Sorting

Comp Sci 2C03

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Acknowledgments: Material mainly based on *Algorithms* by Robert Sedgewick and Kevin Wayne (Chapters 2.1, 2.2 and 2.3)

Selection Sort (see demo)

- Scans from left to right
- In iteration i, find index min of smallest remaining entry.
- Swap a[i] and a[min].

Pelee	Agung	Agung	Agung	Agung	Agung
Etna	Etna	Etna	Etna	Etna	Etna
Krakatoa	Krakatoa	Krakatoa	Krakatoa	Krakatoa	Krakatoa
Agung	Pelee	Pelee	Pelee	Pelee	Pelee
Vesuvius	Vesuvius	Vesuvius	Vesuvius	Vesuvius	St. Helens
St. Helens	St. Helens	St. Helens	St. Helens	St. Helens	Vesuvius
initial	after $i = 1$	after $i = 2$	after $i = 3$	after $i = 4$	after $i = 5$

Two Useful Sorting Abstractions

Less. Is item `v` less than `w` ?

```
private static boolean less(Comparable v, Comparable w)
{  return v.compareTo(w) < 0;  }
```

Exchange. Swap item in array `a[]` at index `i` with the one at index `j`.

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

Selection Sort: Java

```
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);
      }
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Selection Sort: Mathematical Analysis

Proposition

Selection sort uses $(N–1) + (N–2) + \ldots + 1 + 0 \sim N^2/2$ *compares and* N *exchanges, i.e.* $T(N) = \Theta(N^2/2 + N) = \Theta(N^2)$.

- *Running time insensitive to input*. Quadratic time, even if input is sorted.
- *Data movement is minimal*. Linear number of exchanges.

Trace of Selection Sort

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum

entries in red are a[min]

entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

Insertion Sort (see demo)

- Scan from the left to right.
- In iteration i, swap a[i] with each larger entry to its left.

initial	after $i=2$	after $i=3$	after $i=4$	after $i=5$	after $i=6$
Pelee	Etna	Etna	Agung	Agung	Agung
Etna	Pelee	Krakatoa	Etna	Etna	Etna
Krakatoa	Krakatoa	Pelee	Krakatoa	Krakatoa	Krakatoa
Agung	Agung	Agung	Pelee	Pelee	Pelee
Vesuvius	Vesuvius	Vesuvius	Vesuvius	Vesuvius	St. Helens
St. Helens	St. Helens	St. Helens	St. Helens	St. Helens	Vesuvius

Insertion Sort: Java

```
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; j--)
            if (less(a[j], a[j-1]))
               exch(a, j, j-1);
            else break;
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Insertion Sort: Analysis

- Best case. If the array is in ascending order, insertion sort makes N–1 compares and 0 exchanges, .i.e. $T(N) = \Omega(N)$.

  A E E L M O P R S T X

- Worst case. If the array is in descending order (and no duplicates), insertion sort makes $\sim \frac{1}{2}N^2$ compares and $\sim \frac{1}{2}N^2$ exchanges, i.e. $T(N) = O(N^2)$.

  X T S R P O M L F E A

- Average case. To sort a randomly-ordered (uniform or Gaussian distribution) array with distinct keys, insertion sort uses $\sim \frac{1}{4}N^2$ compares and $\sim \frac{1}{4}N^2$ exchanges on average, i.e. $T_{average}(N) = O(N^2)$.

Trace of Insertion Sort

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in gray do not move

entry in red is a[j]

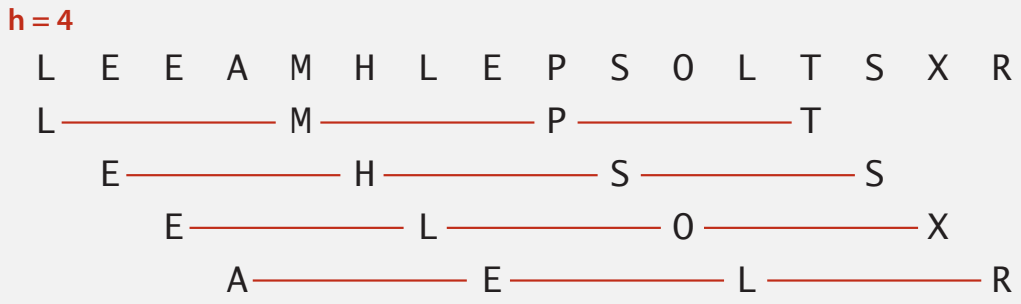
entries in black moved one position right for insertion

Trace of insertion sort (array contents just after each insertion)

Shell Sort: Ideas (see demo for h-sort)

Idea. Move entries more than one position at a time by h-sorting the array.

an h-sorted array is h interleaved sorted subsequences



Shellsort. [Shell 1959] h-sort array for decreasing sequence of values of h.

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

- How to h-sort an array? Insertion sort, with stride length h.

Shellsort example: increments 7, 3, 1

input

S O R T E X A M P L E

7-sort

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

3-sort

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

1-sort

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

result

A E E L M O P R S T X

Shellsort: Java (one of many possibilities)

```
public class Shell
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;

      int h = 1;
      while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, ...

      while (h >= 1)
      {  // h-sort the array.
         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;
      }
   }

   private static boolean less(Comparable v, Comparable w)
   { /* as before */ }
   private static void exch(Comparable[] a, int i, int j)
   { /* as before */ }
}
```

3x+1 increment sequence

insertion sort

move to next increment

Shellsort: which increment sequence to use?

Powers of two. 1, 2, 4, 8, 16, 32, ...

No.

Powers of two minus one. 1, 3, 7, 15, 31, 63, ...

Maybe.

→ 3x + 1. 1, 4, 13, 40, 121, 364, ...

OK. Easy to compute.

Sedgewick. 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, ...

Good. Tough to beat in empirical studies.

merging of $(9 \times 4^i) - (9 \times 2^i) + 1$ and $4^i - (3 \times 2^i) + 1$

Shell Properties

- Useful in practice.
  - Fast unless array size is huge (used for small subarrays).
  - Tiny, fixed footprint for code (used in some embedded systems).
  - Hardware sort prototype.
- Time complexity (the case $3x + 1$):
  - Best: $\Omega(N \log N)$
  - Worst: $O(N^{1.5})$
  - Average: open research problem even for uniform input distribution

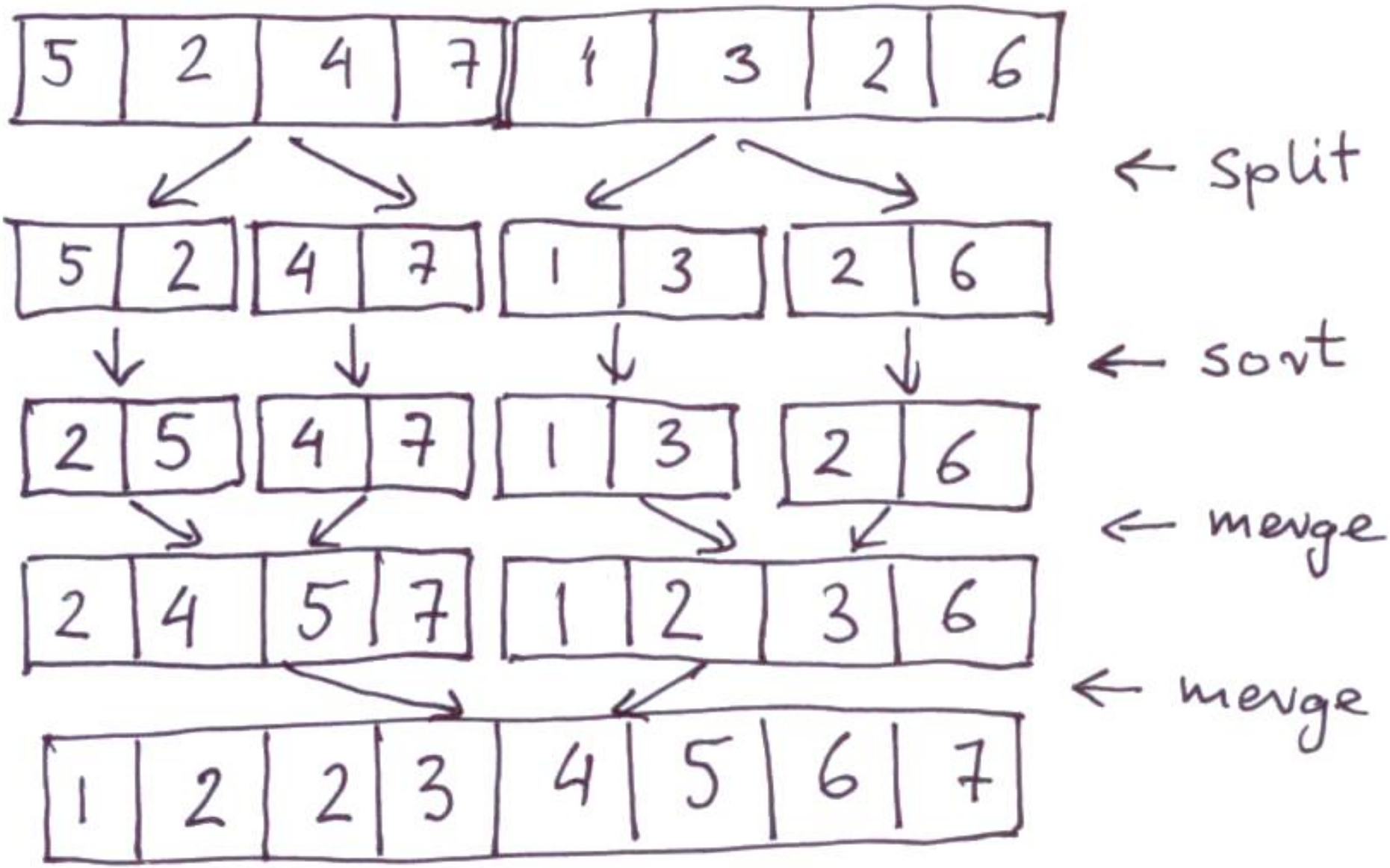
Mergesort: Basic Idea

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: Example (see demo for *merge*)

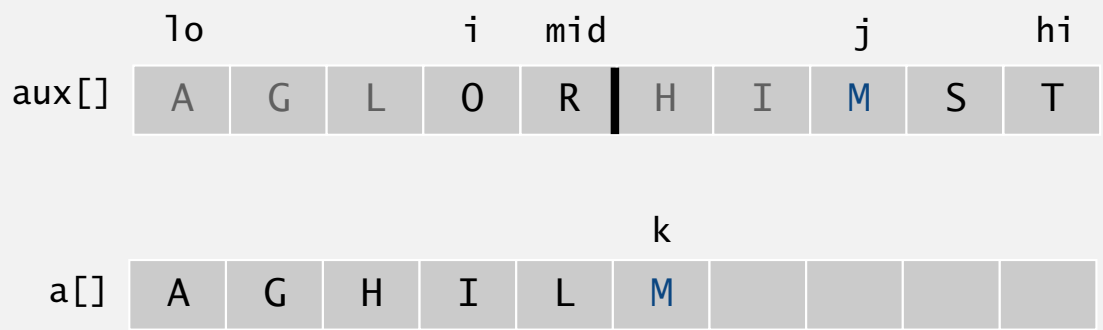


Mergesort: Java

```
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



Mergesort: Time Complexity I

Proposition

Mergesort uses $\Theta(N \log N)$ compares to sort an array of length N.

Proof. Sketch.

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

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

where $\lceil x \rceil$ is the smallest integer $\geq x$, i.e. $\lceil 1.5 \rceil = 2$, $\lceil 3.1 \rceil = 4$,
and $\lfloor x \rfloor$ is the biggest integer $\leq x$, i.e. $\lfloor 1.5 \rfloor = 1$, $\lfloor 3.1 \rfloor = 3$.
We solve the recurrence when N is a power of 2:

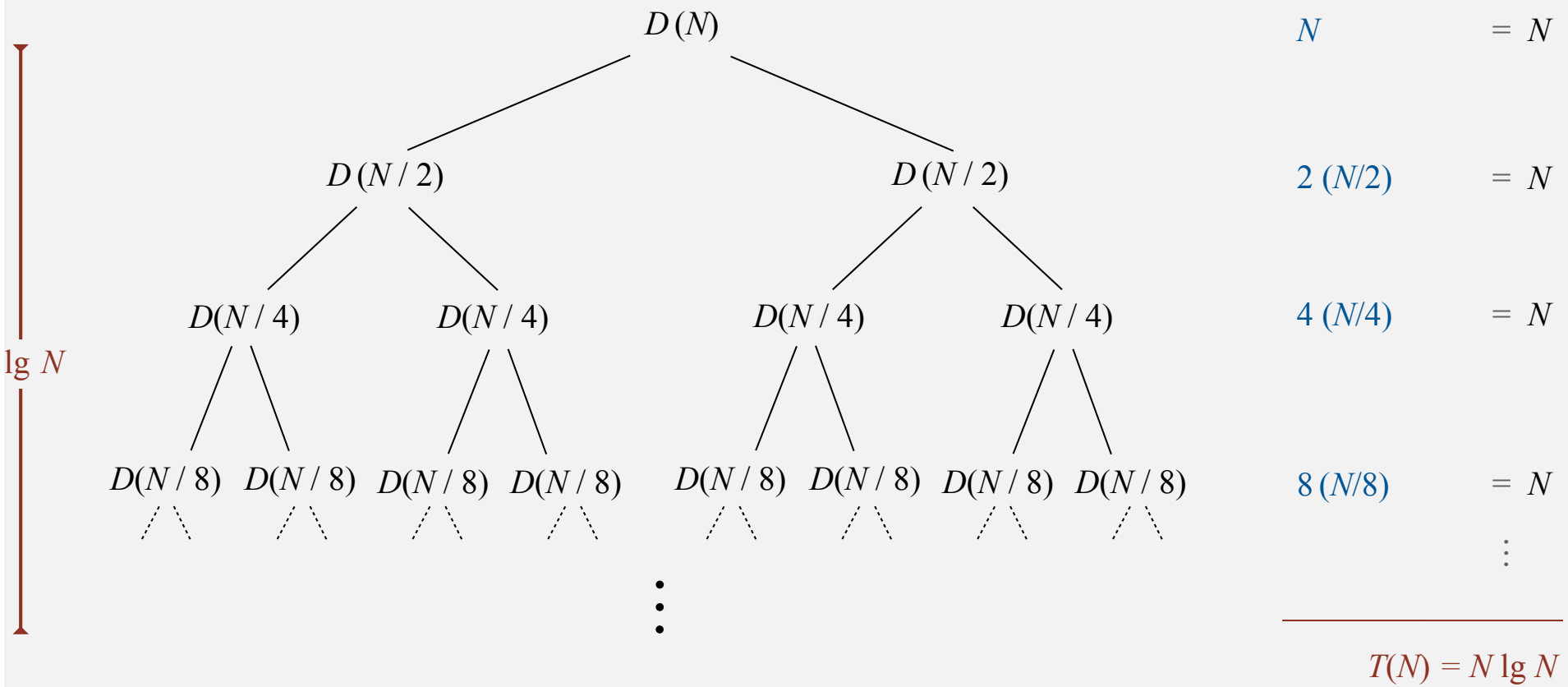
$$D(N) = 2D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0.$$

The result holds for all N, but general proof is a little bit messy. □

Mergesort: Time Complexity II

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]

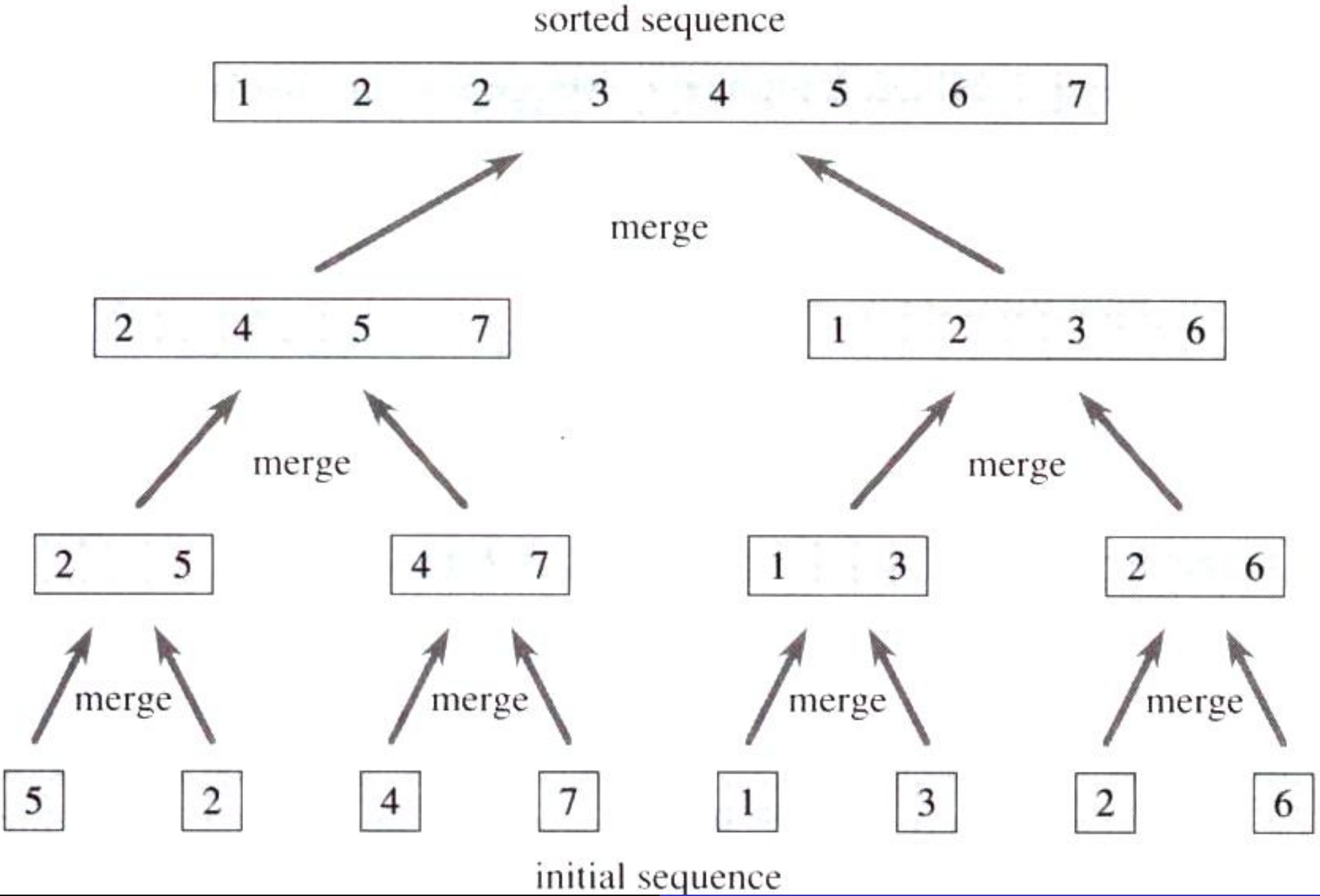


Mergesort: Practical Improvements

Use insertion sort for small subarrays.

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

Mergesort: Bottom Up

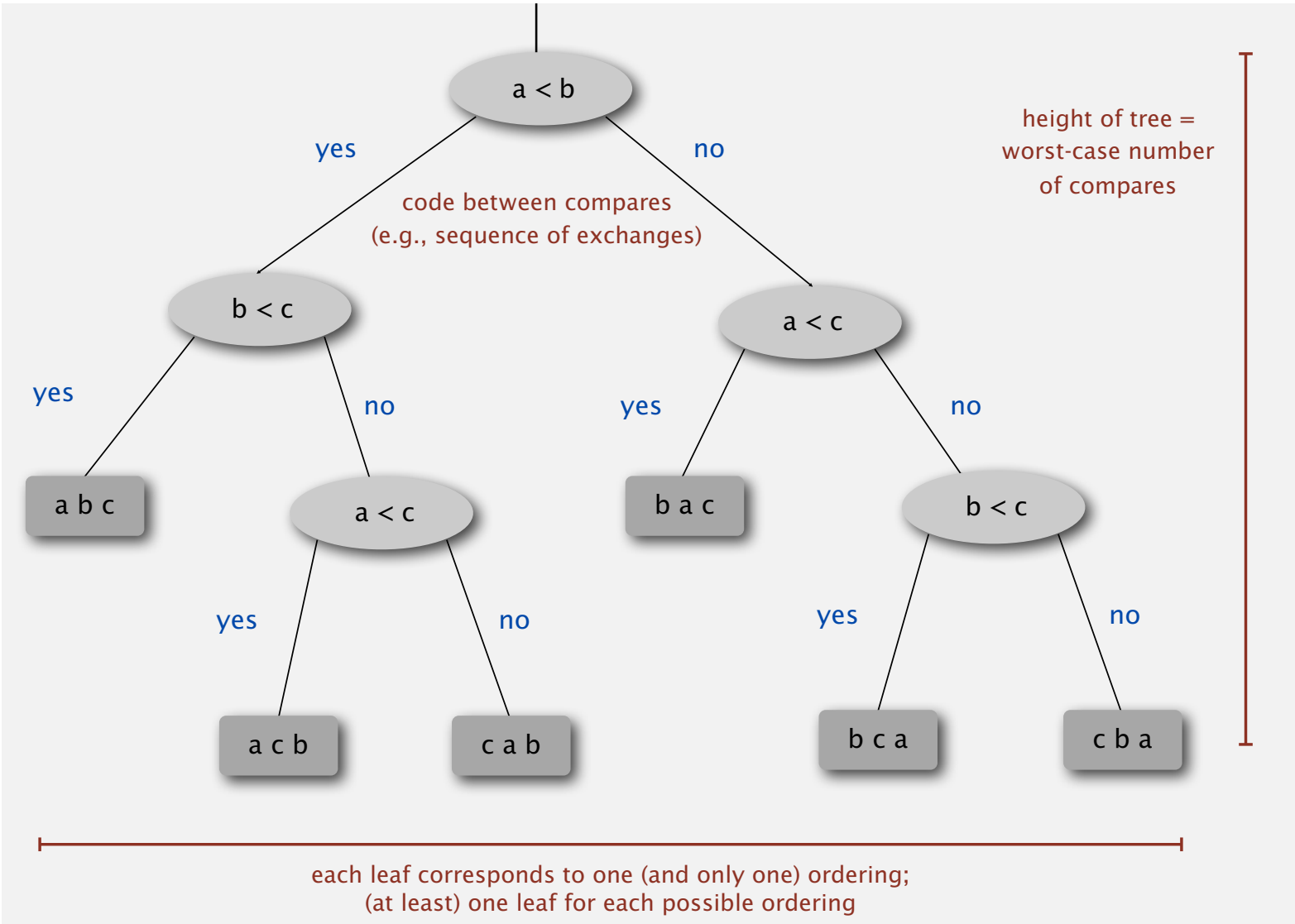


Mergesort: Bottom Up - Java

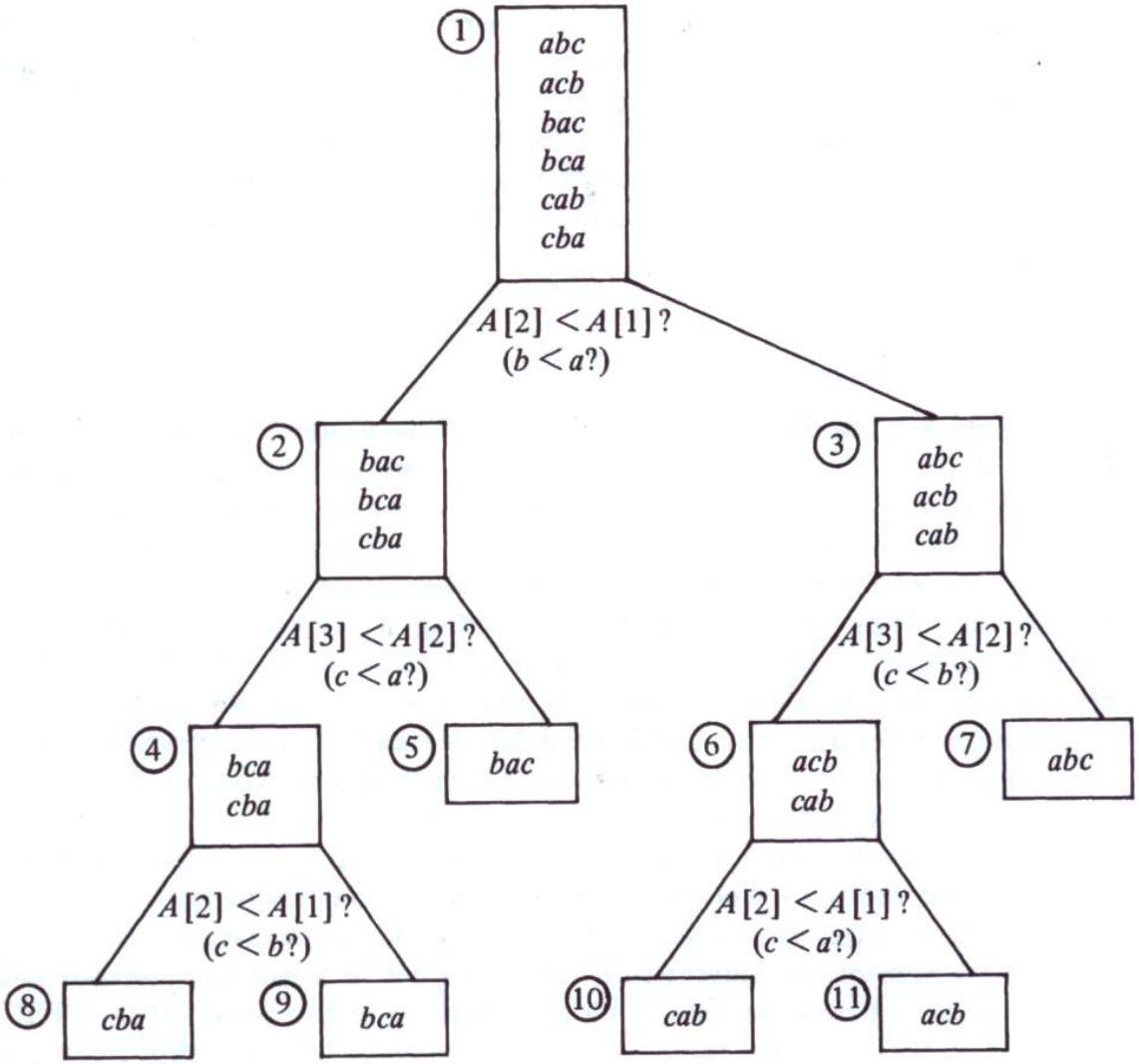
```
public class MergeBU
{
   private static void merge(...)
   {  /* as before */  }

   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));
   }
}
```

Lower Bound For Sorting: Decision Tree

For each particular sequence any sorting can be represented as a **decision tree.**(below for keys a, b, c, note that $3! = 6$).



For each particular sequence any sorting can be represented as a **decision tree.**(below for keys a, b, c, note that $3! = 6$).



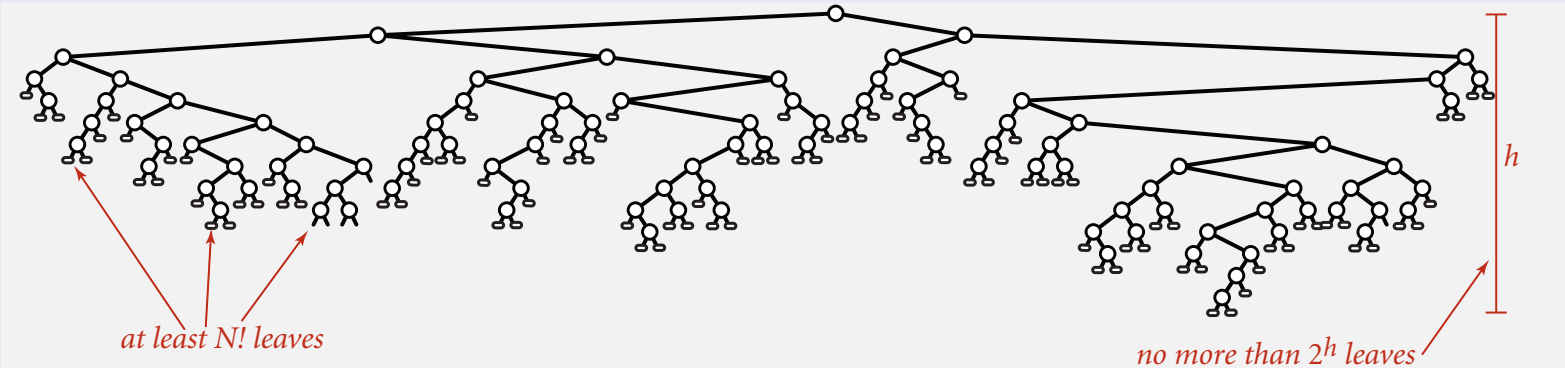
Lower Bound For Sorting

Proposition

Any compare-based sorting algorithm must use at least $\log(N!) \sim N \log N$ *compares in the worst-case, i.e.* $T(N) = \Omega(N \log N)$.

Proof. Sketch.

- 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 $\implies$ at least $N!$ leaves.



- $2^h \geq \log(N!) \implies h \geq \log(N!) \sim N \log(N)$.

Quick Sort: Ideas I (C. A. R. Hoare 1962)

```
Quicksort(S):
If |S| ≤ 3 then
   Sort S
   Output the sorted list
Else
   Choose a splitter a_i ∈ S uniformly at random
   For each element a_j of S

      Put a_j in S^- if a_j < a_i
      Put a_j in S^+ if a_j > a_i
   Endfor
   Recursively call Quicksort(S^-) and Quicksort(S^+)
   Output the sorted set S^-, then a_i, then the sorted set S^+
Endif
```

- For this algorithm, splitter is called *pivot*.

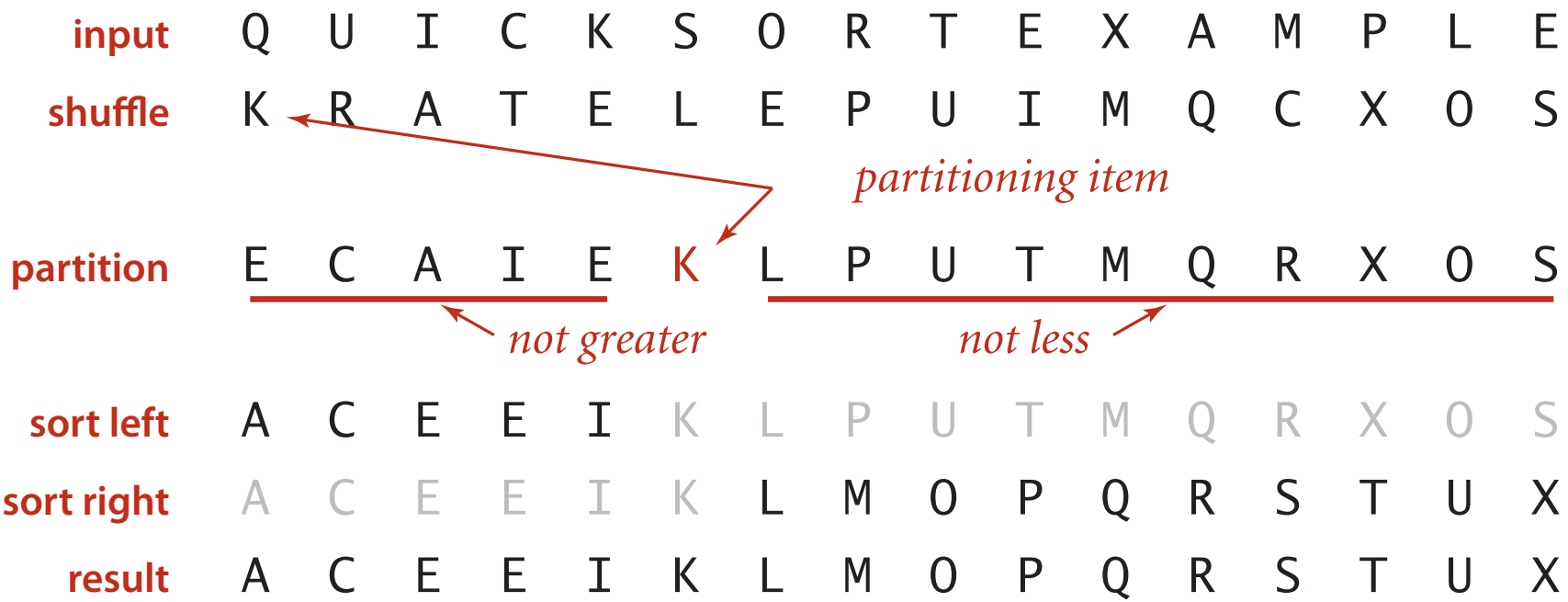
Quick Sort: Ideas II

Step 1. Shuffle the array.

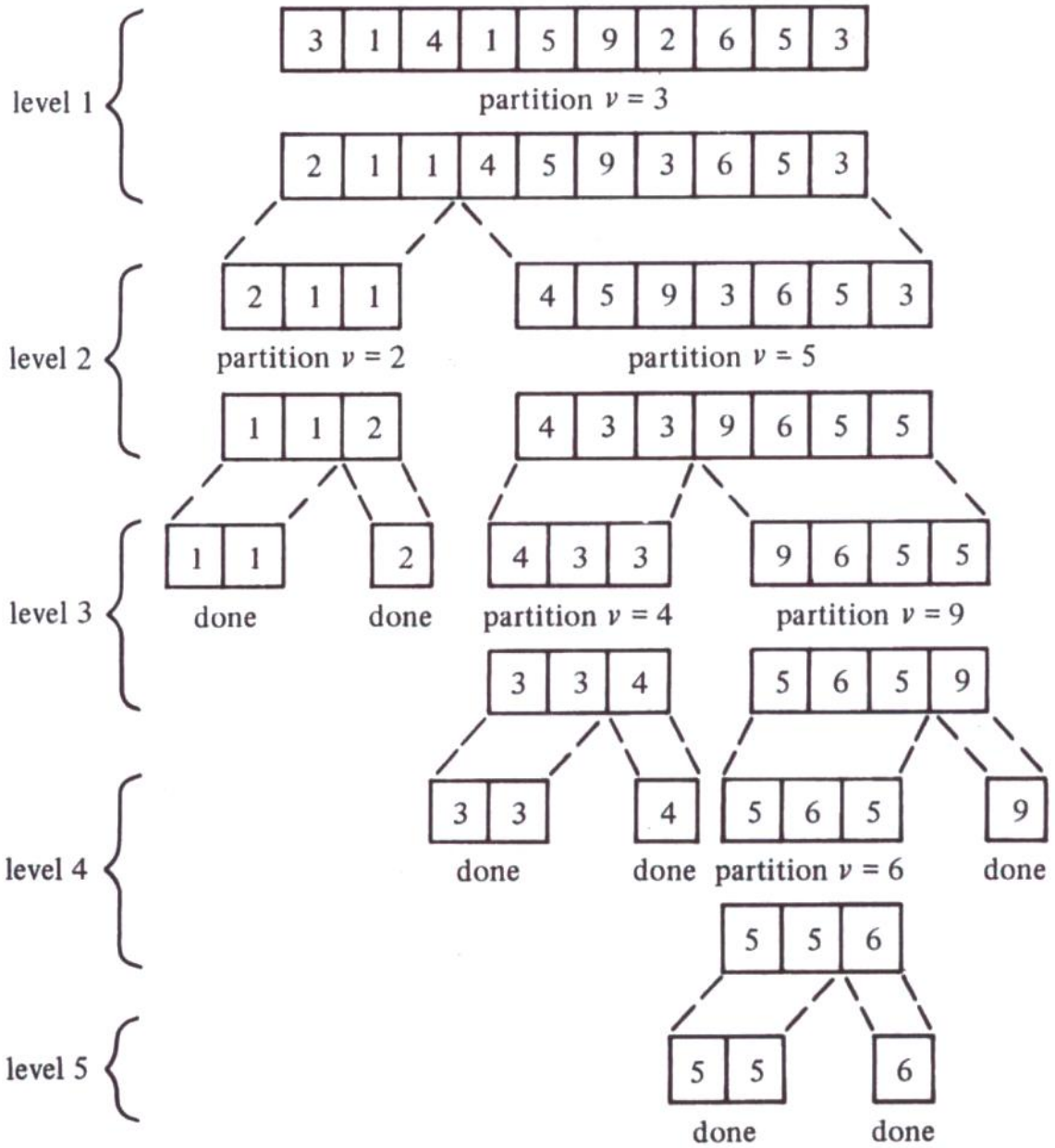
Step 2. Partition the array so that, for some j

- Entry $a[j]$ is in place.
- No larger entry to the left of j.
- No smaller entry to the right of j.

Step 3. Sort each subarray recursively.



Quick Sort: Ideas III



Quick Sort

- **Quick Sort** is considered the fastest sorting when input is random. WHY?
- Because it allows a very efficient implementation which is superfast for half of cases!
- Less abstract formulation:

Input: unsorted array $A[1..n]$
Output: sorted array $A[1..n]$

We want to use **only one** array.

- **QUICK SORT Algorithm:**

QuickSort($A[1..n]$)
  $v \leftarrow PIVOT(A[1..n])$ $\leftarrow$ it chooses the splitter
  $k \leftarrow PARTITION(v, A[1..n])$ $\leftarrow$ it maintains S^- and S^+ in A.
QuickSort($A[1..k-1]$)
QuickSort($A[k..n]$)

Pivot

How to implement: $v \leftarrow PIVOT(A[1..n])$?

- $v \leftarrow Random(A[1..n])$ $\leftarrow$ not feasible in full
- $v \leftarrow A[n]$ $\leftarrow$ as good as random if input has uniform distribution
- $v \leftarrow$ largest from the first two different values of $A[i]$, starting from $A[1]$ $\leftarrow$ original choice of C. A. R. Hoare, slightly better if distribution not entirely uniform
- $v \leftarrow$ median, if it can be approximated in advance
- $v \leftarrow$ a formula that depends on input distribution, if known.

Pivot

- Consider the case $v = A[n]$ (*simplest*)
- We may now write:

QuickSort(A, p, r)
IF $p < r$ THEN
  $q \leftarrow PARTITION(A, p, r)$ $\leftarrow$ must guarantee $p = r$ at some point
  QuickSort$(A, p, q-1)$
  QuickSort$(A, q+1, n)$
ENDIF

- It is use as QuickSort$(A, 1, n)$ for $A[1.n]$.

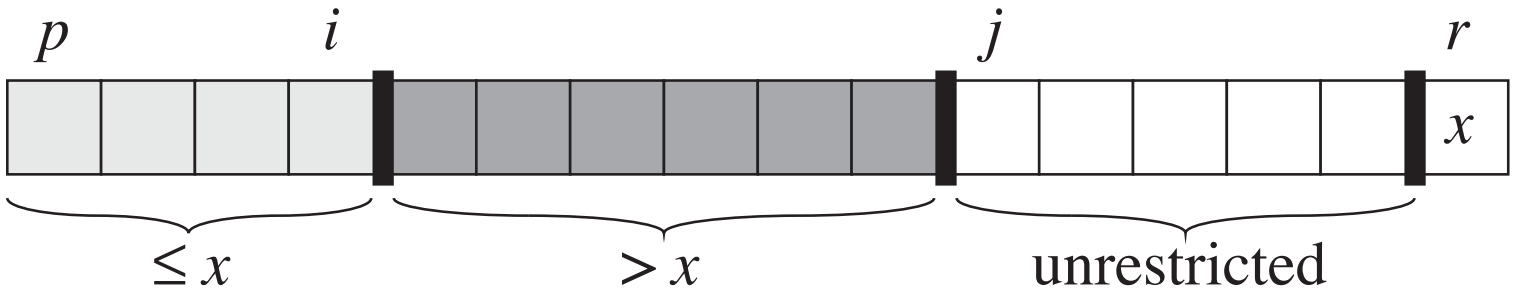
Partition

- **Partition is tricky and important.**

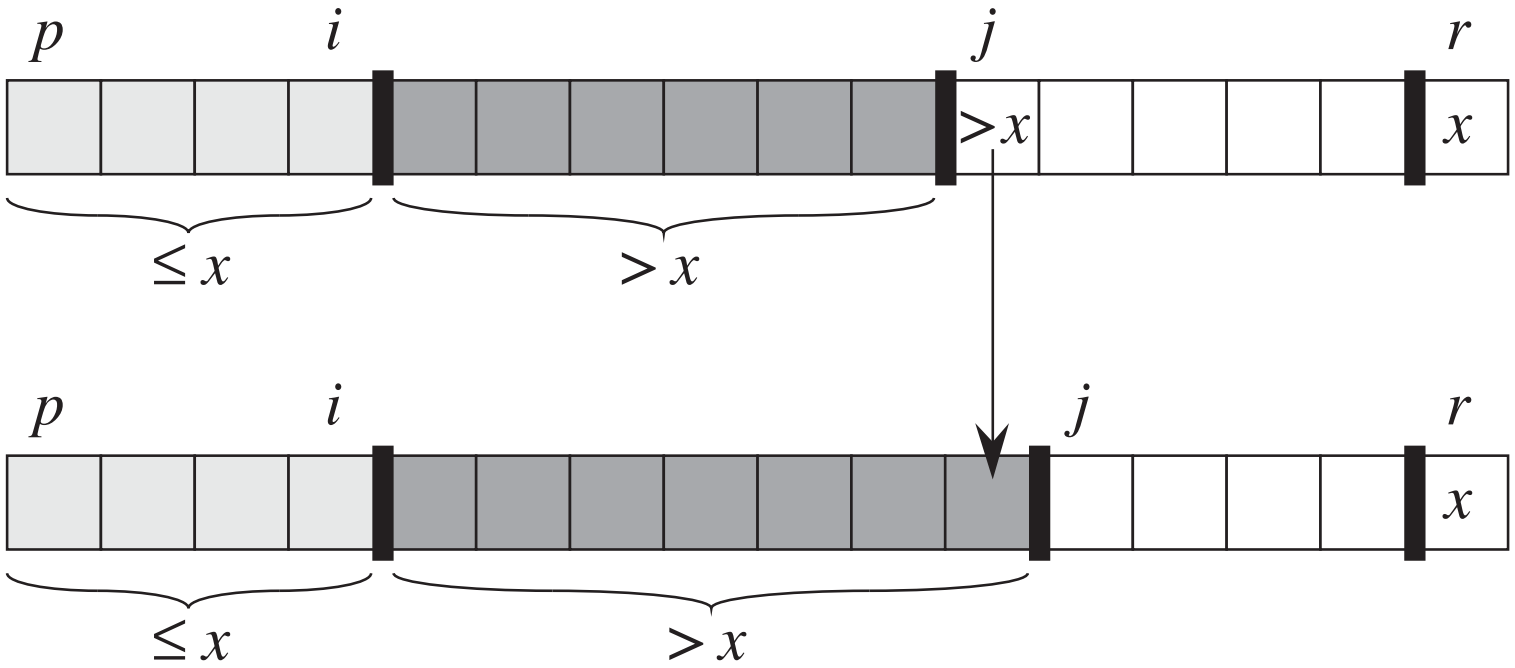
$PARTITION(A, p, r)$
$x \leftarrow A[r]$
$i \leftarrow p - 1$
FOR $j = p$ TO $r - 1$ DO
  IF $A[j] \leq x$ THEN
    BEGIN
      $i \leftarrow i + 1$
      $swap(A[i], A[j])$
    END
  ENDIF
ENDFOR
$swap(A[i + 1], [A[r])$
$return(i + 1)$

nothing if $A[j] > x$.
It happens in about $\frac{1}{2}$ cases!
Very efficient

- The procedure *Partition* maintains 4 regions on a subarray $A[p..r]$

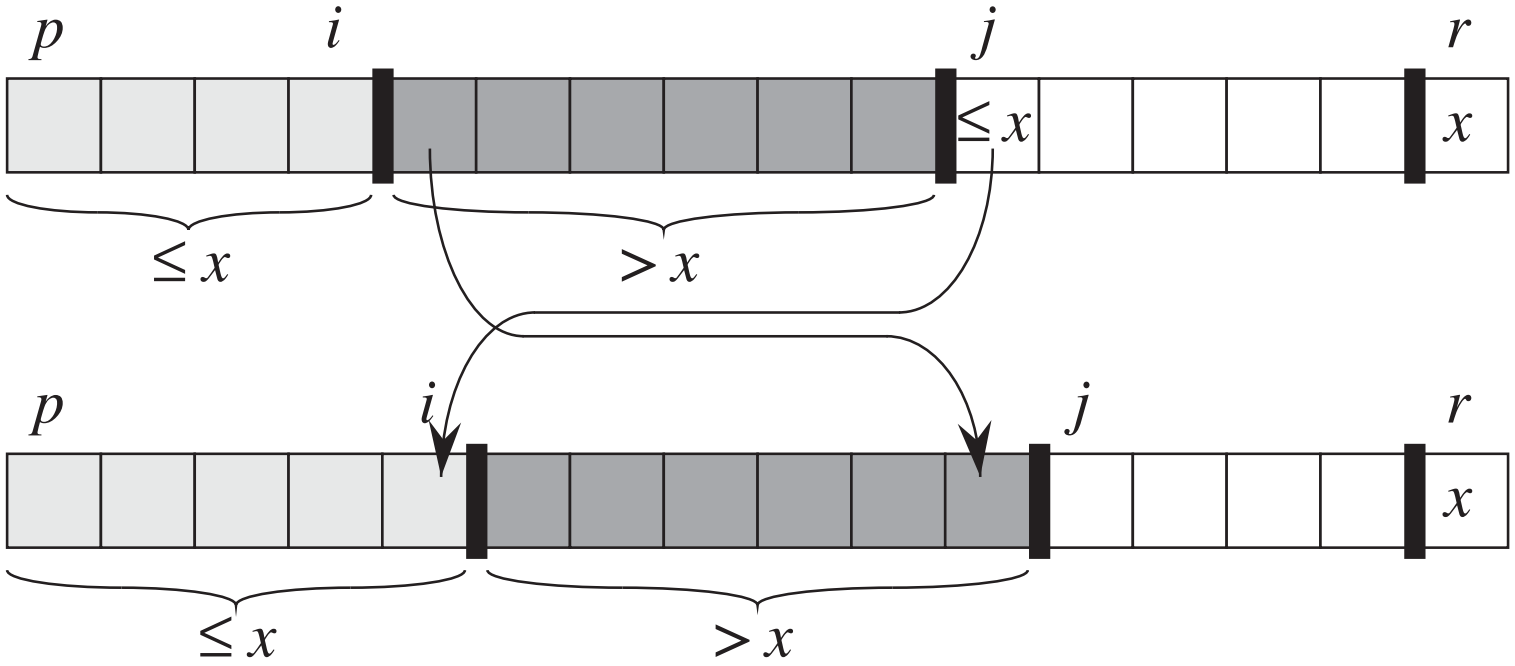


- Swapping when $A[j] > x$:



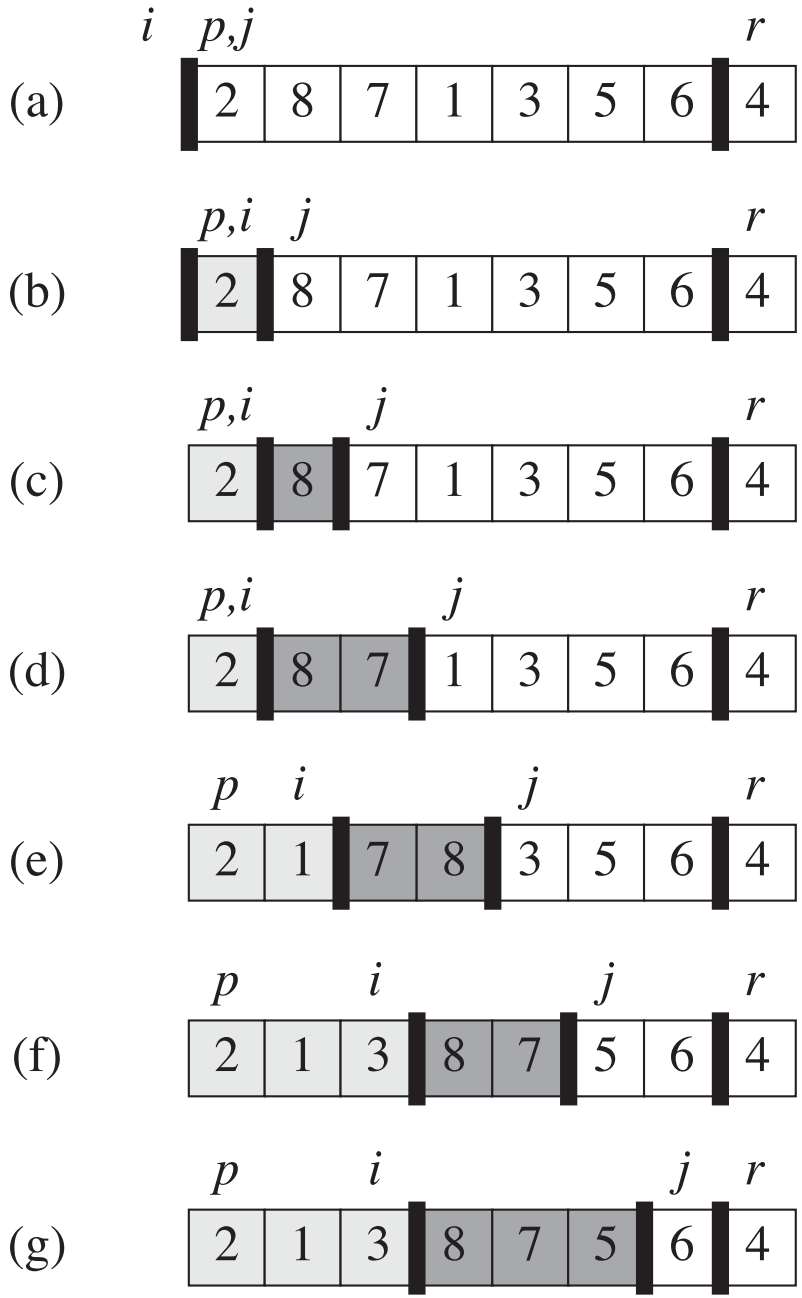
- **The only action is $j \leftarrow j + 1$!**

- Swapping when $A[j] \leq x$:



- i is incremented
- $A[i]$ and $A[j]$ are swapped
- j is incremented

Example


i p,j r
(a) 2 8 7 1 3 5 6 4
p,i j r
(b) 2 8 7 1 3 5 6 4
p,i j r
(c) 2 8 7 1 3 5 6 4
p,i j r
(d) 2 8 7 1 3 5 6 4
p i j r
(e) 2 1 7 8 3 5 6 4
p i j r
(f) 2 1 3 8 7 5 6 4
p i j r
(g) 2 1 3 8 7 5 6 4

Partition: Java

```
private static int partition(Comparable[] a, int lo, int hi)
{
   int i = lo, j = hi+1;
   while (true)
   {
      while (less(a[++i], a[lo]))
         if (i == hi) break;

      while (less(a[lo], a[--j]))
         if (j == lo) break;

      if (i >= j) break;
      exch(a, i, j);
   }

   exch(a, lo, j);
   return j;
}
```

find item on left to swap

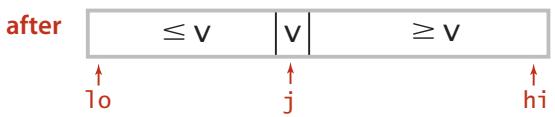
find item on right to swap

check if pointers cross

swap

swap with partitioning item

return index of item now known to be in place

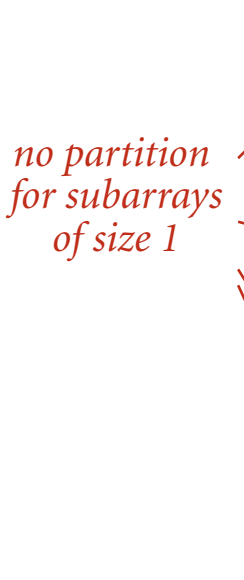


Quicksort: Java

```
public class Quick
{
   private static int partition(Comparable[] a, int lo, int hi)
   {  /* see previous slide */  }

   public static void sort(Comparable[] a)
   {
      StdRandom.shuffle(a);
      sort(a, 0, a.length - 1);
   }

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

shuffle needed for performance guarantee (stay tuned)

Quicksort: Trace



	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort trace (array contents after each partition)

Quicksort: Time Complexity

- Best case. Number of compares is $\sim n \log n$, i.e. $\Omega(n \log n)$.
- Worst case. Number of compares is $\sim 1/2 n^2$, i.e. $O(n^2)$.
- Average case. Expected number of compares is $\sim 1.39 n \log n$, i.e. $\Theta(n \log n)$.
  - 39% more compares than *mergesort*.
  - Faster than *mergesort* in practice because of less data movement (see page 33).

Quicksort: Practical Improvements

Insertion sort small subarrays.

- Even *quicksort* has too much overhead for tiny subarrays.
- Cutoff to insertion sort for $\approx$ 10 items.

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