E. *I don't know.*

Mergesort: animation

50 random items

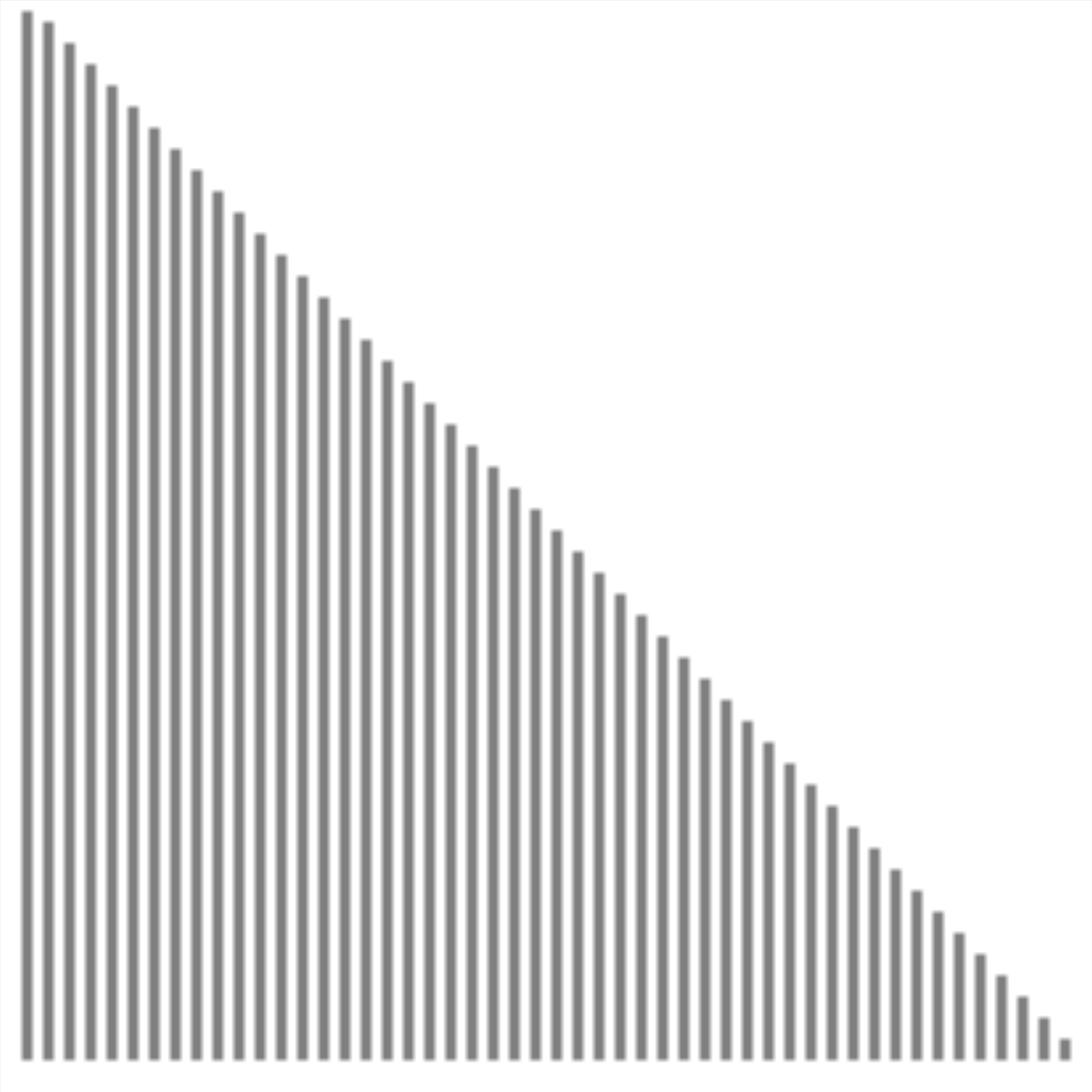


http://www.sorting-algorithms.com/merge-sort

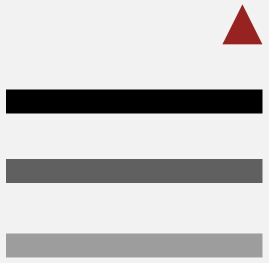
algorithm position
in order
current subarray
not in order

Mergesort: animation

50 reverse-sorted items



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



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 analysis: number of compares

Proposition. Mergesort uses $\leq N \lg N$ compares to sort an array of length N.

Pf sketch. The number of compares $C(N)$ to mergesort an array of length N satisfies the recurrence:

$$C(N) \leq \underbrace{C(\lceil N/2 \rceil)}_{\text{left half}} + \underbrace{C(\lfloor N/2 \rfloor)}_{\text{right half}} + \underbrace{N - 1}_{\text{merge}} \quad \text{for } N > 1, \text{ with } C(1) = 0.$$

We solve this simpler recurrence, and assume N is a power of 2:

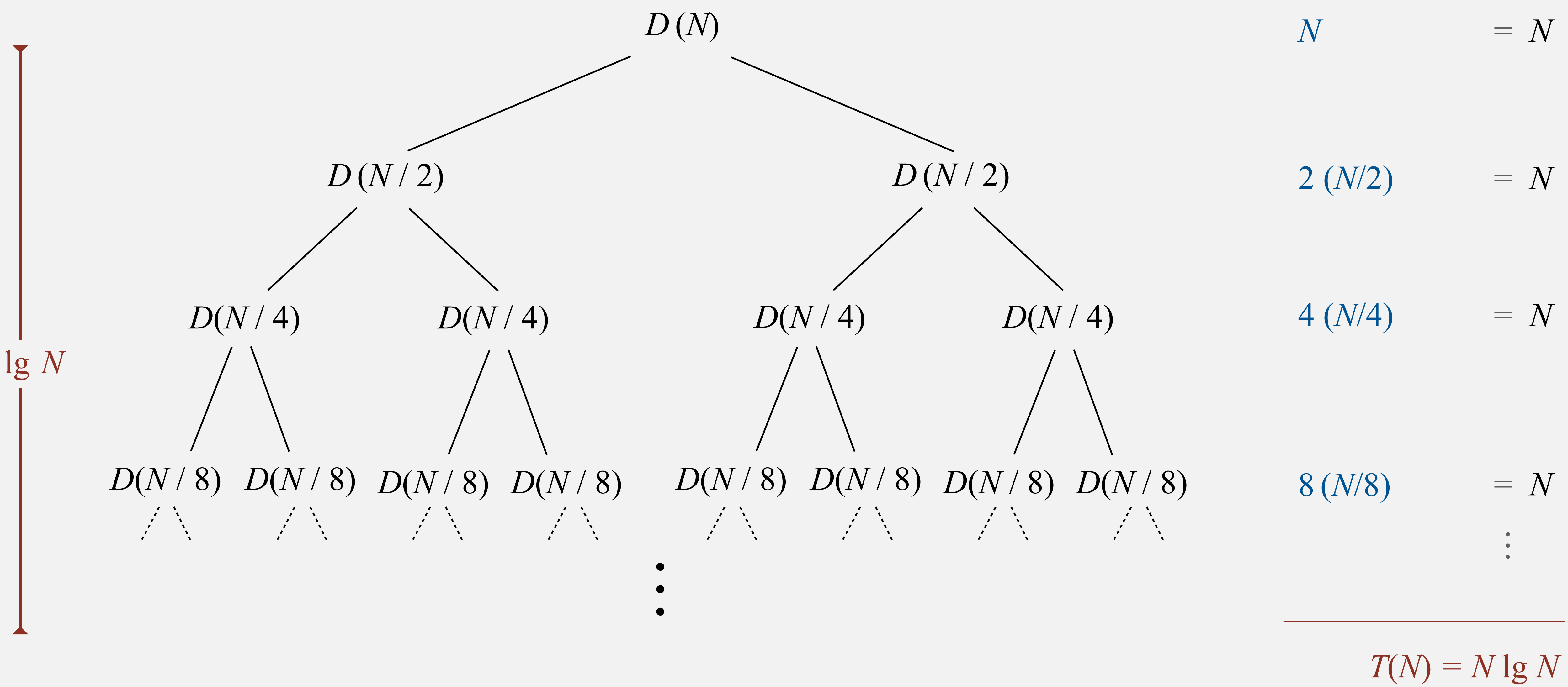
result holds for all N (analysis cleaner in this case)

$$D(N) = 2\,D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0.$$

Divide-and-conquer recurrence

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf by picture. [assuming N is a power of 2]



Mergesort analysis: number of array accesses

Proposition. Mergesort uses $\le 6 N \lg N$ array accesses to sort an array of length N.

Pf sketch. The number of array accesses $A(N)$ satisfies the recurrence:

$$A(N) \le A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N \text{ for } N > 1, \text{ with } A(1) = 0.$$

Key point. Any algorithm with the following structure takes $N \log N$ time:

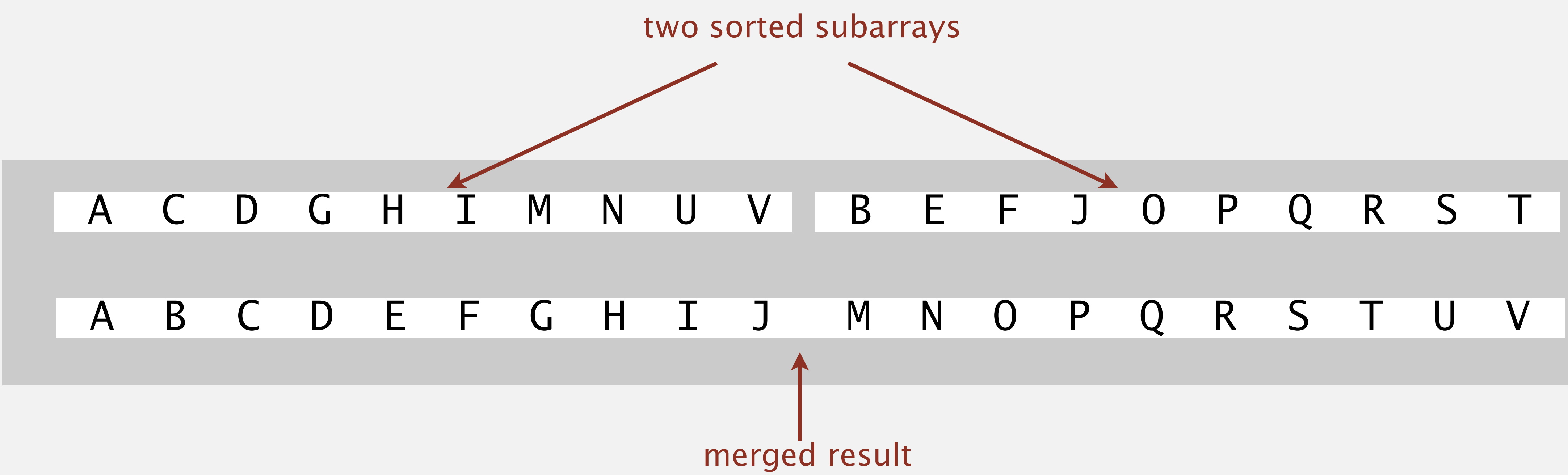
```
public static void f(int N)
{
    if (N == 0) return;
    f(N/2);          solve two problems
    f(N/2);          of half the size
    linear(N);       do a linear amount of work
}
```

Notable algorithms. FFT, hidden-line removal, Kendall-tau distance, …

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.



Def. A sorting algorithm is in-place if it uses $\leq c \log N$ extra memory.

Ex. Insertion sort, selection sort, shellsort.

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].

Stability: mergesort

Proposition. Mergesort is stable.

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

   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);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort(Comparable[] a)
   {  /* as before */  }
}
```

Pf. Suffices to verify that merge operation is stable.

Stability: mergesort

Proposition. Merge operation is stable.

```
private static void merge(...)
{
   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }
}
```

0	1	2	3	4		5	6	7	8	9	10
A_1	A_2	A_3	B	D		A_4	A_5	C	E	F	G

Pf. Takes from left subarray if equal keys.

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: 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);
}
```

Mergesort: practical improvements

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.

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)          aux[k] = a[j++];
      else if (j > hi)           aux[k] = a[i++];
      else if (less(a[j], a[i])) aux[k] = a[j++];
      else                       aux[k] = a[i++];
   }
}

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

merge from a[] to aux[]

assumes aux[] is initialize to a[] once, before recursive calls

switch roles of aux[] and a[]

Java 6 system sort

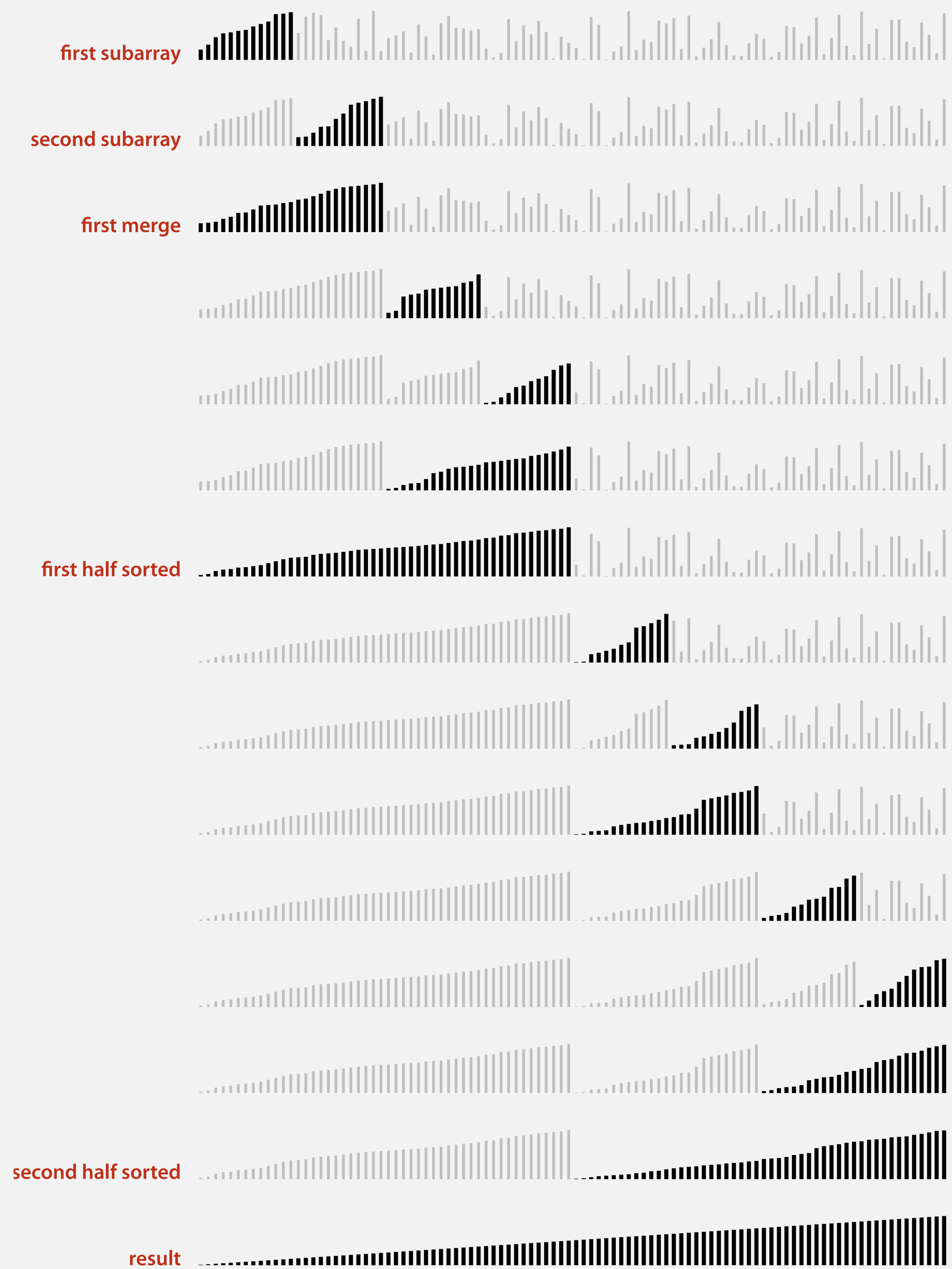
Basic algorithm for sorting objects = mergesort.

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

Arrays.sort(a)

http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/tip/src/share/classes/java/util/Arrays.java

Mergesort with cutoff to insertion sort: visualization



http://algs4.cs.princeton.edu

2.2 Mergesort

- mergesort
- **bottom-up mergesort**
- sorting complexity
- divide-and-conquer

Bottom-up mergesort

Basic plan.

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

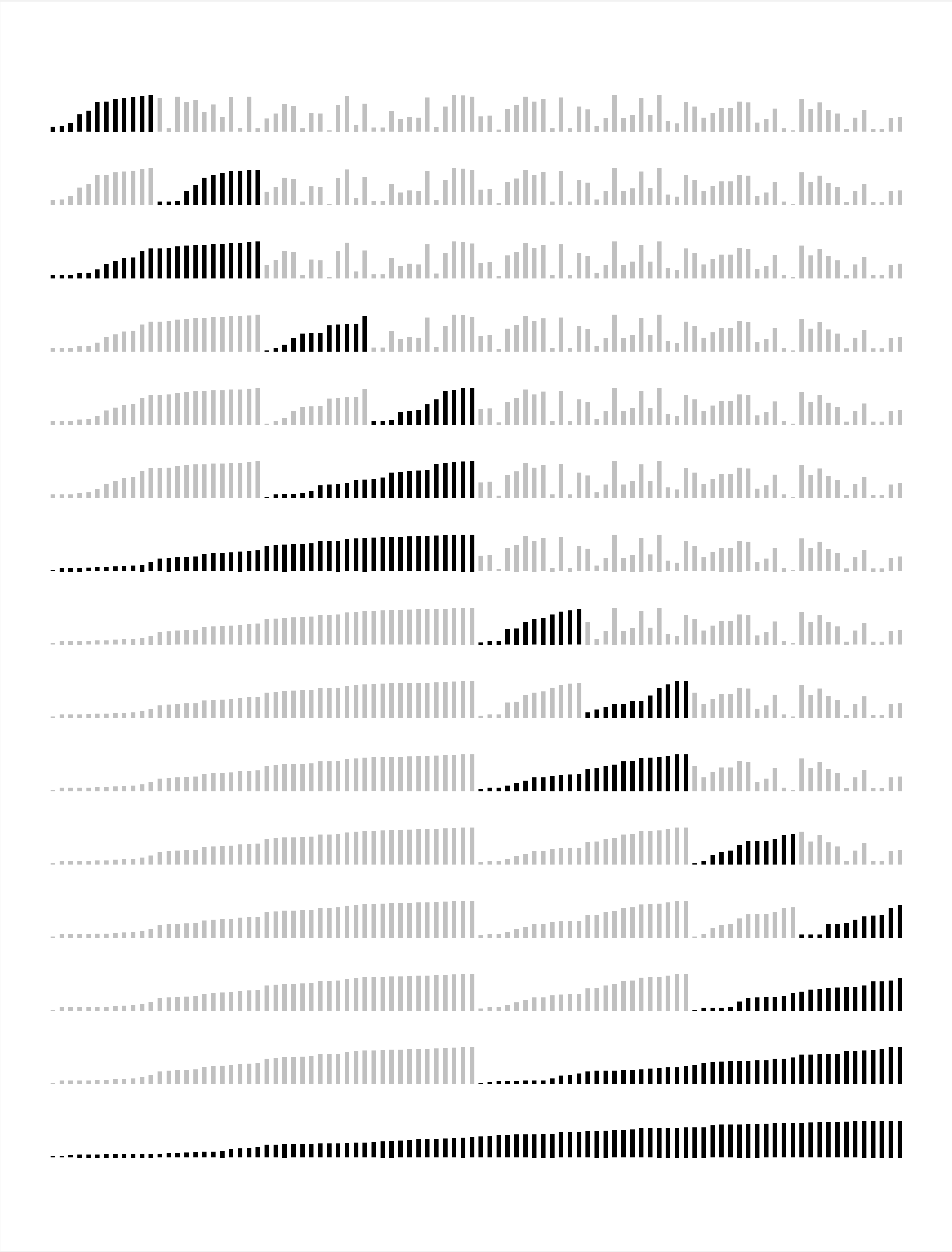
	a[i] 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																
`merge(a, aux, 0, 0, 1)`	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 2, 2, 3)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 4, 4, 5)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 6, 6, 7)`	E	M	G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

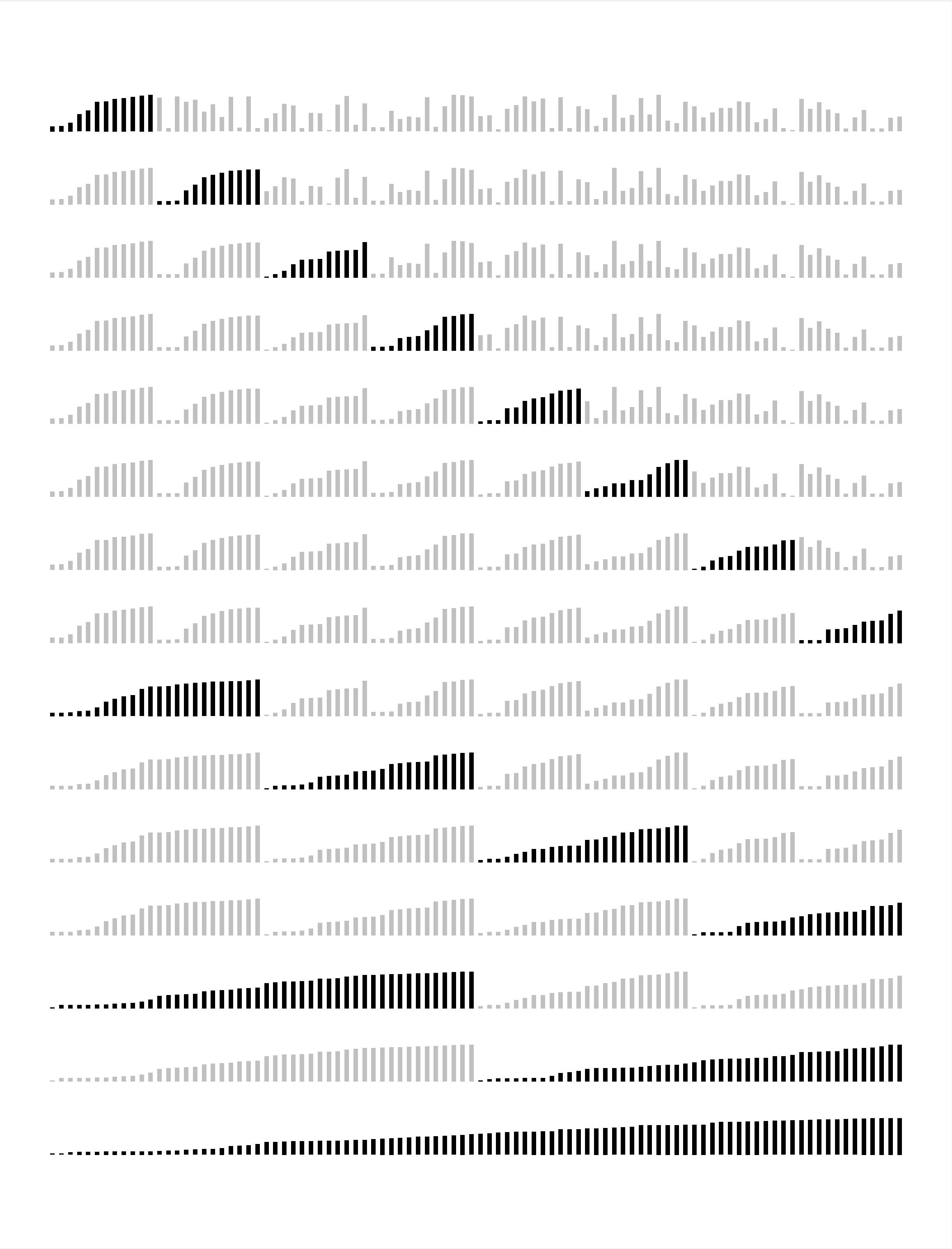
```
public class MergeBU
{
   private static void merge(...)
   {  /* as before */  }

   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));
   }
}
```

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

Mergesort: visualizations



top-down mergesort (cutoff = 12)



bottom-up mergesort (cutoff = 12)

Natural mergesort

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

input

1	5	10	16	3	4	23	9	13	2	7	8	12	14

first run

1	5	10	16	3	4	23	9	13	2	7	8	12	14

second run

1	5	10	16	3	4	23	9	13	2	7	8	12	14

merge two runs

1	3	4	5	10	16	23	9	13	2	7	8	12	14

Tradeoff. Fewer passes vs. extra compares per pass to identify runs.

Timsort

- 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, ...

http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/src/share/classes/java/util/Arrays.java

Timsort bug

Envisage: Engineering Virtualized Services

Envisage | About Envisage | Follow Envisage | Dissemination | Log in

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

February 24, 2015 Envisage Written by Stijn de Gouw. $s

Tim Peters developed the Timsort hybrid sorting algorithm in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort) by Joshua Bloch (the designer of Java Collections who also pointed out that most binary search algorithms were broken). TimSort is today used as the default sorting algorithm for Android SDK, Sun's JDK and OpenJDK. Given the popularity of these platforms this means that the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.

http://envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/

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
timsort		✔	N	$N \lg N$	$N \lg N$	improves mergesort when preexisting order
?	✔	✔	N	$N \lg N$	$N \lg N$	holy sorting grail

http://algs4.cs.princeton.edu

2.2 Mergesort

- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *divide-and-conquer*

Complexity of sorting

Computational complexity. Framework to study efficiency of algorithms for solving a particular problem X.

Model of computation. Allowable operations.

Cost model. Operation counts.

Upper bound. Cost guarantee provided by some algorithm for X.

Lower bound. Proven limit on cost guarantee of all algorithms for X.

Optimal algorithm. Algorithm with best possible cost guarantee for X.

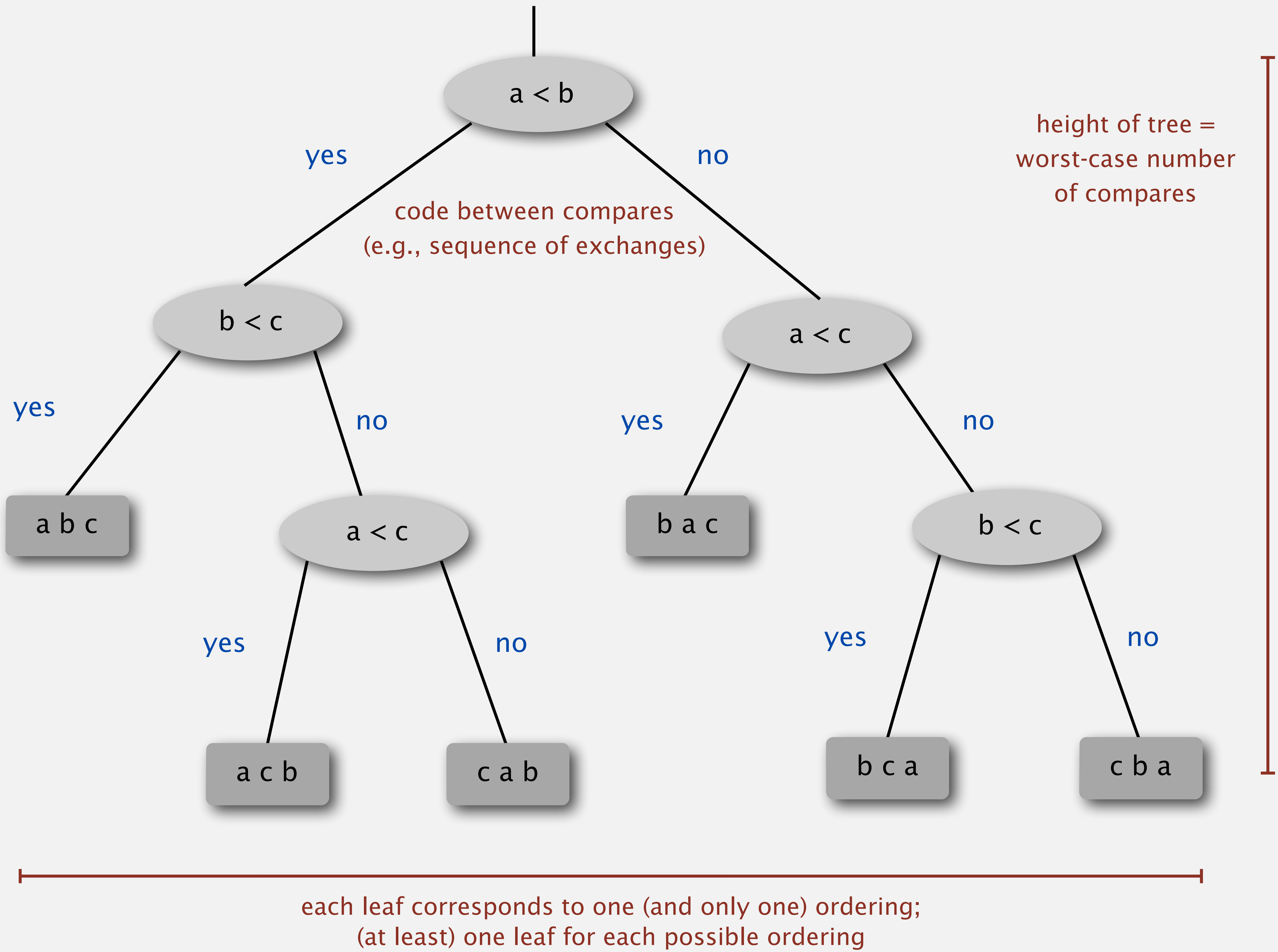
lower bound ~ upper bound

model of computation	*decision tree*
cost model	*# compares*
upper bound	$\sim N \lg N$ *from mergesort*
lower bound	?
optimal algorithm	?

can access information only through compares (e.g., Java `Comparable` framework)

complexity of sorting

Decision tree (for 3 distinct keys a, b, and c)

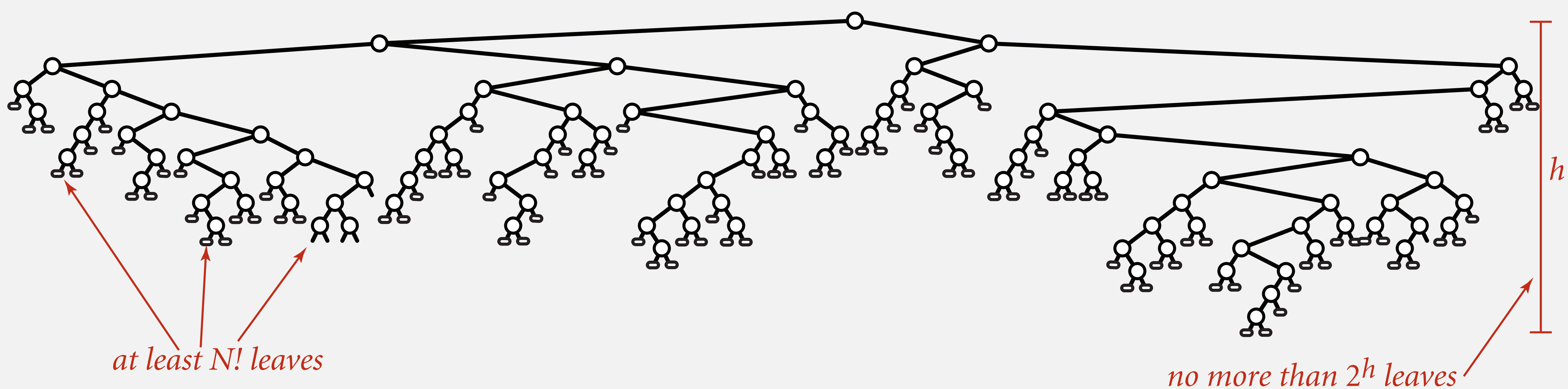


Compare-based lower bound for sorting

Proposition. Any compare-based sorting algorithm must use at least $\lg(N!) \sim N \lg N$ compares in the worst-case.

Pf.

- Assume array consists of N distinct values a_1 through a_N.
- Worst case dictated by **height** h of decision tree.
- Binary tree of height h has at most 2^h leaves.
- $N!$ different orderings $\Rightarrow$ at least $N!$ leaves.



Compare-based lower bound for sorting

Proposition. Any compare-based sorting algorithm must use at least $\lg(N!) \sim N \lg N$ compares in the worst-case.

Pf.

- Assume array consists of N distinct values a_1 through a_N.
- Worst case dictated by height h of decision tree.
- Binary tree of height h has at most 2^h leaves.
- $N!$ different orderings $\Rightarrow$ at least $N!$ leaves.

$$2^h \geq \#\,\text{leaves} \geq N!$$

$$\Rightarrow \; h \geq \lg(N!) \sim N \lg N$$

Stirling's formula

Complexity of sorting

Model of computation. Allowable operations.

Cost model. Operation count(s).

Upper bound. Cost guarantee provided by some algorithm for X.

Lower bound. Proven limit on cost guarantee of all algorithms for X.

Optimal algorithm. Algorithm with best possible cost guarantee for X.

model of computation	*decision tree*
cost model	*# compares*
upper bound	$\sim N \lg N$
lower bound	$\sim N \lg N$
optimal algorithm	*mergesort*

complexity of sorting

First goal of algorithm design: optimal algorithms.

Complexity results in context

Compares? Mergesort is optimal with respect to number compares.

Space? Mergesort is not optimal with respect to space usage.

Lessons. Use theory as a guide.

Ex. Design sorting algorithm that guarantees $\sim \frac{1}{2} N \lg N$ compares?

Ex. Design sorting algorithm that is both time- and space-optimal?

Complexity results in context (continued)

Lower bound may not hold if the algorithm can take advantage of:

- The initial order of the input.
  Ex: insertion sort requires only a linear number of compares on partially-sorted arrays.

- The distribution of key values.
  Ex: 3-way quicksort requires only a linear number of compares on arrays with a constant number of distinct keys. [stay tuned]

- The representation of the keys.
  Ex: radix sorts require no key compares — they access the data via character/digit compares.

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for
Tilde	leading term	$\sim \frac{1}{2} N^2$	$\frac{1}{2} N^2$ $\frac{1}{2} N^2 + 22 N \log N + 3 N$
Big Theta	order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3 N$
Big O	upper bound	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$
Big Omega	lower bound	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$

http://algs4.cs.princeton.edu

1.4 ANALYSIS OF ALGORITHMS

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth classifications*
- ***theory of algorithms***
- *memory*

Types of analyses

Best case. Lower bound on cost.

- Determined by "easiest" input.
- Provides a goal for all inputs.

Worst case. Upper bound on cost.

- Determined by "most difficult" input.
- Provides a guarantee for all inputs.

Average case. Expected cost for random input.

- Need a model for "random" input.
- Provides a way to predict performance.

Ex 1. Array accesses for brute-force 3-SUM.

Best: $\sim \frac{1}{2} N^3$

Average: $\sim \frac{1}{2} N^3$

Worst: $\sim \frac{1}{2} N^3$

Ex 2. Compares for binary search.

Best: ~ 1

Average: $\sim \lg N$

Worst: $\sim \lg N$

Types of analyses

Best case. Lower bound on cost.

Worst case. Upper bound on cost.

Average case. “Expected” cost.

Actual data might not match input model?

- Need to understand input to effectively process it.
- Approach 1: design for the worst case.
- Approach 2: randomize, depend on probabilistic guarantee.

Theory of algorithms

Goals.

- Establish "difficulty" of a problem.
- Develop "optimal" algorithms.

Approach.

- Suppress details in analysis: analyze "to within a constant factor."
- Eliminate variability in input model: focus on the worst case.

Upper bound. Performance guarantee of algorithm for any input.

Lower bound. Proof that no algorithm can do better.

Optimal algorithm. Lower bound = upper bound (to within a constant factor).

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ $\vdots$	classify algorithms
Big O	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ $\vdots$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ $\vdots$	develop lower bounds

Theory of algorithms: example 1

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 1-SUM = "*Is there a* 0 *in the array?* "

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 1-SUM: Look at every array entry.
- Running time of the optimal algorithm for 1-SUM is $O(N)$.

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all N entries (any unexamined one might be 0).
- Running time of the optimal algorithm for 1-SUM is $\Omega(N)$.

Optimal algorithm.

- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is optimal: its running time is $\Theta(N)$.

Theory of algorithms: example 2

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^3)$.

Theory of algorithms: example 2

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.

- Ex. Improved algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^2 \log N)$.

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all N entries to solve 3-SUM.
- Running time of the optimal algorithm for solving 3-SUM is $\Omega(N)$.

Open problems.

- Optimal algorithm for 3-SUM?
- Subquadratic algorithm for 3-SUM?
- Quadratic lower bound for 3-SUM?

Algorithm design approach

Start.

- Develop an algorithm.
- Prove a lower bound.

Gap?

- Lower the upper bound (discover a new algorithm).
- Raise the lower bound (more difficult).

Golden Age of Algorithm Design.

- 1970s-.
- Steadily decreasing upper bounds for many important problems.
- Many known optimal algorithms.

Caveats.

- Overly pessimistic to focus on worst case?
- Need better than "to within a constant factor" to predict performance.

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10\ N^2$	$10\ N^2$ $10\ N^2 + 22\ N \log N$ $10\ N^2 + 2\ N + 37$	provide approximate model
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$½\ N^2$ $10\ N^2$ $5\ N^2 + 22\ N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10\ N^2$ $100\ N$ $22\ N \log N + 3\ N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$½\ N^2$ N^5 $N^3 + 22\ N \log N + 3\ N$	develop lower bounds

Common mistake. Interpreting big-Oh as an approximate model.

This course. Focus on approximate models: use Tilde-notation