# BBM 202 - ALGORITHMS

HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

# MERGESORT

**Acknowledgement:** The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

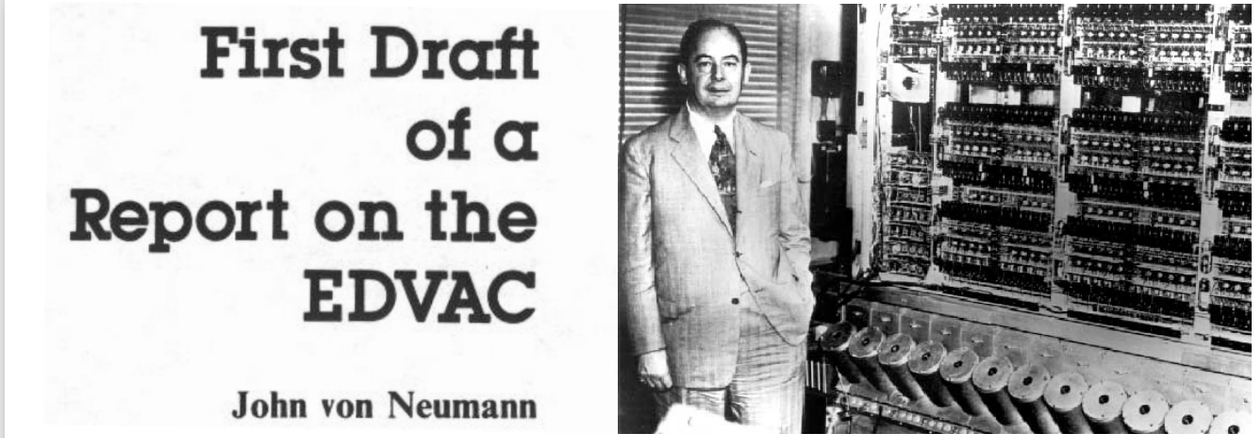
# Mergesort

## Basic plan.

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

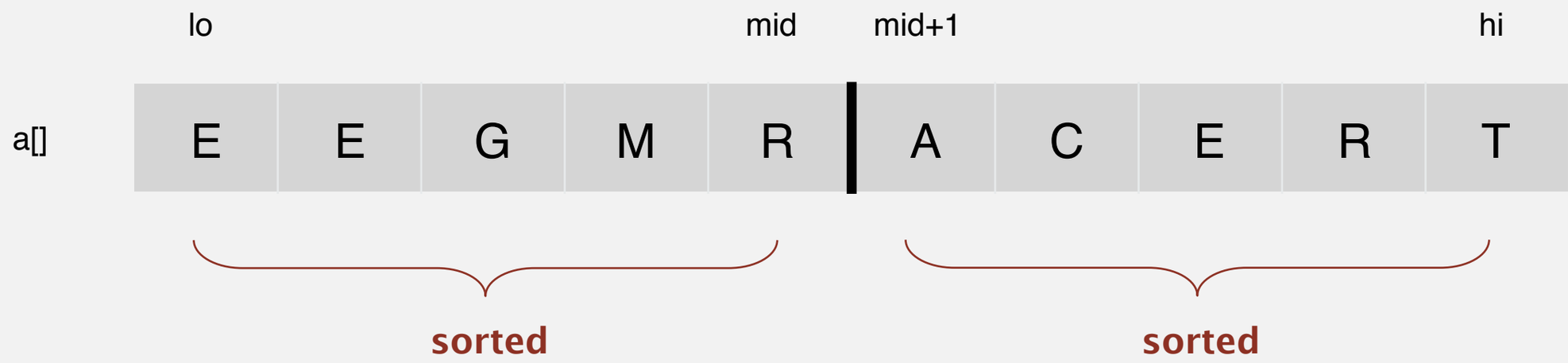
| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input | M | E | R | G | E | S | O | R | T | E | X | A | M | P | L | E |
| sort left half | E | E | G | M | O | R | R | S | T | E | X | A | M | P | L | E |
| sort right half | E | E | G | M | O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

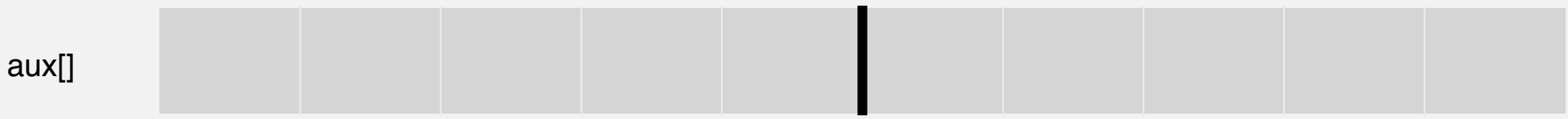
Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



# Abstract in-place merge

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]`.


## copy to auxiliary array

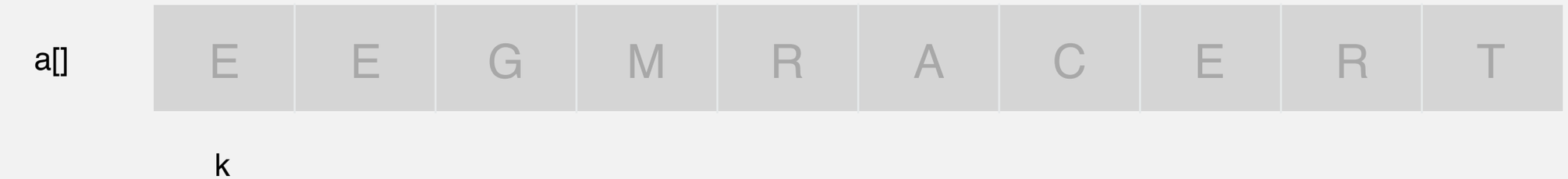


# Abstract in-place merge

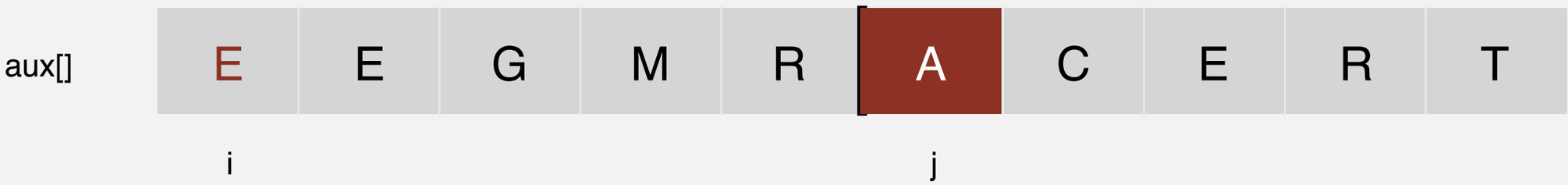
Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.


# Abstract in-place merge

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]`.

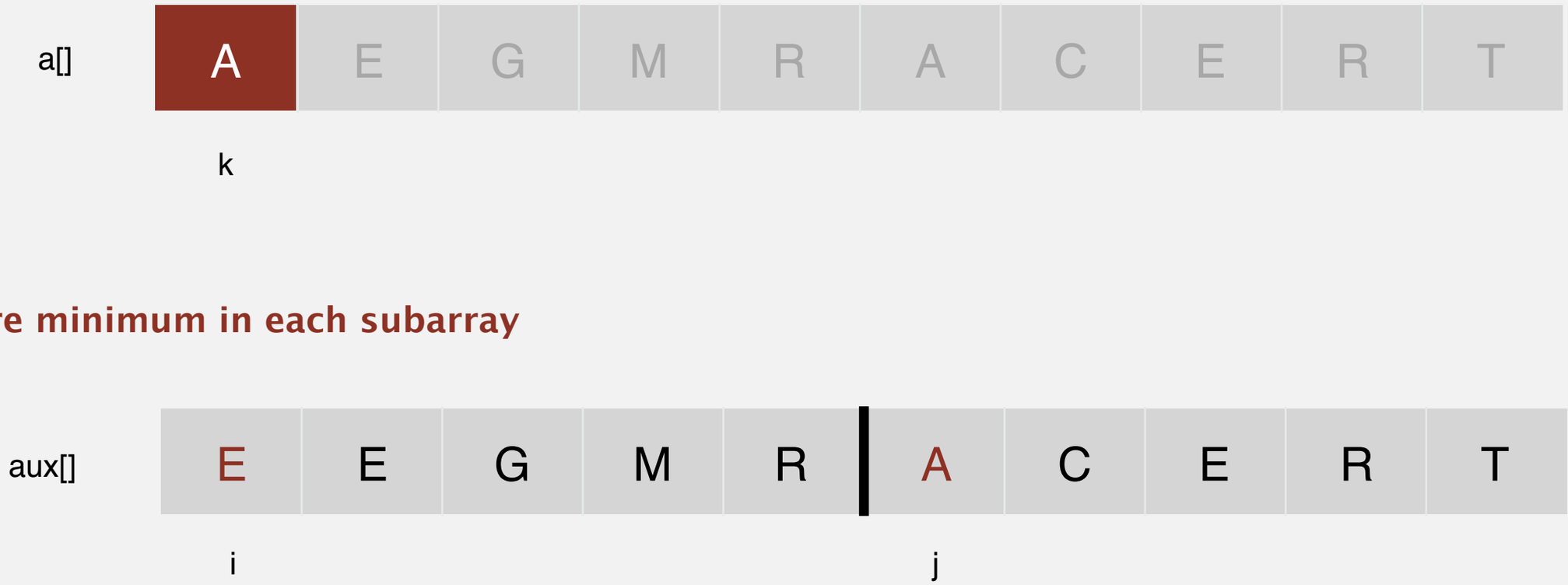



**compare minimum in each subarray**



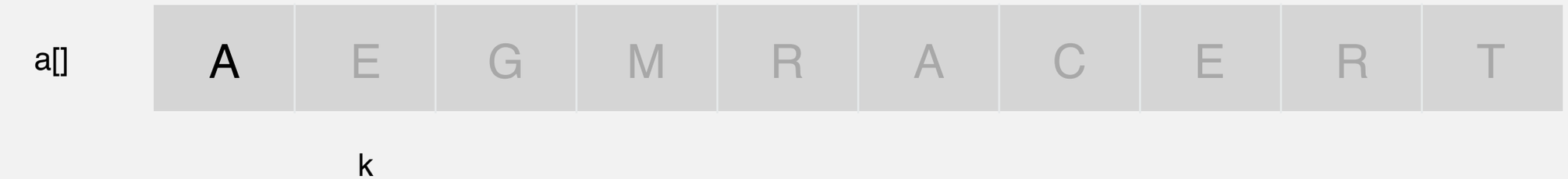
# Abstract in-place merge

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]`.



# Abstract in-place merge

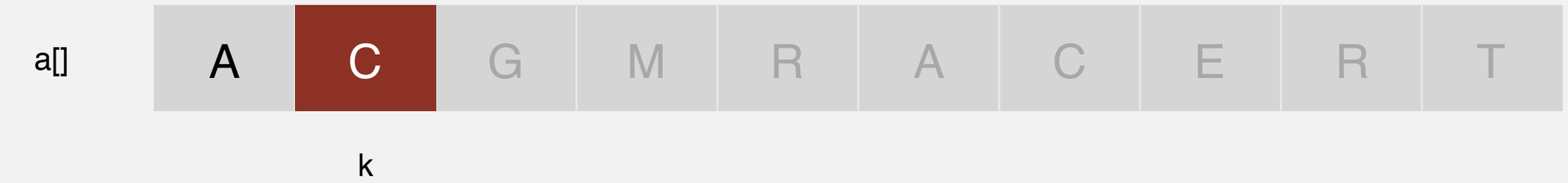
Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.




**compare minimum in each subarray**

# Abstract in-place merge

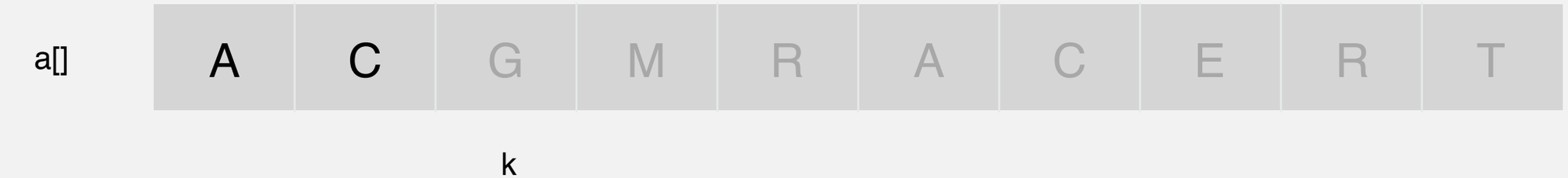
Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.




**compare minimum in each subarray**

# Abstract in-place merge

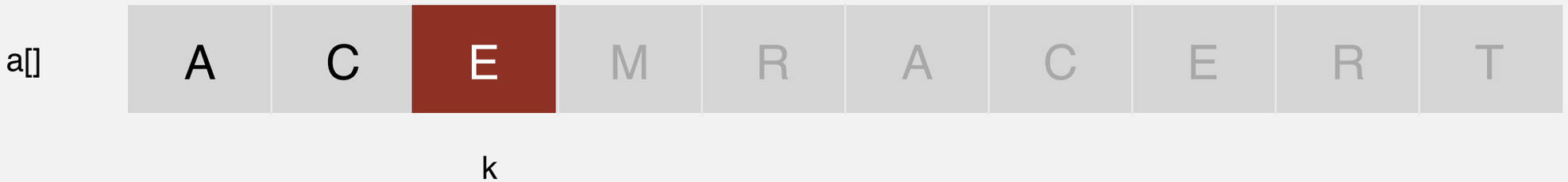
Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.




**compare minimum in each subarray**

# Abstract in-place merge

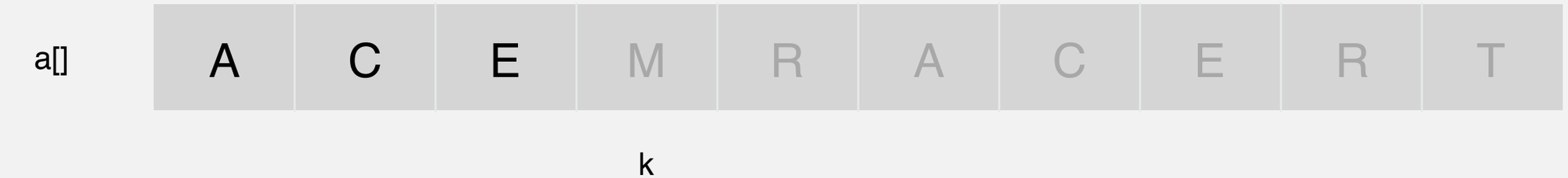
Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.




**compare minimum in each subarray**

# Abstract in-place merge

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]`.




**compare minimum in each subarray**

# Abstract in-place merge

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]`.


k

**compare minimum in each subarray**

# Abstract in-place merge

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]`.


**compare minimum in each subarray**

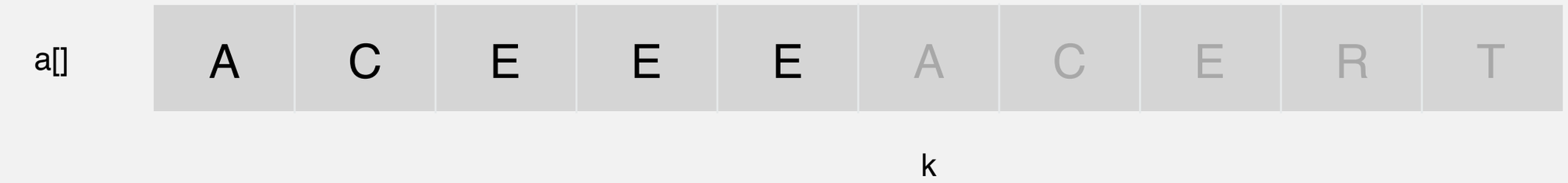
# Abstract in-place merge

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]`.


**compare minimum in each subarray**

# Abstract in-place merge

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]`.




**compare minimum in each subarray**

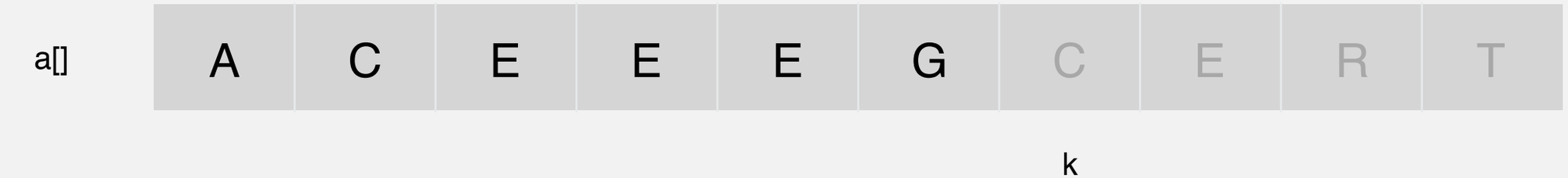
# Abstract in-place merge

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]`.


**compare minimum in each subarray**

# Abstract in-place merge

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]`.




**compare minimum in each subarray**

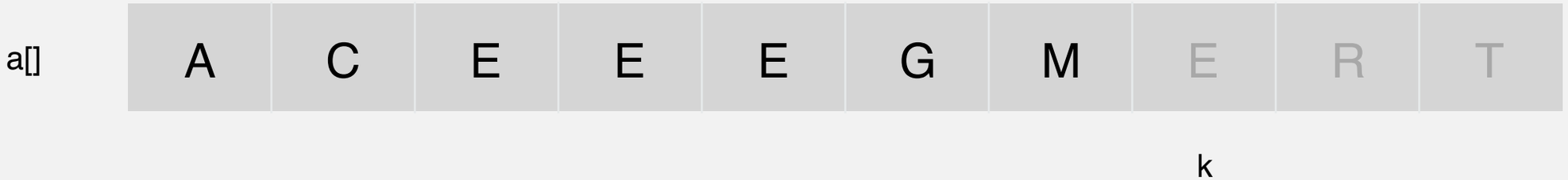
# Abstract in-place merge

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]`.


**compare minimum in each subarray**

# Abstract in-place merge

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]`.




**compare minimum in each subarray**

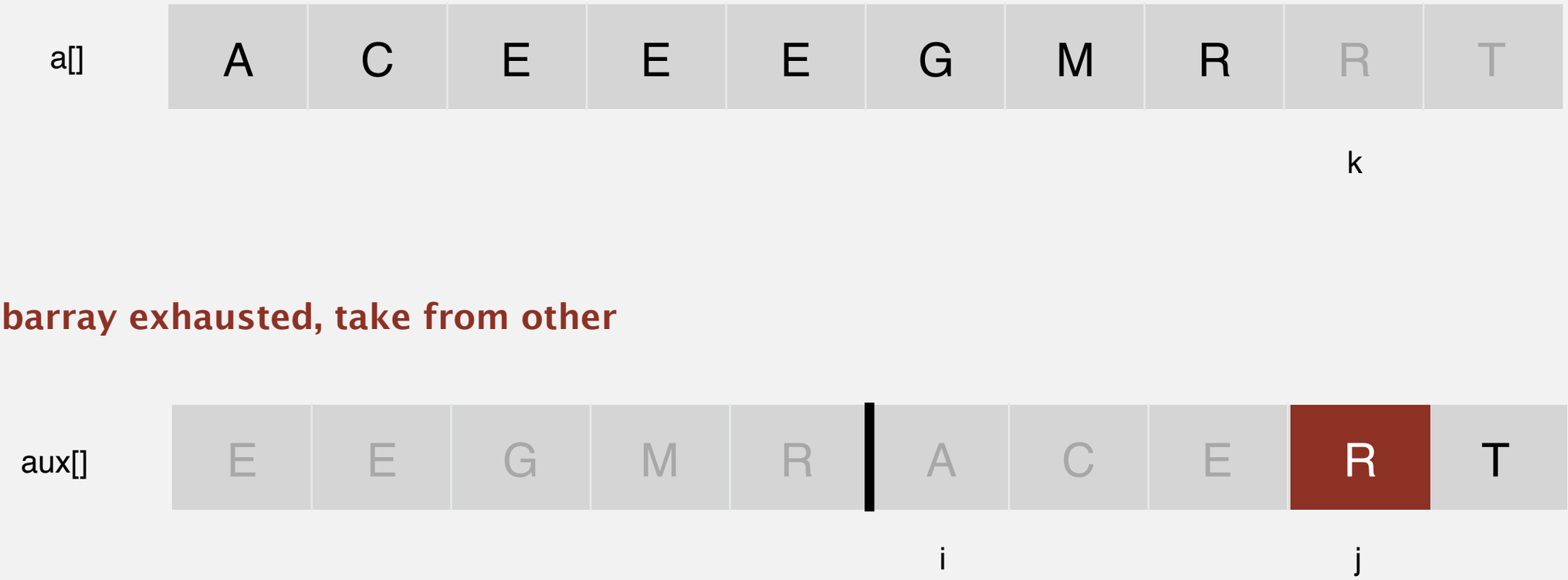
# Abstract in-place merge

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]`.


**compare minimum in each subarray**

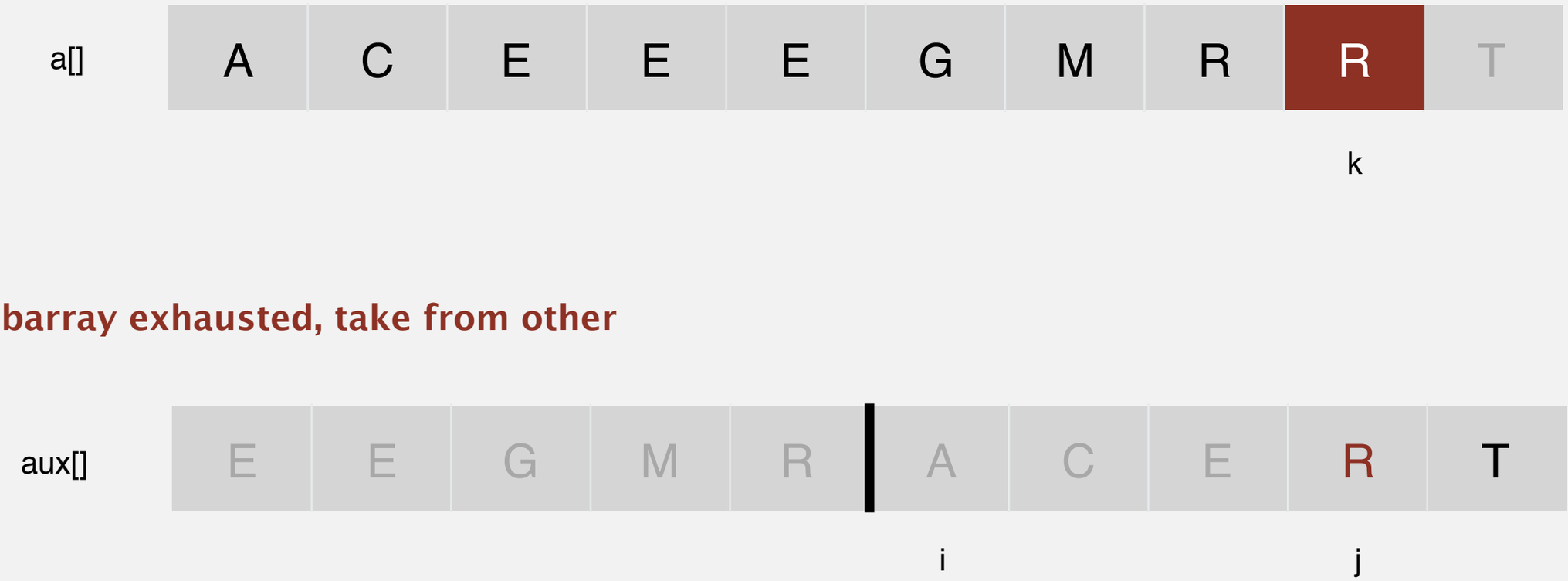
# Abstract in-place merge

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]`.



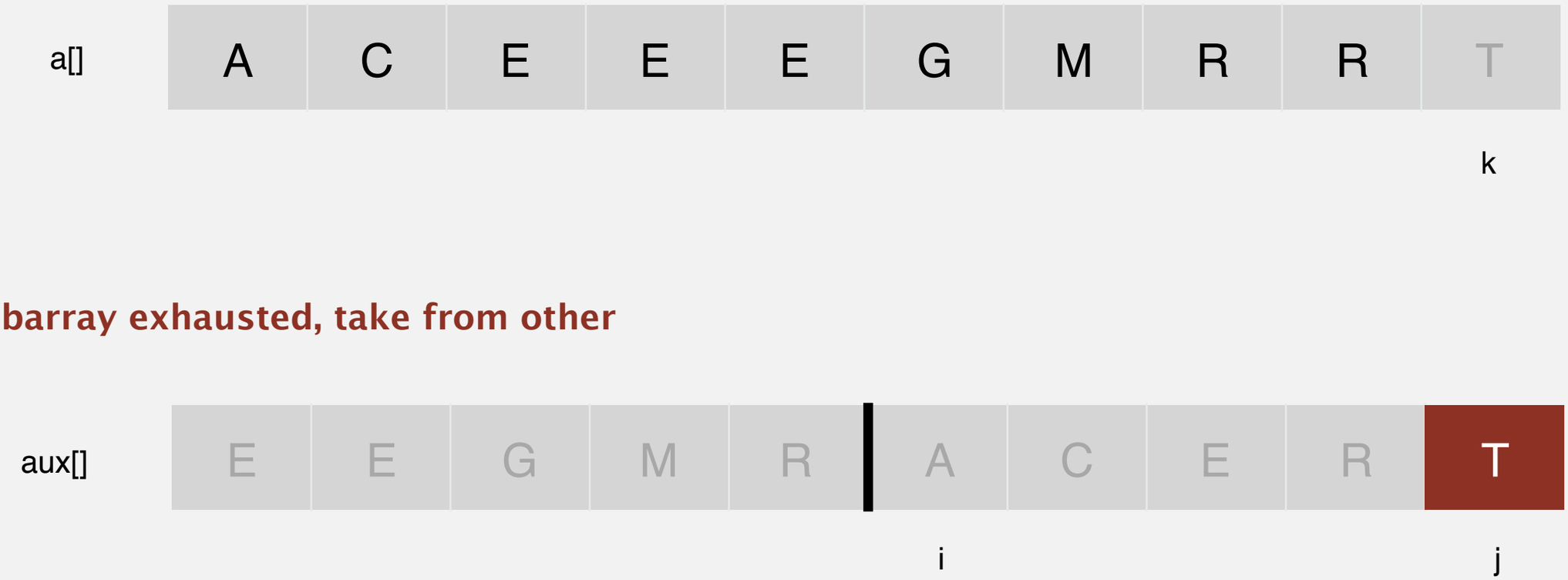
# Abstract in-place merge

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]`.



# Abstract in-place merge

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]`.



# Abstract in-place merge

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]`.

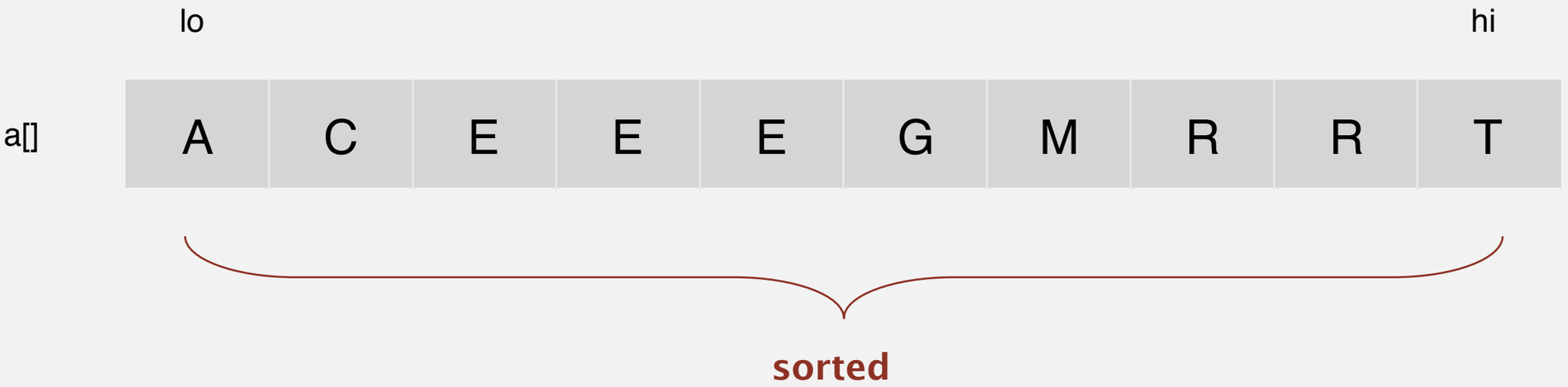

**one subarray exhausted, take from other**

# Abstract in-place merge

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]`.

# Abstract in-place merge

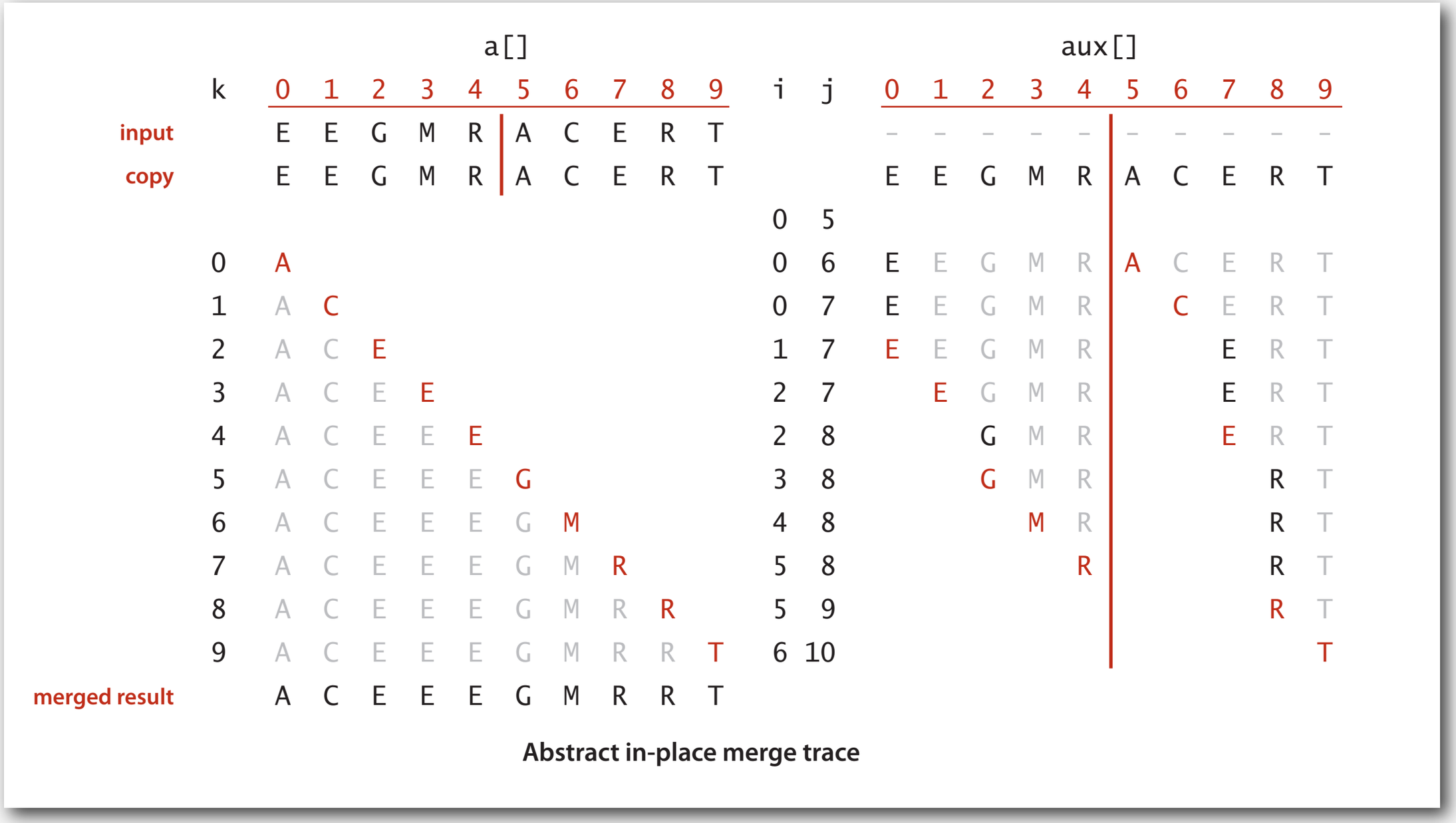
Goal. Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



# Merging

Q. How to combine two sorted subarrays into a sorted whole.

A. Use an auxiliary array.



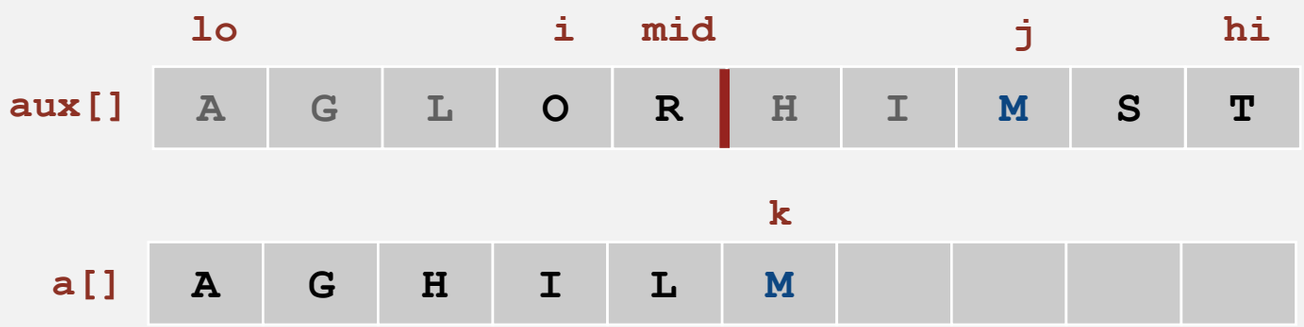

Abstract in-place merge trace

# Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   assert isSorted(a, lo, mid);    // precondition: a[lo..mid]   sorted
   assert isSorted(a, mid+1, hi);  // precondition: a[mid+1..hi] sorted

   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++];
   }

   assert isSorted(a, lo, hi);     // postcondition: a[lo..hi] sorted
}
```

copy

merge



# Assertions

Assertion. Statement to test assumptions about your program.

- Helps detect logic bugs.
- Documents code.

Java assert statement. Throws an exception unless boolean condition is true.

```
assert isSorted(a, lo, hi);
```

Can enable or disable at runtime. ⇒ No cost in production code.

```
java -ea MyProgram   // enable assertions
java -da MyProgram   // disable assertions (default)
```

Best practices. Use to check internal invariants. Assume assertions will be disabled in production code (so do not use for external argument-checking).

# Mergesort: Java implementation

```
public class Merge
{
   private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
   {  /* 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)
   {
      aux = new Comparable[a.length];
      sort(a, aux, 0, a.length - 1);
   }
}
```

# Mergesort: trace

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

lo

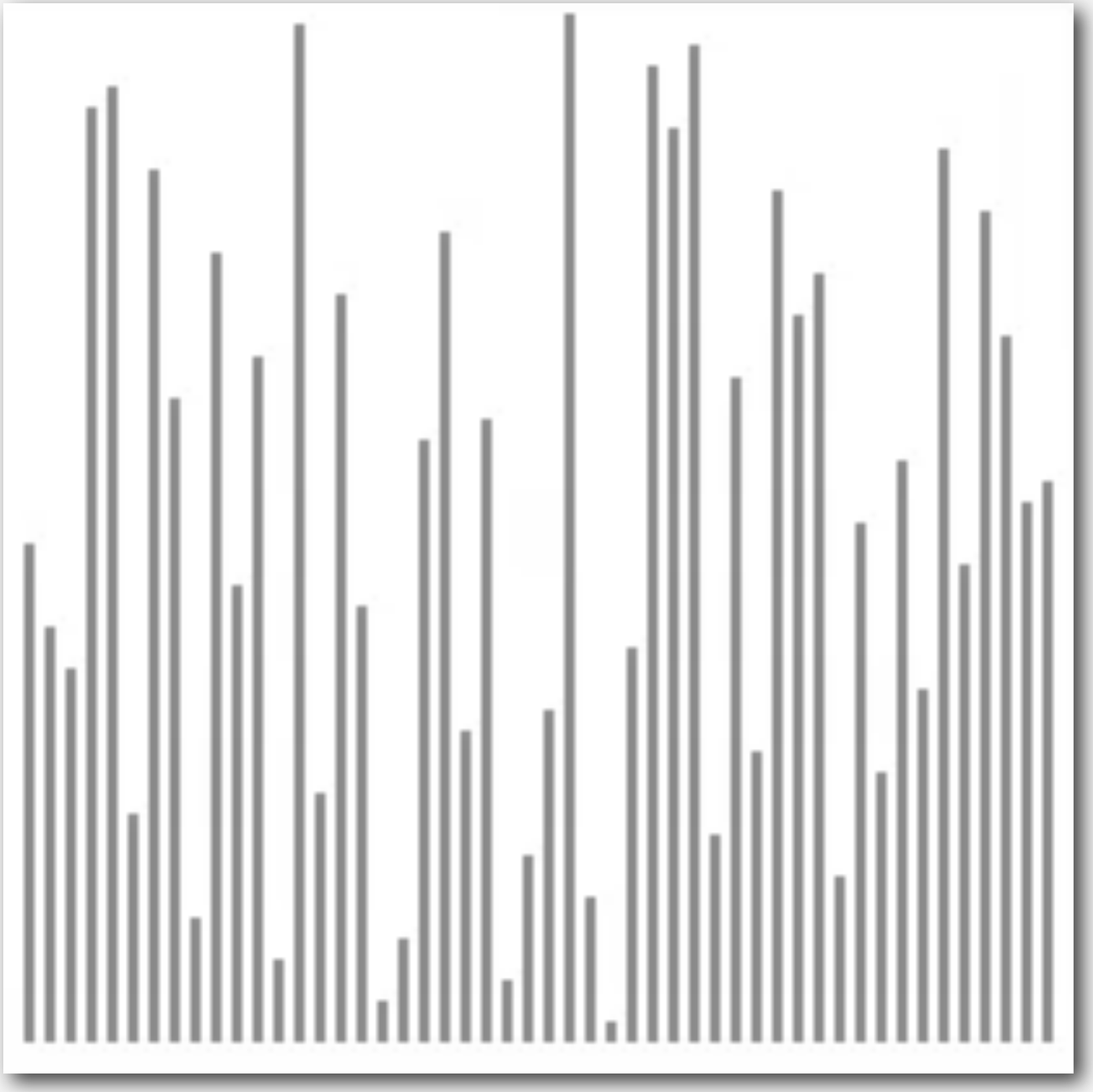
hi

**Trace of merge results for top-down mergesort**

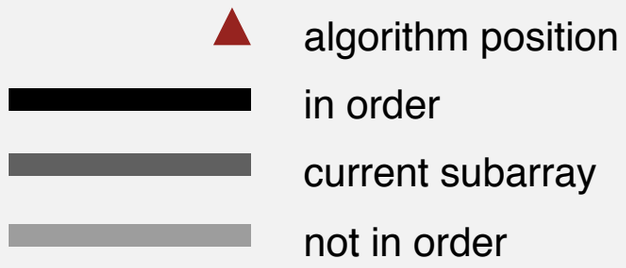
result after recursive call

# Mergesort: animation

50 random items

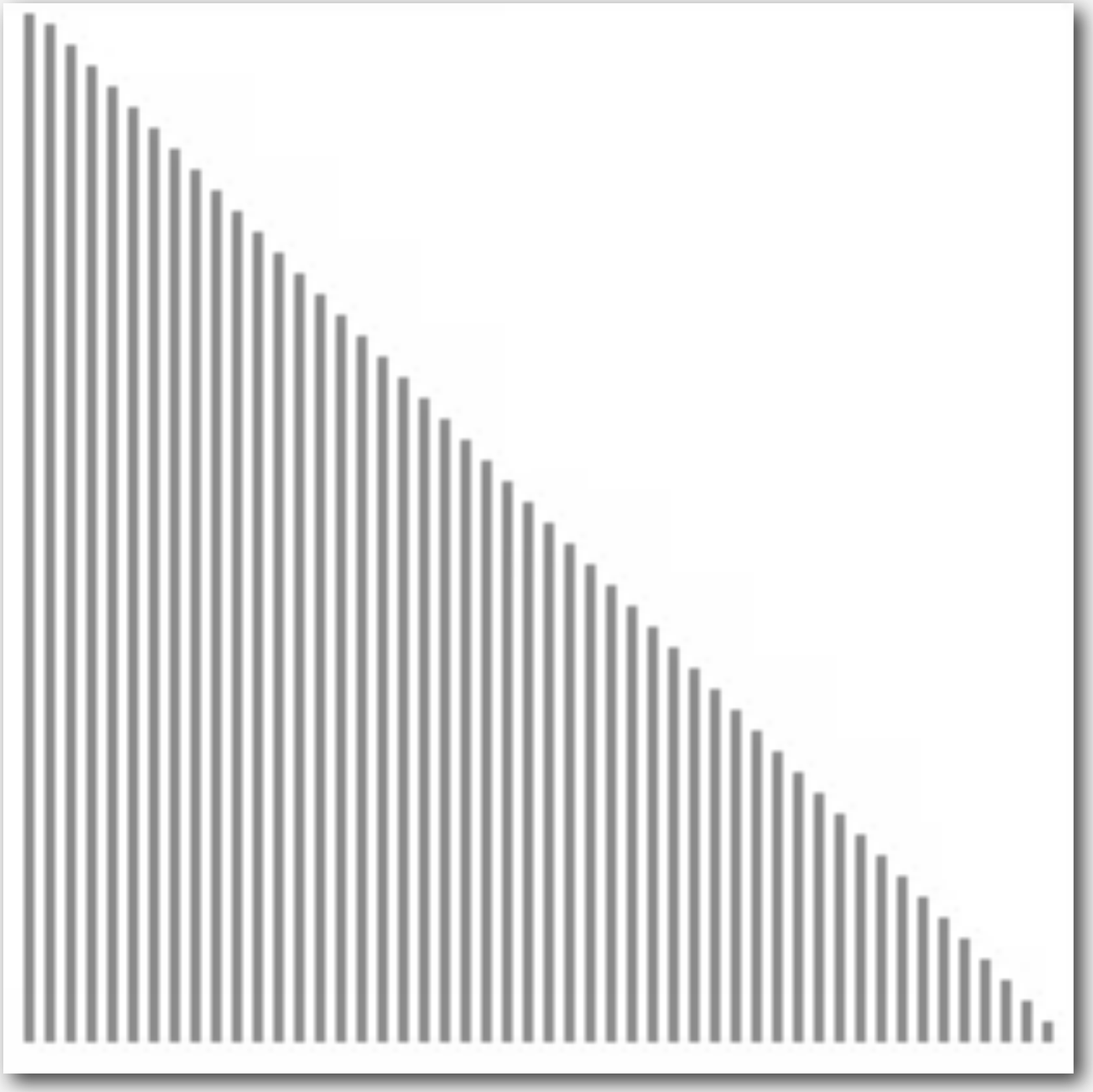


http://www.sorting-algorithms.com/merge-sort

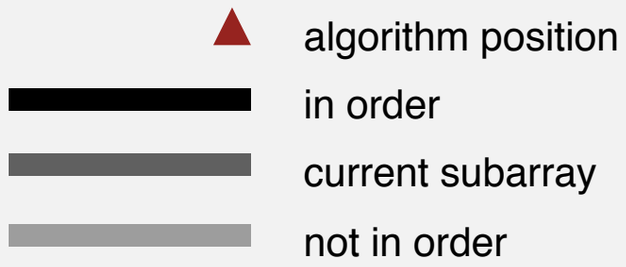


# Mergesort: animation

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: number of compares and array accesses

Proposition. Mergesort uses at most $N \lg N$ compares and $6\, N \lg N$ array accesses to sort any array of size $N$.

Pf sketch. The number of compares $C(N)$ and array accesses $A(N)$ to mergesort an array of size $N$ satisfy the recurrences:

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

left half · right half · merge

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

We solve the recurrence when $N$ is a power of 2.

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

# Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   assert isSorted(a, lo, mid);    // precondition: a[lo..mid]   sorted
   assert isSorted(a, mid+1, hi);  // precondition: a[mid+1..hi] sorted

   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++];
   }

   assert isSorted(a, lo, hi);     // postcondition: a[lo..hi] sorted
}
```

copy

2N

merge

2N

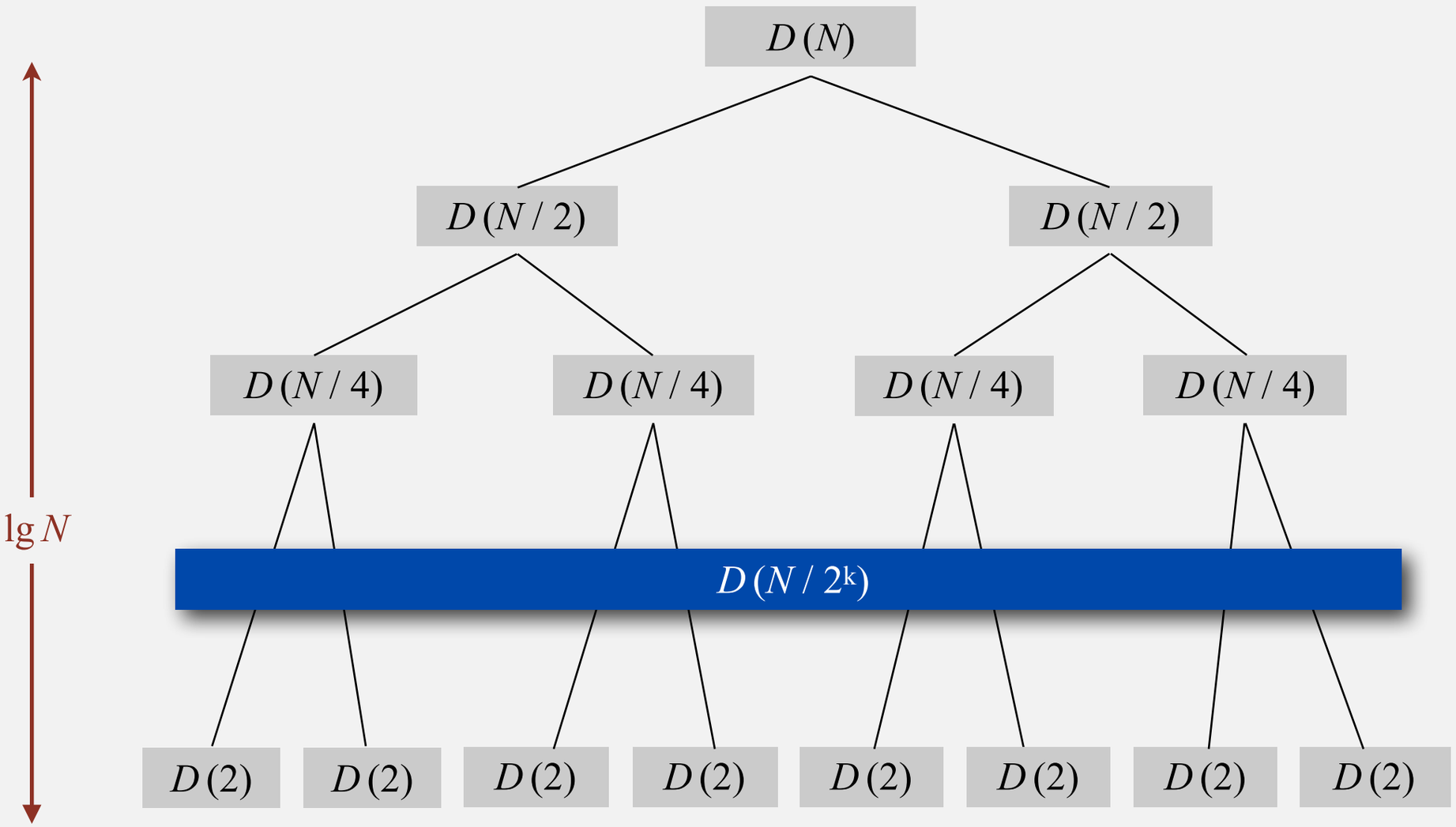
2N

**Proof:** Each merge uses at most $6N$ array accesses ($2N$ for the copy, $2N$ for the move back, and at most $2N$ for compares). The result follows from the same argument as for PROPOSITION F.

# Divide-and-conquer recurrence: proof by picture

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




| | |
|---|---|
| $N$ | $= N$ |
| $2\ (N/2)$ | $= N$ |
| $4\ (N/4)$ | $= N$ |
| ... | |
| $2^k\ (N/2^k)$ | $= N$ |
| ... | |
| $N/2\ (2)$ | $= N$ |
| | $N \lg N$ |

# Divide-and-conquer recurrence: proof by expansion

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 2. [assuming $N$ is a power of 2]

$$
\begin{aligned}
D(N) &= 2\,D(N/2) + N && \text{given} \\
D(N)/N &= 2\,D(N/2)/N + 1 && \text{divide both sides by N} \\
&= D(N/2)/(N/2) + 1 && \text{algebra} \\
&= D(N/4)/(N/4) + 1 + 1 && \text{apply to first term} \\
&= D(N/8)/(N/8) + 1 + 1 + 1 && \text{apply to first term again} \\
&\ldots \\
&= D(N/N)/(N/N) + 1 + 1 + \ldots + 1 \\
&= \lg N && \text{stop applying, D(1) = 0}
\end{aligned}
$$

# Divide-and-conquer recurrence: proof by induction

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 3. [assuming $N$ is a power of 2]

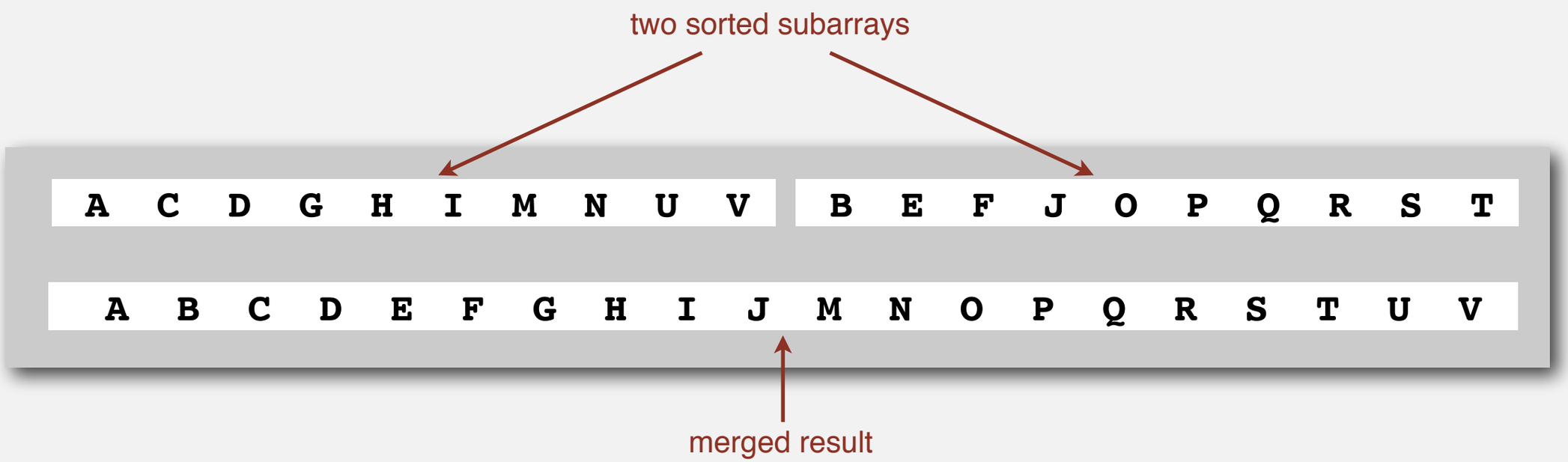
- Base case: $N = 1$.
- Inductive hypothesis: $D(N) = N \lg N$.
- Goal: show that $D(2N) = (2N) \lg (2N)$.

$$
\begin{aligned}
D(2N) &= 2\,D(N) + 2N && \text{given} \\
&= 2\,N \lg N + 2N && \text{inductive hypothesis} \\
&= 2\,N\,(\lg (2N) - 1) + 2N && \text{algebra} \\
&= 2\,N \lg (2N) && \text{QED}
\end{aligned}
$$

# Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to $N$.

Pf. The array `aux[]` needs to be of size $N$ 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 for the bored. In-place merge. [Kronrod, 1969]

# Mergesort: practical improvements

Use insertion sort for small subarrays.

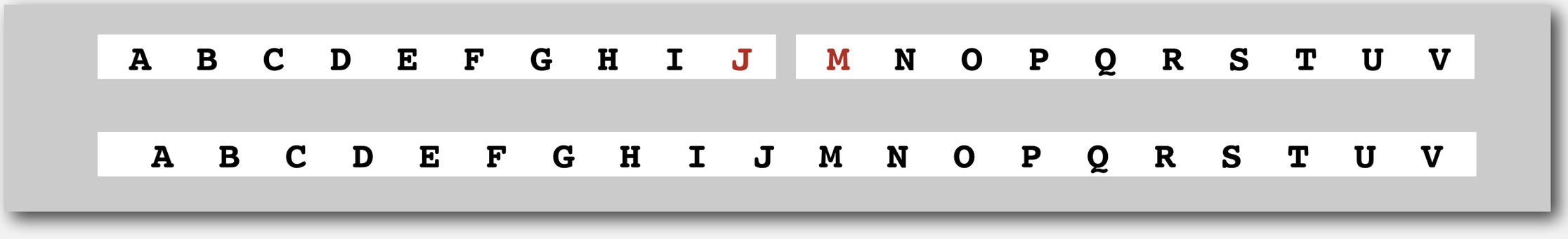
- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 7 items.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1) Insertion.sort(a, lo, hi);
   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 biggest item in first half ≤ smallest item in second half?
- Helps for partially-ordered arrays.




```
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

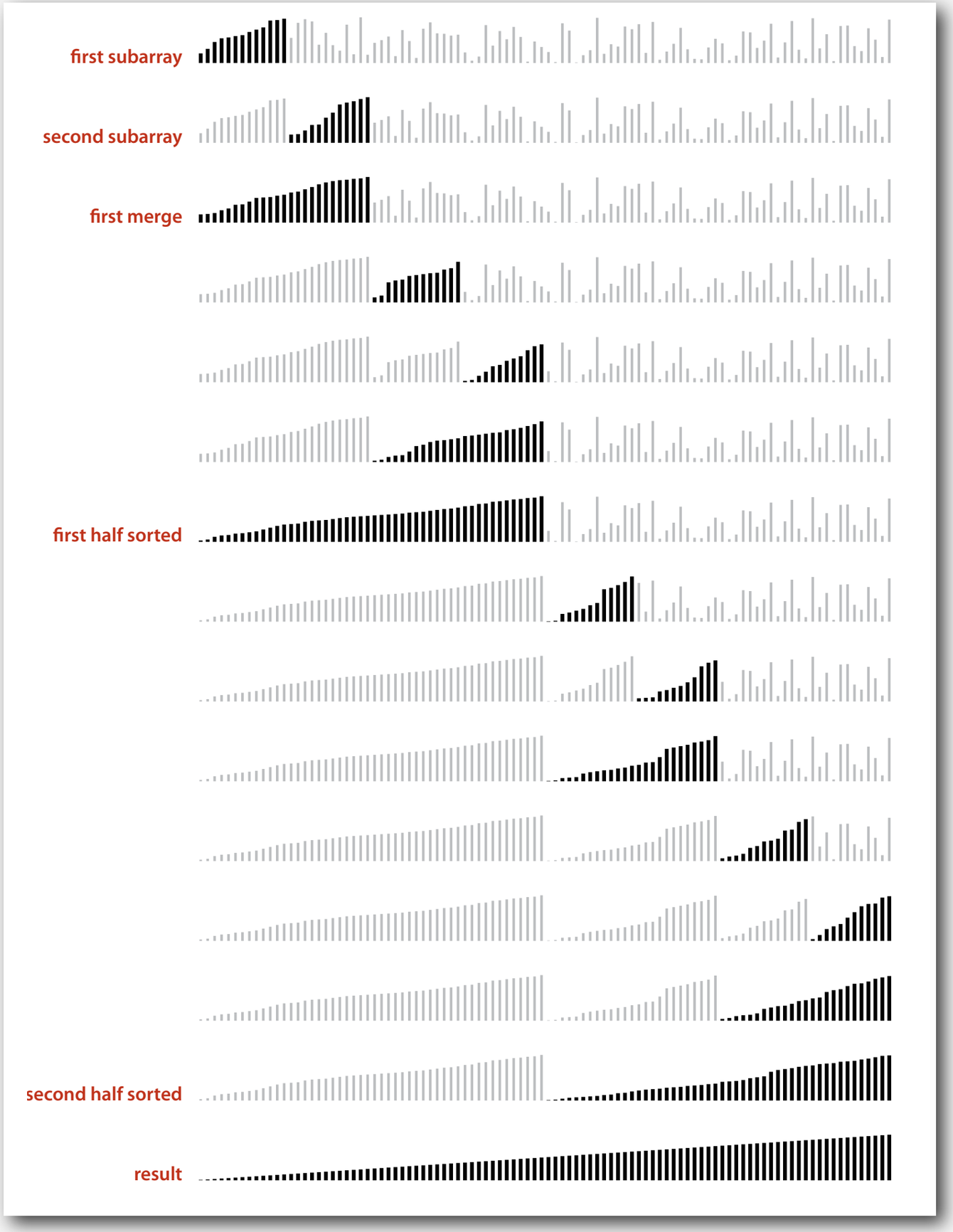
Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

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

merge from a[] to aux[]

switch roles of aux[] and a[]

# Mergesort: visualization



# Bottom-up mergesort

## Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16, ....

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

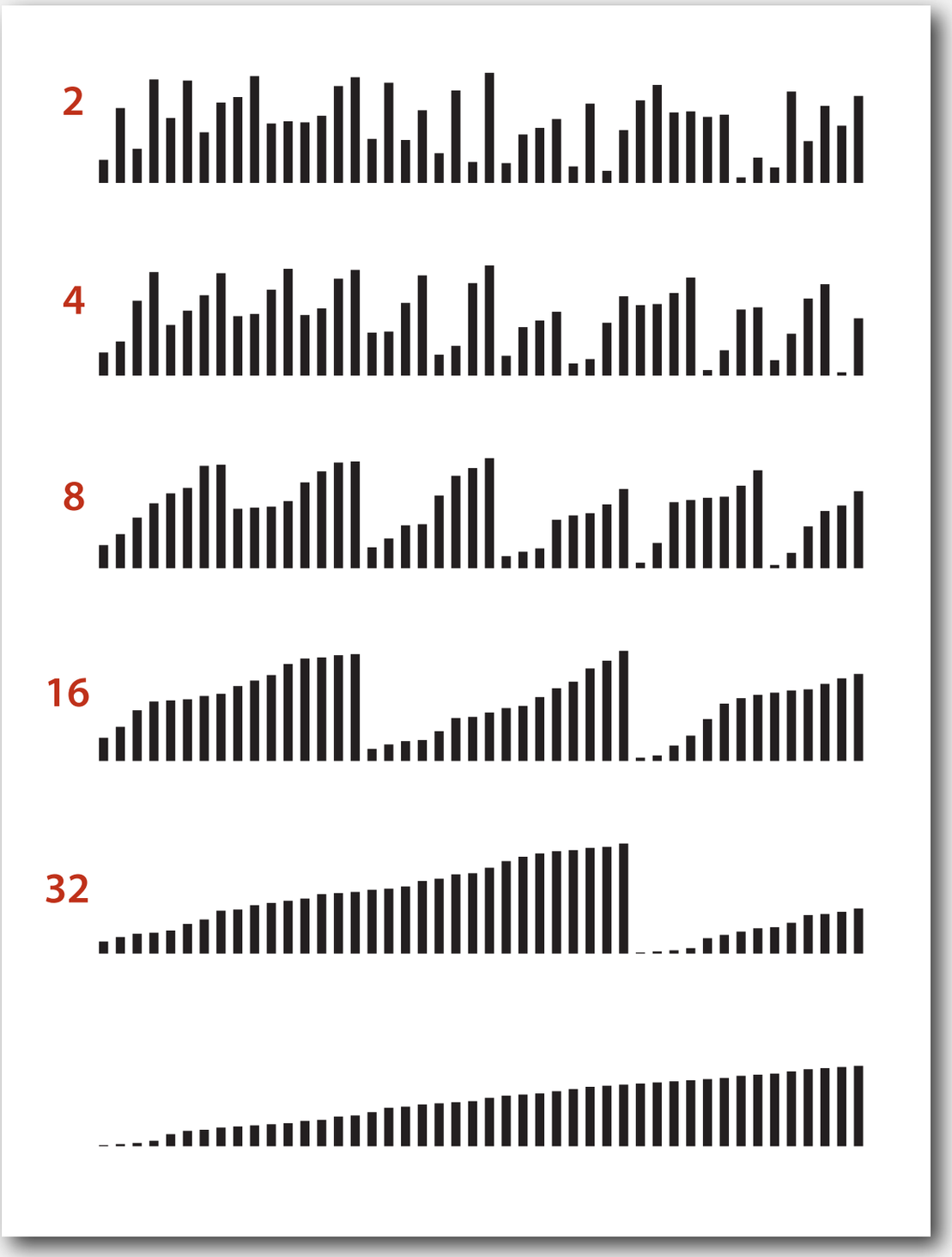
## Bottom line. No recursion needed!

# Bottom-up mergesort: Java implementation

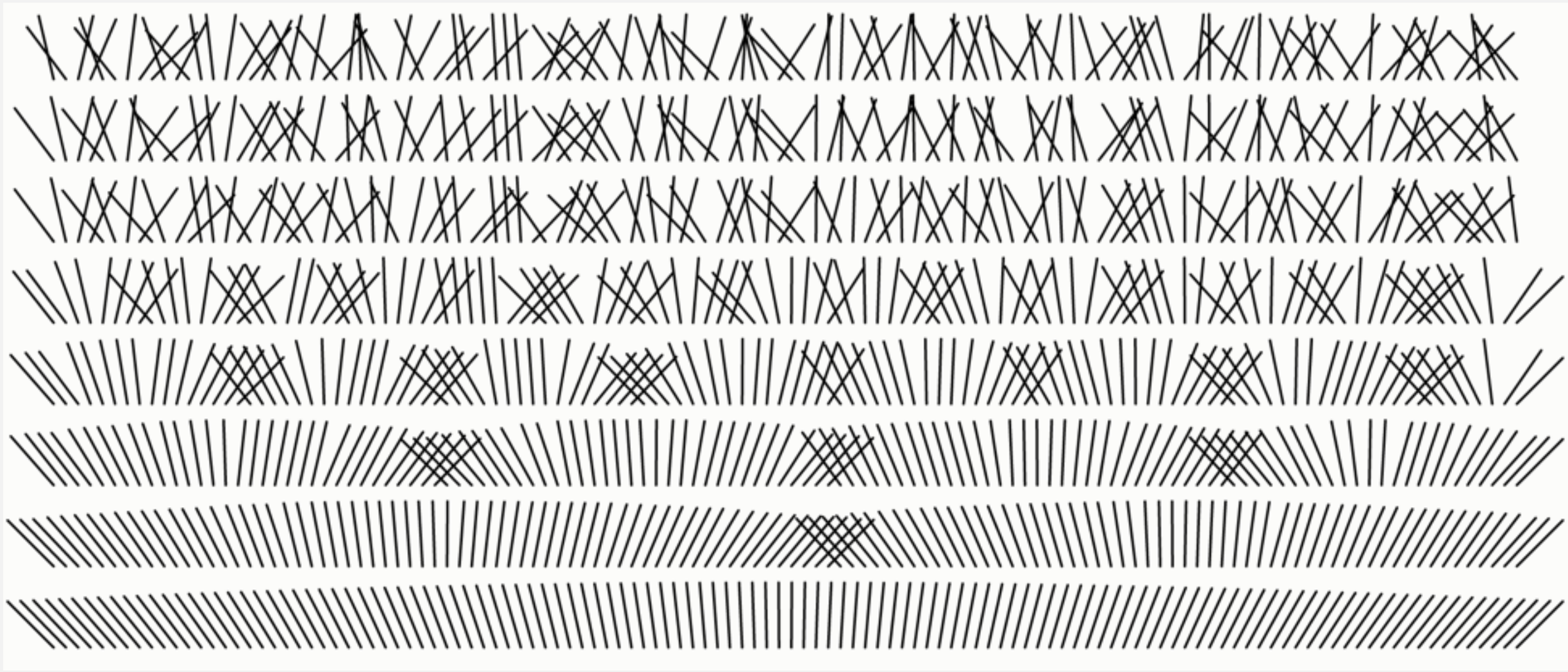
```
public class MergeBU
{
   private static Comparable[] aux;

   private static void merge(Comparable[] a, int lo, int mid, int hi)
   {  /* as before */  }

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

Bottom line. Concise industrial-strength code, if you have the space.

# Bottom-up mergesort: visual trace



# Bottom-up mergesort: visual trace



http://bl.ocks.org/mbostock/39566aca95eb03ddd526

# 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 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 $X$.

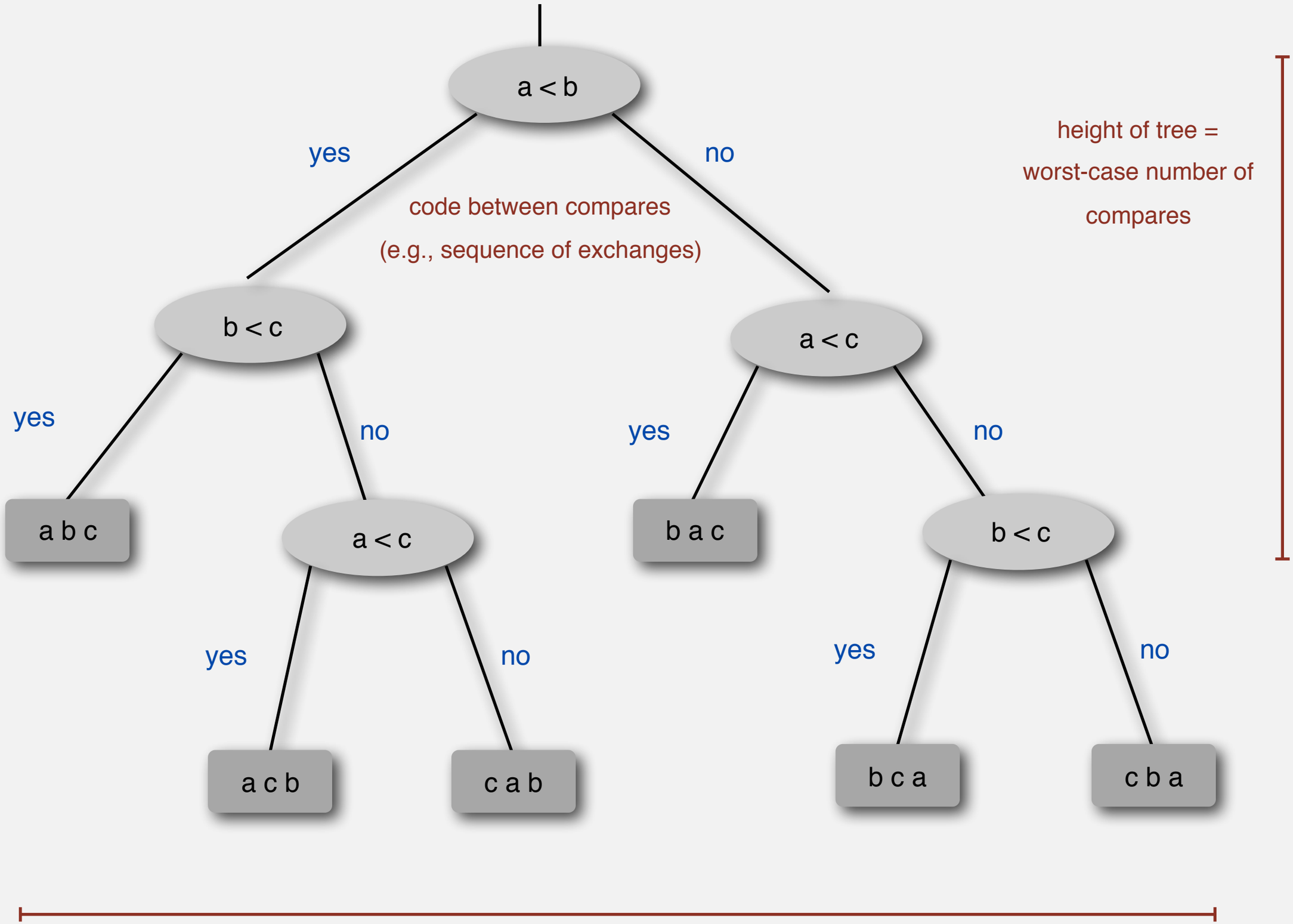
lower bound ~ upper bound

Example: sorting.

- Model of computation: decision tree.
- Cost model: # compares.
- Upper bound: ~ $N \lg N$ from mergesort.
- Lower bound: ?
- Optimal algorithm: ?

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

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



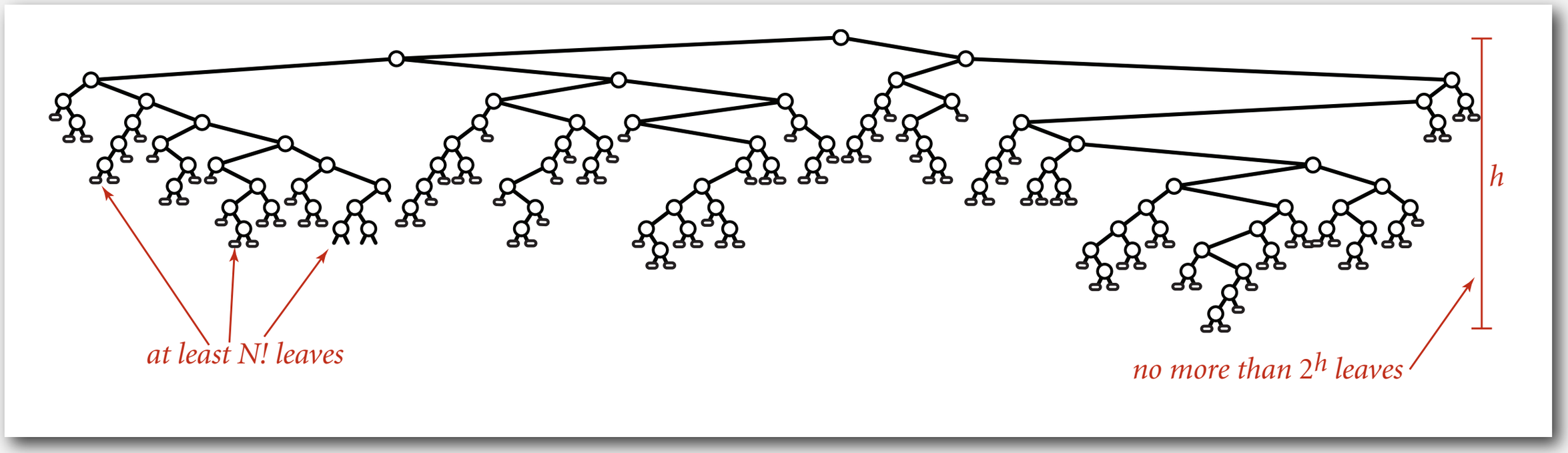

(at least) one leaf for each possible ordering

# 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 $X$.

Example: sorting.

- Model of computation: decision tree.
- Cost model: # compares.
- Upper bound: $\sim N \lg N$ from mergesort.
- Lower bound: $\sim N \lg N$.
- Optimal algorithm = mergesort.

First goal of algorithm design: optimal algorithms.

# Complexity results in context

Other operations? Mergesort is optimal with respect to number of compares (e.g., but not with respect to number of array accesses).

Space?

- Mergesort is not optimal with respect to space usage.
- Insertion sort, selection sort, and shellsort are space-optimal.

Challenge. Find an algorithm that is both time- and space-optimal.
[stay tuned]

Lessons. Use theory as a guide.

Ex. Don't try to design sorting algorithm that guarantees $\frac{1}{2} N \lg N$ compares.

# Complexity results in context (continued)

Lower bound may not hold if the algorithm has information about:

- The initial order of the input.
- The distribution of key values.
- The representation of the keys.

Partially-ordered arrays. Depending on the initial order of the input, we may not need $N \lg N$ compares.

insertion sort requires only N−1 compares if input array is sorted

Duplicate keys. Depending on the input distribution of duplicates, we may not need $N \lg N$ compares.

stay tuned for 3-way quicksort

Digital properties of keys. We can use digit/character compares instead of key compares for numbers and strings.

stay tuned for radix sorts

# Comparable interface: review

Comparable interface: sort using a type's natural order.

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

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }

   …

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

natural order

# Comparator interface

Comparator interface: sort using an alternate order.

```
public interface Comparator<Key>
```

| | |
|---|---|
| `int compare(Key v, Key w)` | *compare keys v and w* |

Required property. Must be a total order.

Ex. Sort strings by:

- Natural order.
- Case insensitive.
- Spanish.
- British phone book.
- ...

```
Now is the time
is Now the time
café cafetero cuarto churro nube ñoño
McKinley Mackintosh
```

pre-1994 order for digraphs ch and ll and rr

# Comparator interface: system sort

To use with Java system sort:

- Create `Comparator` object.
- Pass as second argument to `Arrays.sort()`.

```
String[] a;
...
Arrays.sort(a);
...
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
...
Arrays.sort(a, Collator.getInstance(new Locale("es")));
...
Arrays.sort(a, new BritishPhoneBookOrder());
...
```

uses natural order

uses alternate order defined by `Comparator<String>` object

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

# Comparator interface: using with our sorting libraries

To support comparators in our sort implementations:

- Use `Object` instead of `Comparable`.
- Pass `Comparator` to `sort()` and `less()` and use it in `less()`.

**insertion sort using a Comparator**

```
public static void sort(Object[] a, Comparator comparator)
{
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i; j > 0 && less(comparator, a[j], a[j-1]); j--)
         exch(a, j, j-1);
}

private static boolean less(Comparator c, Object v, Object w)
{  return c.compare(v, w) < 0;   }

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

# Comparator interface: implementing

To implement a comparator:

- Define a (nested) class that implements the `Comparator` interface.
- Implement the `compare()` method.

```
public class Student
{
   public static final Comparator<Student> BY_NAME    = new ByName();
   public static final Comparator<Student> BY_SECTION = new BySection();
   private final String name;
   private final int section;
   ...

   private static class ByName implements Comparator<Student>
   {
      public int compare(Student v, Student w)
      {  return v.name.compareTo(w.name);  }
   }

   private static class BySection implements Comparator<Student>
   {
      public int compare(Student v, Student w)
      {  return v.section - w.section;  }
   }
}
```

one Comparator for the class

this technique works here since no danger of overflow

# Comparator interface: implementing

To implement a comparator:

- Define a (nested) class that implements the `Comparator` interface.
- Implement the `compare()` method.

```
Arrays.sort(a, Student.BY_NAME);
```

| Andrews | 3 | A | 664-480-0023 | 097 Little |
|---|---|---|---|---|
| Battle | 4 | C | 874-088-1212 | 121 Whitman |
| Chen | 3 | A | 991-878-4944 | 308 Blair |
| Fox | 3 | A | 884-232-5341 | 11 Dickinson |
| Furia | 1 | A | 766-093-9873 | 101 Brown |
| Gazsi | 4 | B | 766-093-9873 | 101 Brown |
| Kanaga | 3 | B | 898-122-9643 | 22 Brown |
| Rohde | 2 | A | 232-343-5555 | 343 Forbes |

```
Arrays.sort(a, Student.BY_SECTION);
```

| Furia | 1 | A | 766-093-9873 | 101 Brown |
|---|---|---|---|---|
| Rohde | 2 | A | 232-343-5555 | 343 Forbes |
| Andrews | 3 | A | 664-480-0023 | 097 Little |
| Chen | 3 | A | 991-878-4944 | 308 Blair |
| Fox | 3 | A | 884-232-5341 | 11 Dickinson |
| Kanaga | 3 | B | 898-122-9643 | 22 Brown |
| Battle | 4 | C | 874-088-1212 | 121 Whitman |
| Gazsi | 4 | B | 766-093-9873 | 101 Brown |

# Stability

A typical application. First, sort by name; then sort by section.

`Selection.sort(a, Student.BY_NAME);`

| | | | | |
|---|---|---|---|---|
| Andrews | 3 | A | 664-480-0023 | 097 Little |
| Battle | 4 | C | 874-088-1212 | 121 Whitman |
| Chen | 3 | A | 991-878-4944 | 308 Blair |
| Fox | 3 | A | 884-232-5341 | 11 Dickinson |
| Furia | 1 | A | 766-093-9873 | 101 Brown |
| Gazsi | 4 | B | 766-093-9873 | 101 Brown |
| Kanaga | 3 | B | 898-122-9643 | 22 Brown |
| Rohde | 2 | A | 232-343-5555 | 343 Forbes |

`Selection.sort(a, Student.BY_SECTION);`

| | | | | |
|---|---|---|---|---|
| Furia | 1 | A | 766-093-9873 | 101 Brown |
| Rohde | 2 | A | 232-343-5555 | 343 Forbes |
| Chen | 3 | A | 991-878-4944 | 308 Blair |
| Fox | 3 | A | 884-232-5341 | 11 Dickinson |
| Andrews | 3 | A | 664-480-0023 | 097 Little |
| Kanaga | 3 | B | 898-122-9643 | 22 Brown |
| Gazsi | 4 | B | 766-093-9873 | 101 Brown |
| Battle | 4 | C | 874-088-1212 | 121 Whitman |

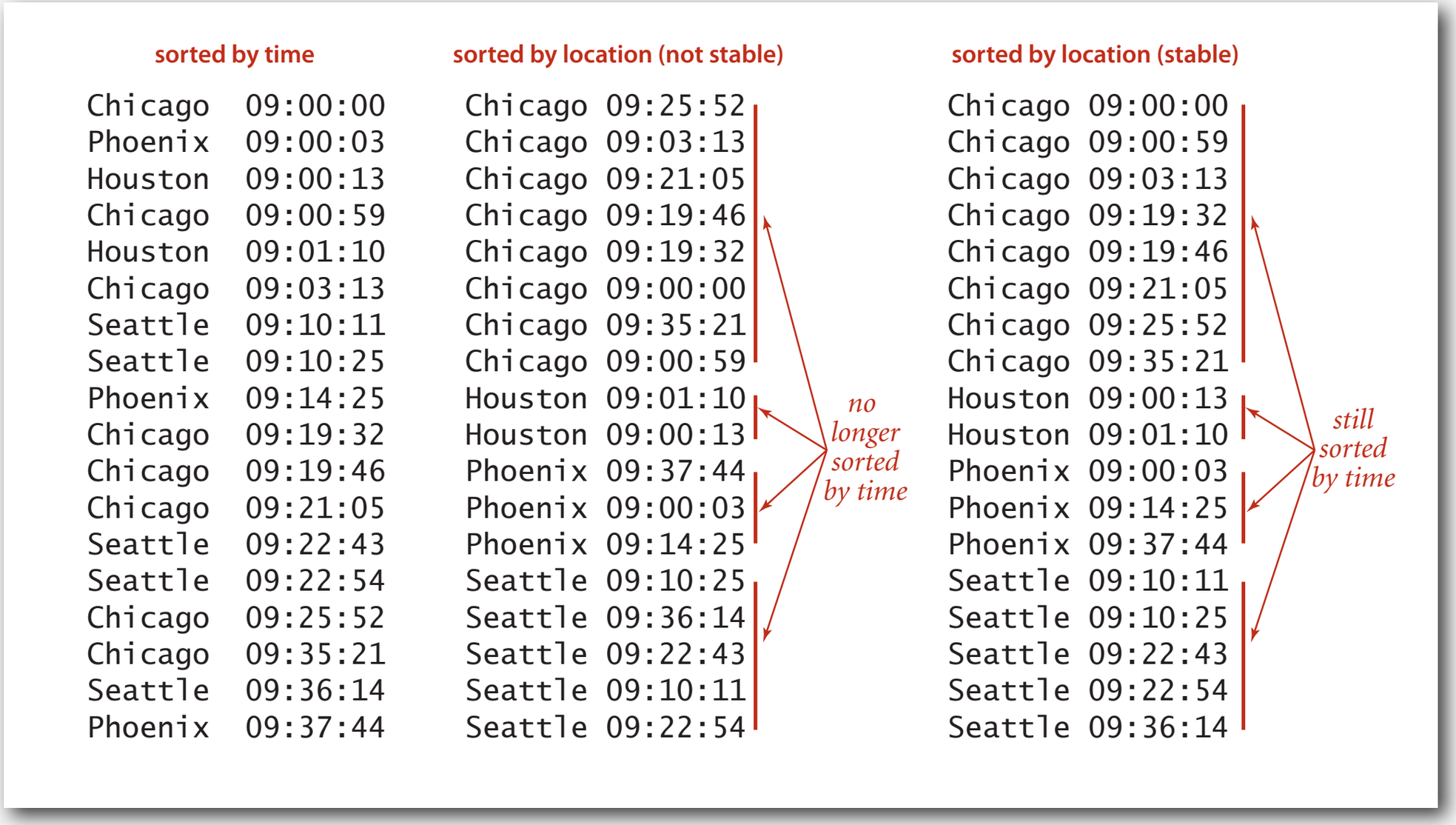
@#%&@! Students in section 3 no longer sorted by name.

A stable sort preserves the relative order of items with equal keys.

# Stability

Q. Which sorts are stable?

A. Insertion sort and mergesort (but not selection sort or shellsort).




Note. Need to carefully check code ("less than" vs "less than or equal to").

# Stability: insertion sort

Proposition. Insertion sort is stable.

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
            exch(a, j, j-1);
   }
}
```

| i | j | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|
| 0 | 0 | $B_1$ | $A_1$ | $A_2$ | $A_3$ | $B_2$ |
| 1 | 0 | $A_1$ | $B_1$ | $A_2$ | $A_3$ | $B_2$ |
| 2 | 1 | $A_1$ | $A_2$ | $B_1$ | $A_3$ | $B_2$ |
| 3 | 2 | $A_1$ | $A_2$ | $A_3$ | $B_1$ | $B_2$ |
| 4 | 4 | $A_1$ | $A_2$ | $A_3$ | $B_1$ | $B_2$ |
| | | $A_1$ | $A_2$ | $A_3$ | $B_1$ | $B_2$ |

Pf. Equal items never move past each other.

# Stability: selection sort

Proposition. Selection sort is not stable.

```
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], a[min]))
               min = j;
         exch(a, i, min);
      }
   }
}
```

| i | min | 0 | 1 | 2 |
|---|---|---|---|---|
| 0 | 2 | $B_1$ | $B_2$ | A |
| 1 | 1 | A | $B_2$ | $B_1$ |
| 2 | 2 | A | $B_2$ | $B_1$ |
| | | A | $B_2$ | $B_1$ |

Pf by counterexample. Long-distance exchange might move an item past some equal item.

# Stability: shellsort

Proposition. Shellsort sort is not stable.

```
public class Shell
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      int h = 1;
      while (h < N/3) h = 3*h + 1;
      while (h >= 1)
      {
         for (int i = h; i < N; i++)
         {
            for (int j = i; j > h && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
         }
         h = h/3;
      }
   }
}
```

| h | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| | $B_1$ | $B_2$ | $B_3$ | $B_4$ | $A_1$ |
| 4 | $A_1$ | $B_2$ | $B_3$ | $B_4$ | $B_1$ |
| 1 | $A_1$ | $B_2$ | $B_3$ | $B_4$ | $B_1$ |
| | $A_1$ | $B_2$ | $B_3$ | $B_4$ | $B_1$ |

Pf by counterexample. Long-distance exchanges.

# Stability: mergesort

Proposition. Mergesort is stable.

```
public class Merge
{
   private static Comparable[] aux;
   private static void merge(Comparable[] a, int lo, int mid, int hi)
   {  /* as before */  }

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

| 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.