

# Lecture 13

## Sorting Algorithm

Department of Computer Science  
Hofstra University

# Lecture Goals

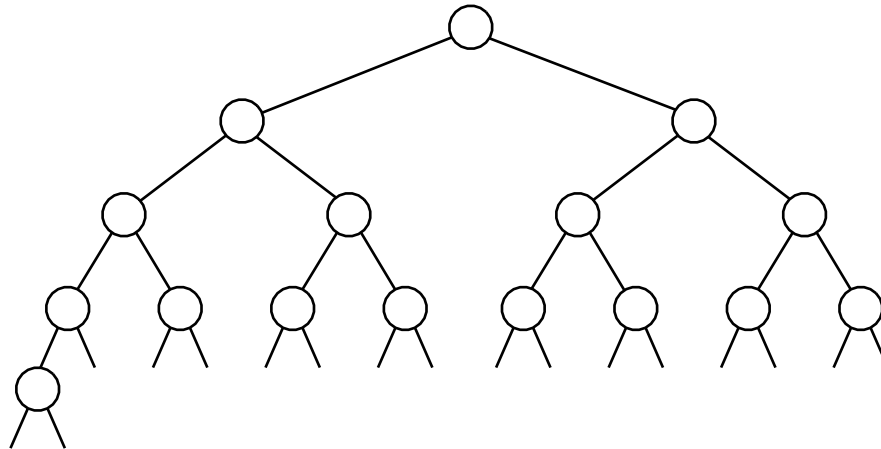
- We introduce *binary heap* for priority queue data abstract, which leads to an efficient sorting algorithm known as *heapsort*.
- We introduce and implement the *randomized quicksort* algorithm and analyze its performance.
- We study the *mergesort* algorithm and show that it guarantees to sort any array of  $n$  items with at most  $n \log(n)$  compares.

# Short Videos of Sorting Algorithms

- Sort Algos // Michael Sambol Michael Sambol
  - [https://www.youtube.com/playlist?list=PL9xmBV\\_5YoZOZSbGAXAP\\_Iq1BeUf4j20pl](https://www.youtube.com/playlist?list=PL9xmBV_5YoZOZSbGAXAP_Iq1BeUf4j20pl)
  - Merge Sort, Quick Sort, Bubble Sort, Insertion Sort, Selection Sort, Heap Sort
- 10 Sorting Algorithms Easily Explained
  - <https://www.youtube.com/watch?v=rbbTd-gkajw>
  - Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Counting Sort, Shell Sort, Tim Sort, Radix Sort

# Heapsort: Binary Heap

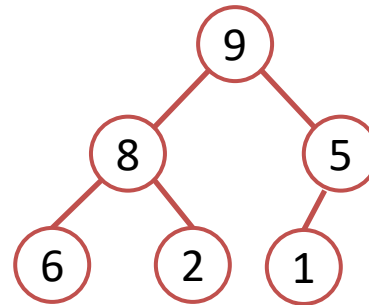
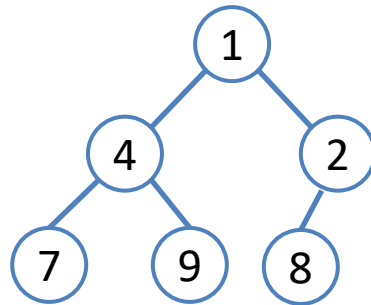
- In a **heap** the highest (or lowest) priority element is always stored at the root, hence the name "heap". A heap is useful data structure when you need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a **priority queue** and **heapsort**.
- A **binary heap** is a complete binary tree which is an efficient data structure satisfies the heap ordering property.
- In a *complete tree*, every level (except possibly the last) is completely filled; the last level is filled from left to right.



complete binary tree with  $n = 16$  nodes (height = 4)

# Heapsort: Binary Heap

- The heap ordering can be one of two types:
- The **min-heap property**: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
- The **max-heap property**: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.



in this lecture  
↙

- A heap is not a sorted structure and can be regarded as partially ordered. As you see from the picture, there is no particular relationship among nodes on any given level, even among the siblings.
- Since a heap is a complete binary tree, it has a smallest possible height - a heap with  $N$  nodes always has  $O(\log N)$  height.

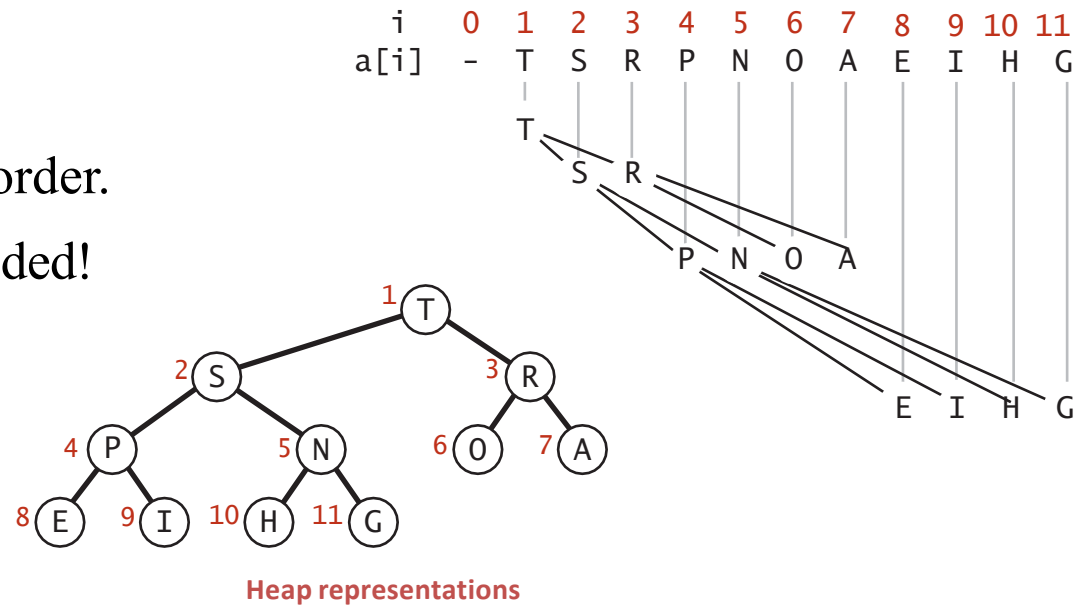
What Is a Binary Heap?

<https://www.youtube.com/watch?v=AE5I0xACpZs>

# Binary Heap: Array Representation

## Array representation.

- Indices start at 1.
- Take nodes in **level** order.
- No explicit links needed!



**Proposition.** Largest key is  $a[1]$ , which is root of binary tree.

**Proposition.** Can use array indices to move through tree.

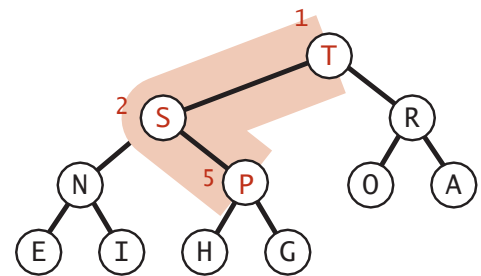
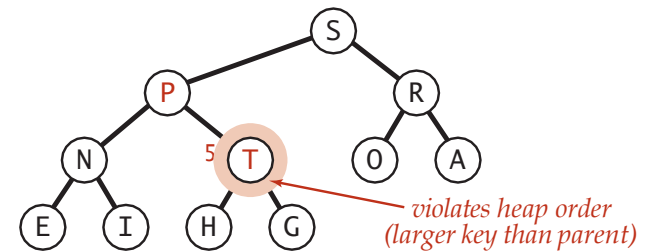
- Parent of node at  $k$  is at  $k/2$ .
- Children of node at  $k$  are at  $2k$  and  $2k+1$ .

# Binary Heap Operations: Promotion

- **Scenario.** A key becomes **larger** than its parent's key.
- **To eliminate the violation:**
- Exchange key in child with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
    while (k > 1 && less(k/2, k))
    {
        exch(k, k/2);
        k = k/2;
    }
}
```

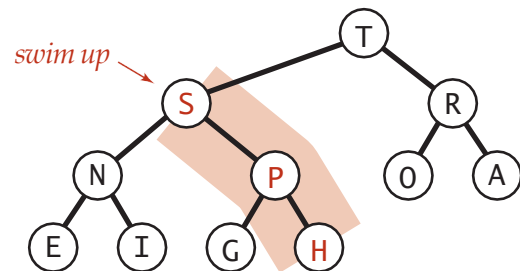
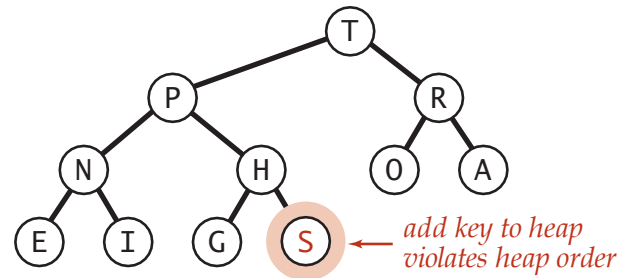
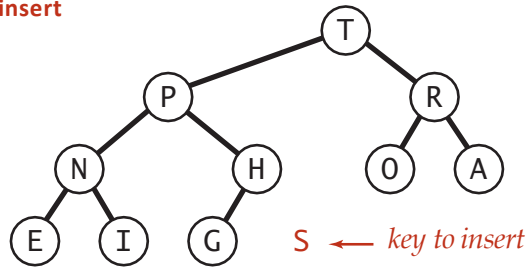
parent of node at k is at k/2



# Binary Heap Operations: Insert

- **Insert.** Add node at end, then **swim** it up.
- **Cost.** At most  $1 + \lg n$  compares.

insert



```
public void insert(Key x)
{
    pq[++n] = x;
    swim(n);
}
```



# Binary Heap Operations: Demotion

- **Scenario.** A key becomes **smaller** than one (or both) of its children's.
- **To eliminate the violation:**
- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

```
private void sink(int k)
```

```
{
```

```
    while (2*k <= n)
```

```
    {
```

```
        int j = 2*k;
```

```
        if (j < n && less(j, j+1)) j++;
```

```
        if (!less(k, j)) break;
```

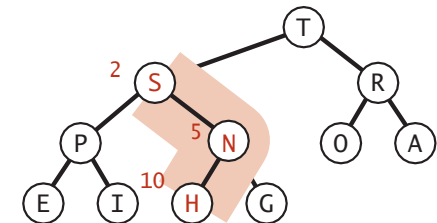
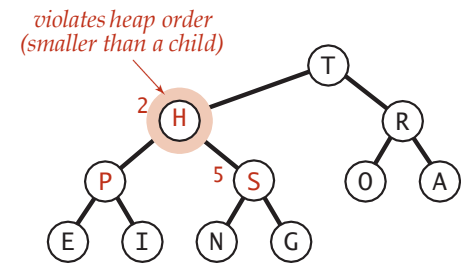
```
        exch(k, j);
```

```
        k = j;
```

```
    }
```

```
}
```

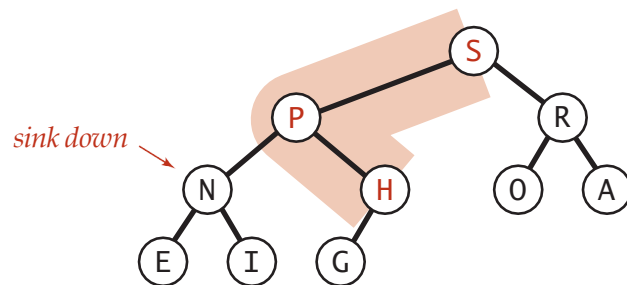
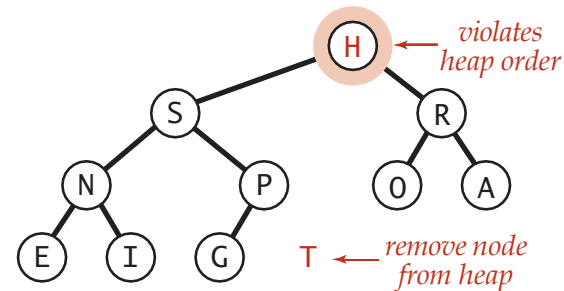
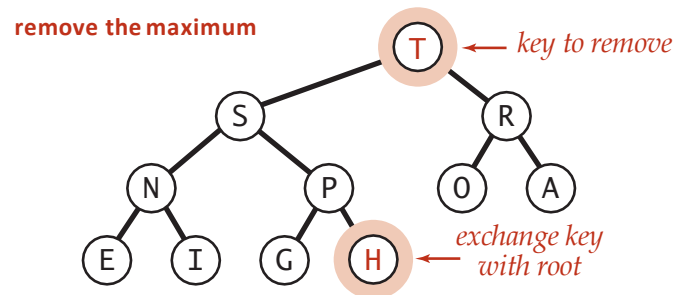
children of node at k  
are  $2*k$  and  $2*k+1$



Top-down reheapify (sink)

# Binary Heap Operations: DeleteMax

- **Delete max.** Exchange root with node at end, then **sink** it down.
- **Cost.** At most  $2 \lg(n)$  compares.

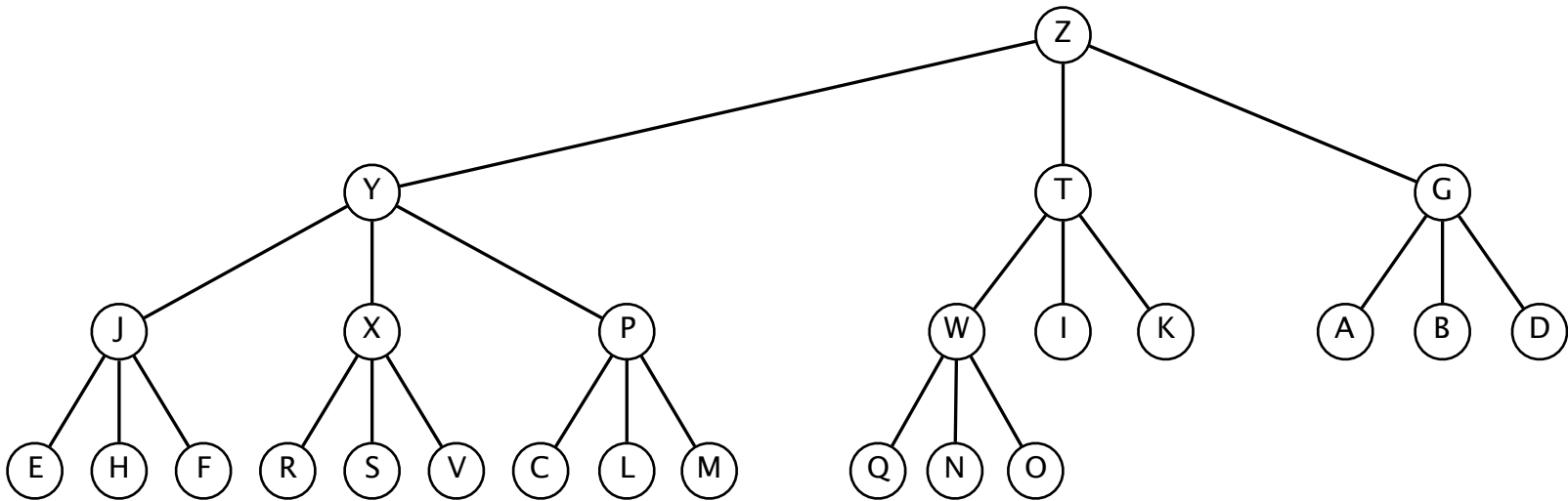


```
public Key delMax()
{
    Key max = pq[1];
    exch(1, n--);
    sink(1);
    pq[n+1] = null;
    return max;
}
```

prevent loitering

# Binary Heap: Practical improvements

- **Multiway heaps.** Complete d-way tree.
- Parent's key no smaller than its children's keys.
- Fact. Height of complete d-way tree on n nodes is  $\sim \log_d n$ .

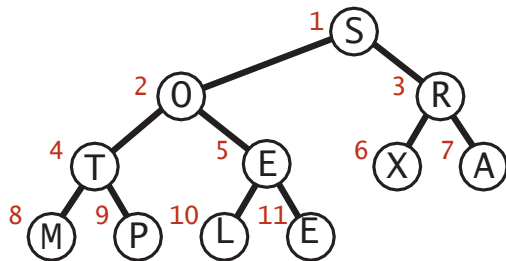


3-way heap

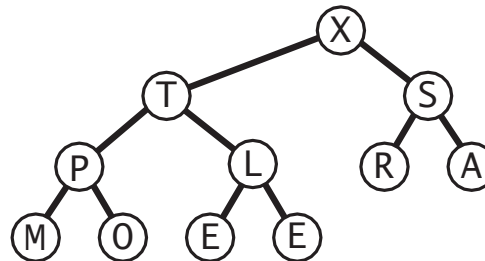
# Heapsort Algorithm

- Basic plan for in-place sort.
- View input array as a complete binary tree.
- **Heap construction**: build a max-heap with all n keys.
- **Sortdown**: repeatedly remove the maximum key.

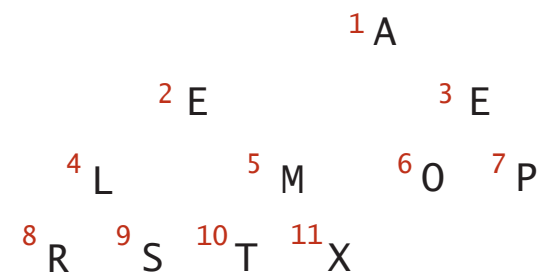
keys in arbitrary order



build max heap  
(in place)



sorted result  
(in place)



1	2	3	4	5	6	7	8	9	10	11
S	O	R	T	E	X	A	M	P	L	E

1	2	3	4	5	6	7	8	9	10	11
X	T	S	P	L	R	A	M	O	E	E

1	2	3	4	5	6	7	8	9	10	11
A	E	E	L	M	O	P	R	S	T	X

Heaps and Heap Sort

<https://www.youtube.com/watch?v=H5kAcmGOn4Q>

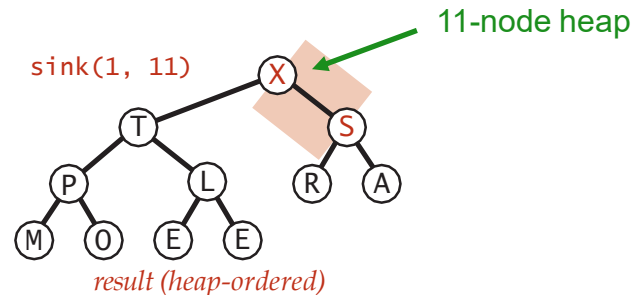
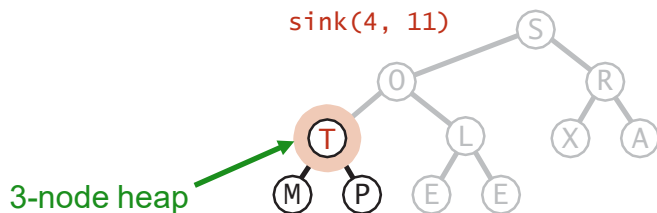
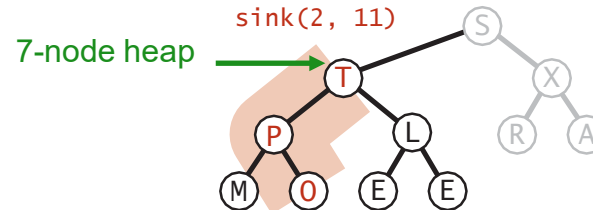
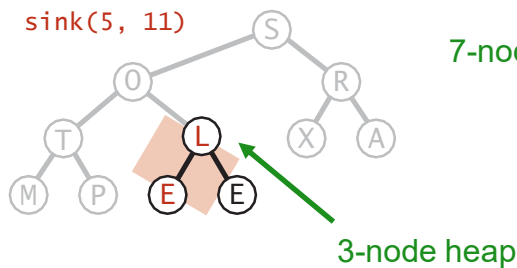
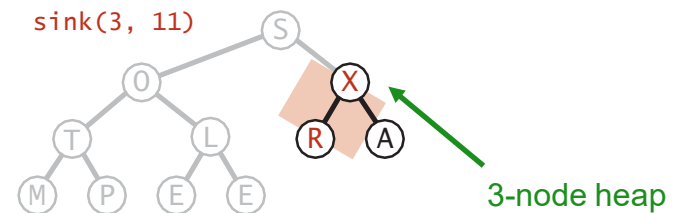
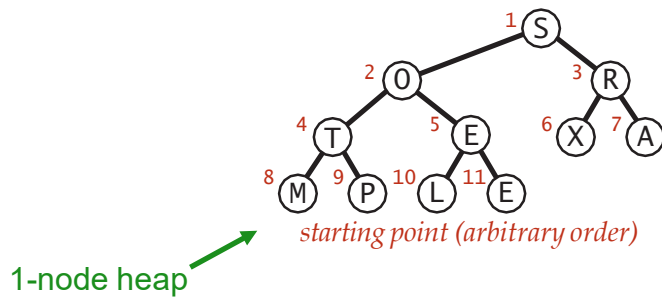
HEAP SORT | Sorting Algorithms | DSA |  
GeeksforGeeks

[https://www.youtube.com/watch?v=MtQL\\_ll5KhQ](https://www.youtube.com/watch?v=MtQL_ll5KhQ)

# Heapsort: Heap Construction

First pass. Build heap using bottom-up method.

```
for (int k = n/2; k >= 1; k--)  
    sink(a, k, n);
```

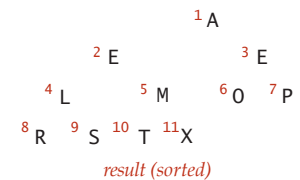
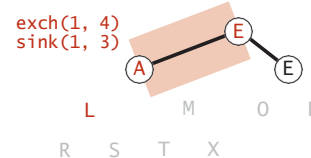
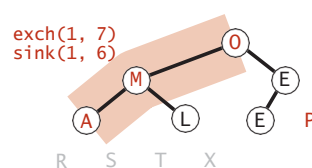
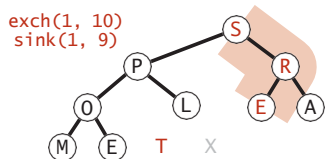
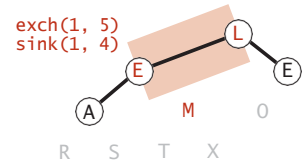
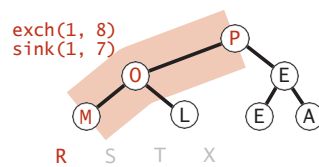
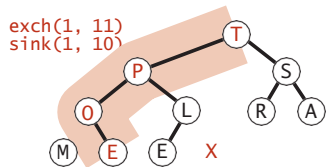
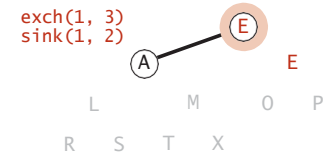
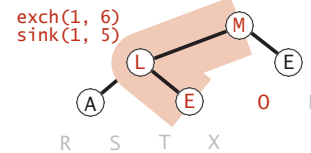
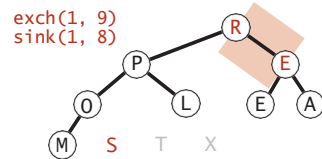
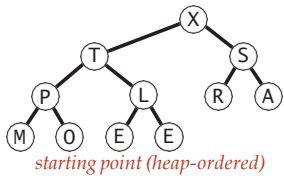


# Heapsort: Sortdown

## Second pass.

- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.

```
while (n > 1)
{
    exch(a, 1, n--);
    sink(a, 1, n);
}
```



# Heapsort: Java Implementation

```
public class Heap
{
    public static void sort(Comparable[] a)
    {
        int n = a.length;
        for (int k = n/2; k >= 1; k--)
            sink(a, k, n);
        while (n > 1)
        {
            exch(a, 1, n);
            sink(a, 1, --n);
        }
    }
}
```

$O(n \log n)$

```
private static void sink(Comparable[] a, int k, int n)
{ /* as before */ }
```

← but make static (and pass arguments)

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

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

← but convert from 1-based indexing to 0-base indexing

```
}
```

# Heapsort: Trace

sink(K, N)

a[i]

N	k	0	1	2	3	4	5	6	7	8	9	10	11
<i>initial values</i>			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
<i>heap-ordered</i>			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X
<i>sorted result</i>			A	E	E	L	M	O	P	R	S	T	X

3-node heap

7-node heap

11-node heap

red: exchanged

black: compared

Heapsort trace (array contents just after each sink)

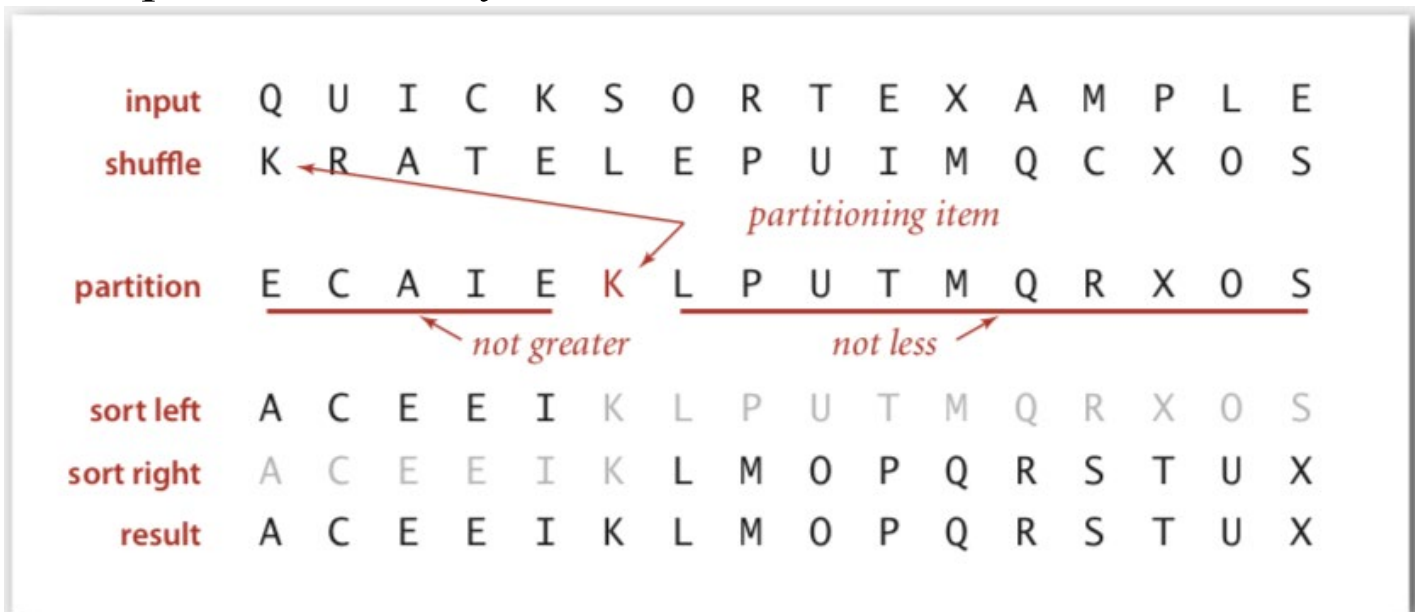


# Quicksort

- Quicksort is a widely used sorting algorithm based on the divide-and-conquer approach.
- ## Pivot Selection in Quicksort
- The pivot is an element chosen from the array that serves as a reference point for partitioning the array into two subarrays. There are several strategies for selecting the pivot:
  - #### First or Last Element as Pivot. One common approach is to choose the last element of the array as the pivot. This method is simple to implement but can lead to poor performance if the array is already sorted or nearly sorted.
  - #### Random Element as Pivot. Selecting a random element as the pivot can help avoid worst-case scenarios and provide more consistent performance across different input distributions.
  - #### Median-of-Three. This method selects the median of the first, middle, and last elements of the array as the pivot. It often provides a good balance between simplicity and performance.
- ## Partitioning Process
- Once the pivot is selected, the partitioning process begins:
  - 1. The pivot is compared with each element in the array.
  - 2. Elements smaller than the pivot are moved to its left.
  - 3. Elements larger than the pivot are moved to its right.
  - 4. The pivot is placed in its final sorted position.

# Quicksort

1. **Shuffle** the array.
2. **Partition** so that, for some pivot  $j$ 
  - entry  $a[j]$  is in place
  - no larger entry to the left of  $j$
  - no smaller entry to the right of  $j$
3. **Sort** each piece recursively.



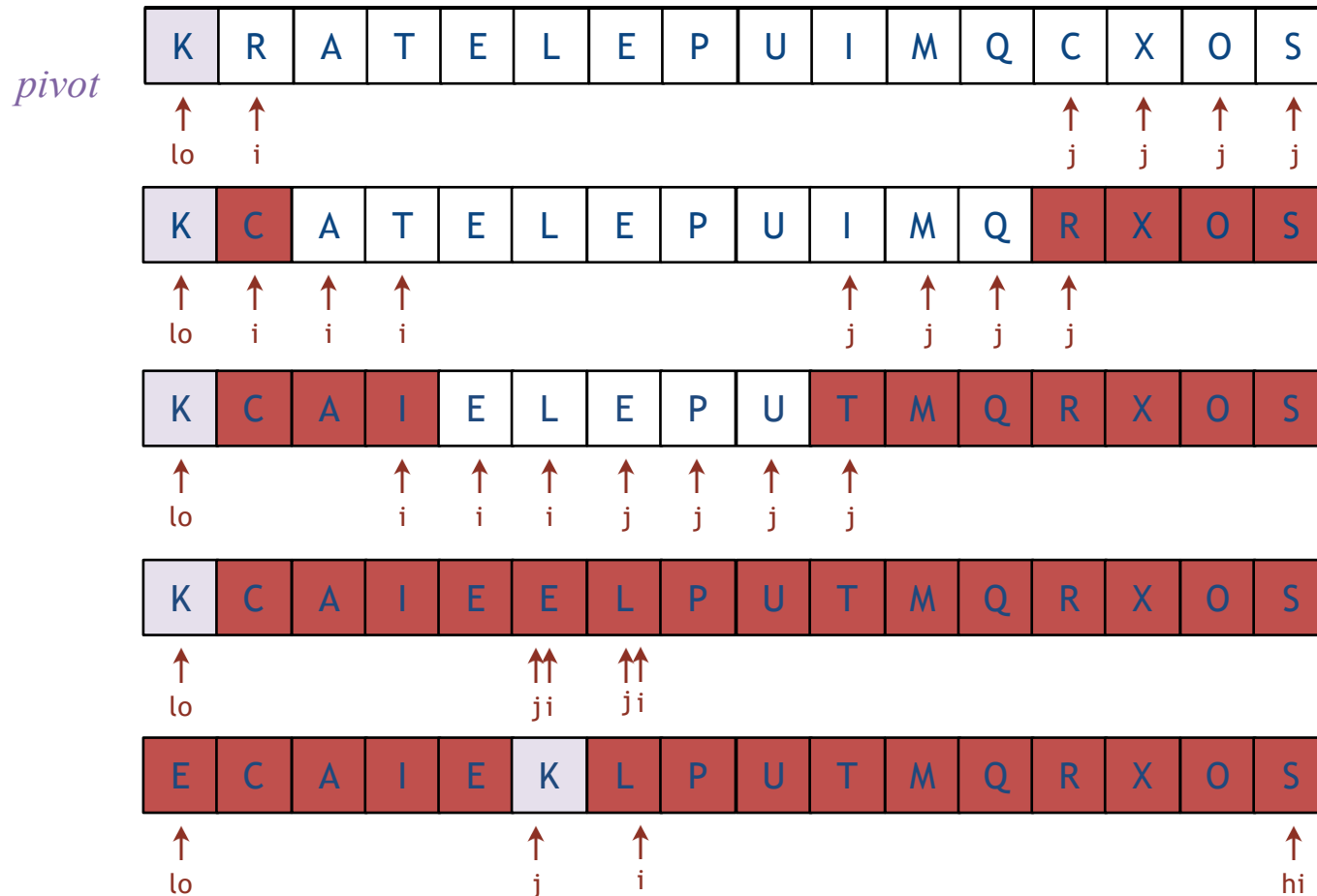
# Partition Operation

Repeat until  $i$  and  $j$  pointers cross.

- **Scan**  $i$  from left to right so long as  $(a[i] < a[lo])$ .
- **Scan**  $j$  from right to left so long as  $(a[j] > a[lo])$ .
- **Exchange**  $a[i]$  with  $a[j]$ .

When pointers cross.

- **Exchange**  $a[lo]$  with  $a[j]$ .



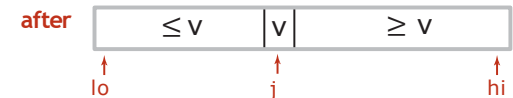
# Partition Operation: Java Implementation

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

        while (less(a[lo], a[--j]))           find item on right to swap
            if (j == lo) break;

        if (i >= j) break;                    check if pointers cross
        exch(a, i, j);                        swap

        exch(a, lo, j);                      swap with partitioning item
        return j;                            return index of item now known to be in place
    }
}
```



# Quicksort: Java Implementation

```
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

← Pivot selection

# Quicksort: Trace

	lo	j	hi	<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>	<u>15</u>
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

no partition  
for subarrays  
of size 1

Quicksort trace (array contents after each partition)

# Quicksort: Best-case Analysis

Best case. Number of compares is  $\sim N \log N$ .

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

# Quicksort: Worst-case Analysis

Worst case. Number of compares is  $\sim \frac{1}{2} N^2$ .

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O



# Quicksort: Practical Improvements

## Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for  $\approx 10$  items.
- Note: could delay insertion sort until one pass at end.

```
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);
}
```

# Quicksort: Summary of Performance Characteristics

**Worst case.** Number of compares is quadratic.

- $N + (N - 1) + (N - 2) + \dots + 1 \sim \frac{1}{2} N^2$ .
- More likely that your computer is struck by lightning bolt.

**Average case.** Number of compares is  $\sim 1.39 N \lg N$ .

- 39% more compares than mergesort.
- **But** faster than mergesort in practice because of less data movement.

**Random shuffle.**

- Probabilistic guarantee against worst case.
- Basis for math model that can be validated with experiments.

# Mergesort Algorithm

Basic plan.

1. Divide array into two halves.
2. Recursively sort each half.
3. 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

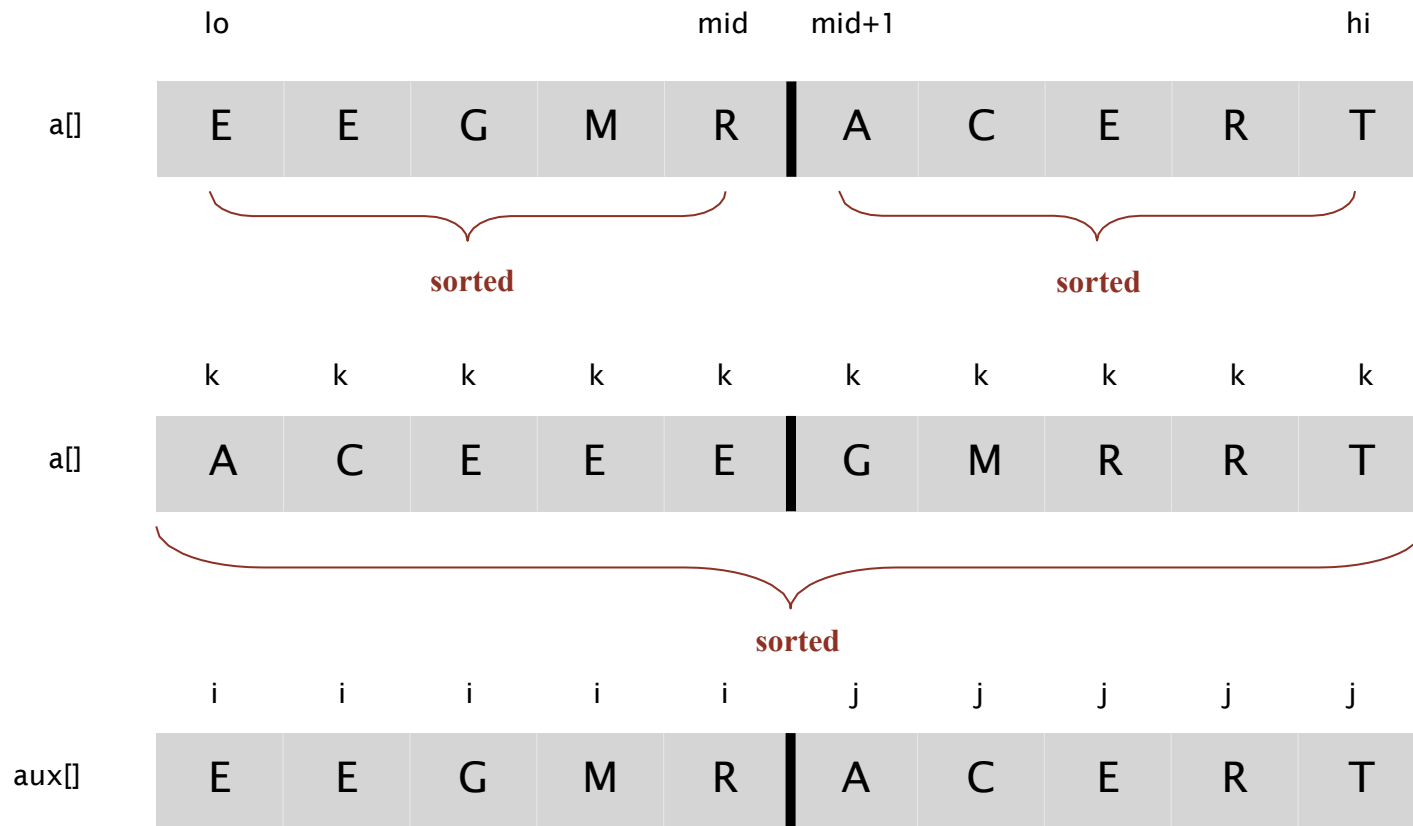
Mergesort overview

Merge Sort Algorithm: A Step-by-Step Visualization  
<https://www.youtube.com/watch?v=ho05egqcPl4>

Merge Sort vs Quick Sort  
<https://www.youtube.com/watch?v=es2T6KY45cA>

# Merge Operation

**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

# Merge Operation: 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



Can enable or disable at runtime.

⇒ No cost in production code.

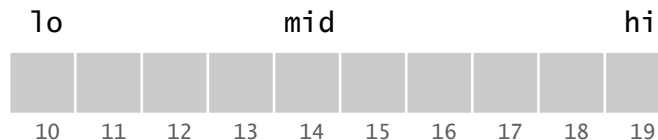
```
java -ea MyProgram // enable assertions
java -da MyProgram // disable assertions (default)
```

# 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)
    {
        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, aux, <sup>lo</sup> 0, 0, <sup>hi</sup> 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: Practical Improvement

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for  $\approx 7$  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 Improvement

Stop if already sorted.

- Is biggest item in first half  $\leq$  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);
}
```

# Bottom-up Mergesort

## Basic plan.

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

Simple and non-recursive version of mergesort. but about 10% slower than recursive, **top-down** mergesort on typical systems

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

# Stability of Sorting Algorithms

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

i	min	0	1	2
0	2	B <sub>1</sub>	B <sub>2</sub>	A
1	1	A	B <sub>2</sub>	B <sub>1</sub>
2	2	A	B <sub>2</sub>	B <sub>1</sub>
		A	B <sub>2</sub>	B <sub>1</sub>

selectsort is not stable

i	j	0	1	2	3
		B <sub>1</sub>	C <sub>1</sub>	C <sub>2</sub>	A <sub>1</sub>
1	3	B <sub>1</sub>	C <sub>1</sub>	C <sub>2</sub>	A <sub>1</sub>
1	3	B <sub>1</sub>	A <sub>1</sub>	C <sub>2</sub>	C <sub>1</sub>
0	1	A <sub>1</sub>	B <sub>1</sub>	C <sub>2</sub>	C <sub>1</sub>

quicksort is not stable

i	j	0	1	2	3	4
0	0	B <sub>1</sub>	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	B <sub>2</sub>
1	0	A <sub>1</sub>	B <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	B <sub>2</sub>
2	1	A <sub>1</sub>	A <sub>2</sub>	B <sub>1</sub>	A <sub>3</sub>	B <sub>2</sub>
3	2	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	B <sub>1</sub>	B <sub>2</sub>
4	4	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	B <sub>1</sub>	B <sub>2</sub>
		A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	B <sub>1</sub>	B <sub>2</sub>

insertsort is stable

0	1	2	3	4	5	6	7	8	9	10
A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	B	D	A <sub>4</sub>	A <sub>5</sub>	C	E	F	G

mergesort is stable

0	1	2	3
B	A	C <sub>1</sub>	C <sub>2</sub>
B	C <sub>2</sub>	C <sub>1</sub>	A
C <sub>1</sub>	C <sub>2</sub>	B	A
C <sub>2</sub>	A	B	C <sub>1</sub>
B	A	C <sub>2</sub>	C <sub>1</sub>
A	B	C <sub>2</sub>	C <sub>1</sub>

heapsort is not stable

# 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
merge		✓	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	n log n guarantee; stable
quick	✓		$n \lg n$	$2 n \ln n$	$\frac{1}{2} n^2$	n log n probabilistic guarantee; fastest in practice
heap	✓		$3 n$	$2 n \lg n$	$2 n \lg n$	n log n guarantee; in-place
?	✓	✓	$n$	$n \lg n$	$n \lg n$	holy sorting grail

# Additional Resources

- Additional sorting algorithms (not covered in exam)