

Lecture 13

Sorting Algorithm

Department of Computer Science
Hofstra University

Lecture Goals

- We introduce simple sorting algorithms, incl. Bubble Sort, Selection Sort, and Insertion Sort, with high time complexity.
- We introduce *binary heap* for priority queue data abstract, which leads to an efficient sorting algorithm known as Heap Sort.
- We introduce the Quick Sort algorithm and analyze its performance.
- We introduce the Merge Sort algorithm and analyze its performance.

Bubble Sort

- Bubble Sort works by repeatedly swapping adjacent elements if they are in the wrong order.
 - We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.
 - In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.
- Time complexity: $O(n^2)$
- Bubble Sort | GeeksforGeeks
 - <https://www.geeksforgeeks.org/bubble-sort-algorithm/>
 - <https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/>
 - <https://www.youtube.com/watch?v=nmhjrI-aW5o>

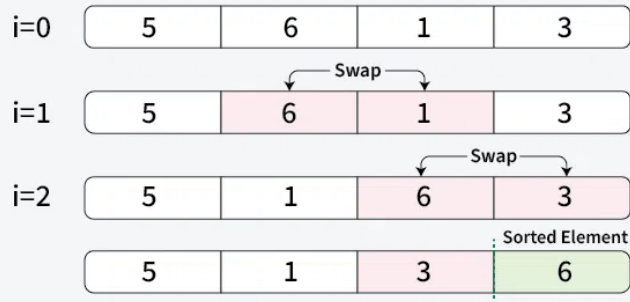
Bubble Sort Time Complexity

- The worst-case condition for bubble sort occurs when elements of the array are arranged in decreasing order. In the worst case, the total number of iterations or passes required to sort a given array is $(n-1)$, where n is the number of elements present in the array.
- **Pass 1:**
Number of comparisons = $(n-1)$, Number of swaps = $(n-1)$
- **Pass 2:**
Number of comparisons = $(n-2)$, Number of swaps = $(n-2)$
- ...
- **Pass $n-1$:**
Number of comparisons = 1
Number of swaps = 1
- *Total number of comparison required to sort the array*
$$= (n-1) + (n-2) + \dots + 2 + 1$$
$$= (n-1) * (n-1+1) / 2 \quad \{ \text{by using sum of } n \text{ natural Number formula} \}$$
$$= (n * (n-1)) / 2$$
- *In worst case, **Total number of swaps = Total number of comparison***
Total number of comparison (Worst case) = $n(n-1)/2$
Total number of swaps (Worst case) = $n(n-1)/2$
- *So worst case time complexity is $O(n^2)$*

Bubble Sort Example

01
Step

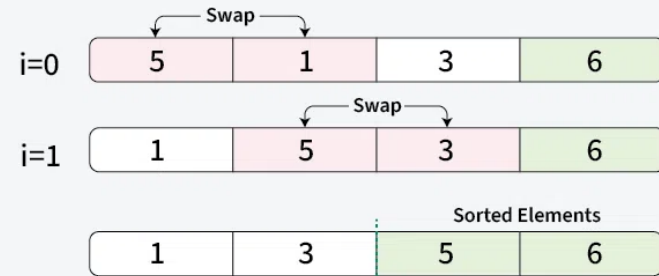
Placing the 1st largest element at its correct position



Bubble sort

02
Step

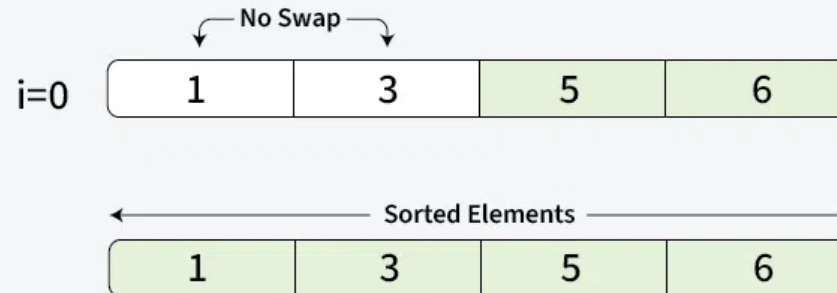
Placing 2nd largest element at its correct position



Bubble sort

03
Step

Placing 3rd largest element at its correct position



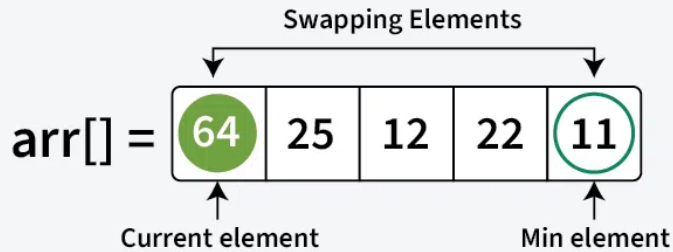
Bubble sort

Selection Sort

- Selection Sort works by repeatedly selecting the smallest element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.
 - First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
 - Then we find the smallest among remaining elements (or second smallest) and move it to its correct position by swapping.
 - Keep going until all elements are sorted.
- Time complexity: $O(n^2)$, as there are two nested loops:
 - Outer loop to select each element one by one with $O(n)$ complexity
 - Inner loop to compare that element with every other element with $O(n)$ complexity
- Selection Sort | GeeksforGeeks
 - <https://www.geeksforgeeks.org/selection-sort-algorithm-2/>
 - <https://www.youtube.com/watch?v=xWBP4lzkoyM>

01
Step

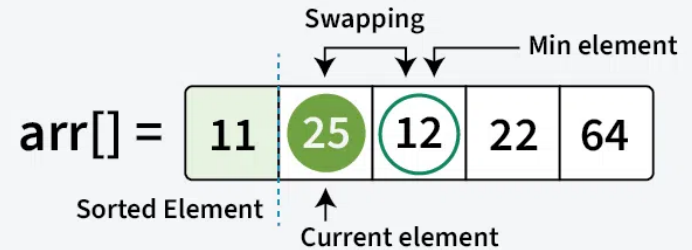
Start from the first element at index 0, find the smallest element in the rest of the array which is unsorted, and swap (11) with current element(64).



Selection Sort Algorithm

02
Step

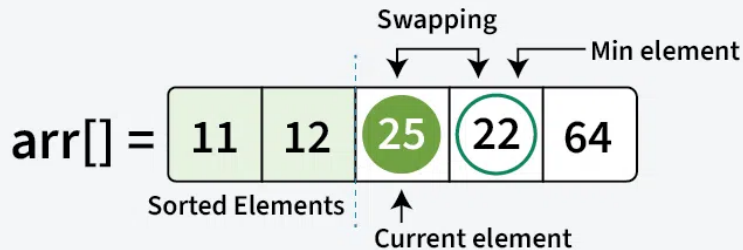
Move to the next element at index 1 (25). Find the smallest in unsorted subarray, and swap (12) with current element (25).



Selection Sort Algorithm

03
Step

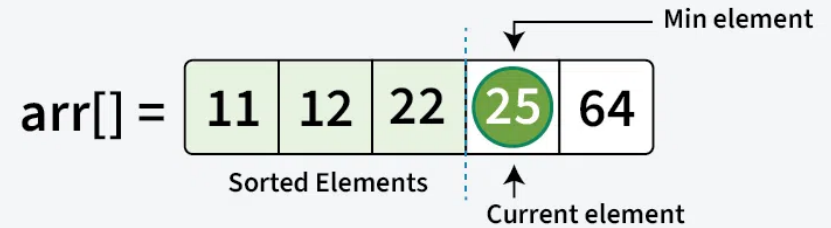
Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).



Selection Sort Algorithm

04
Step

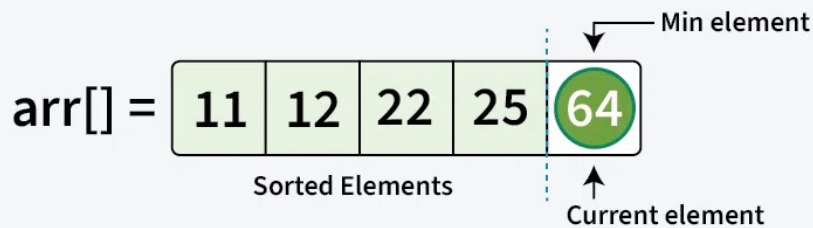
Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).



Selection Sort Algorithm

05
Step

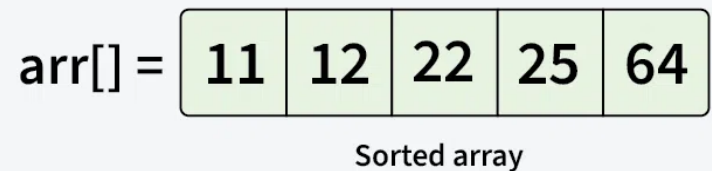
Move to element at index 4 (64), find the minimum from unsorted subarray and swap (64) with current element (64).



Selection Sort Algorithm

06
Step

We get the sorted array at the end.

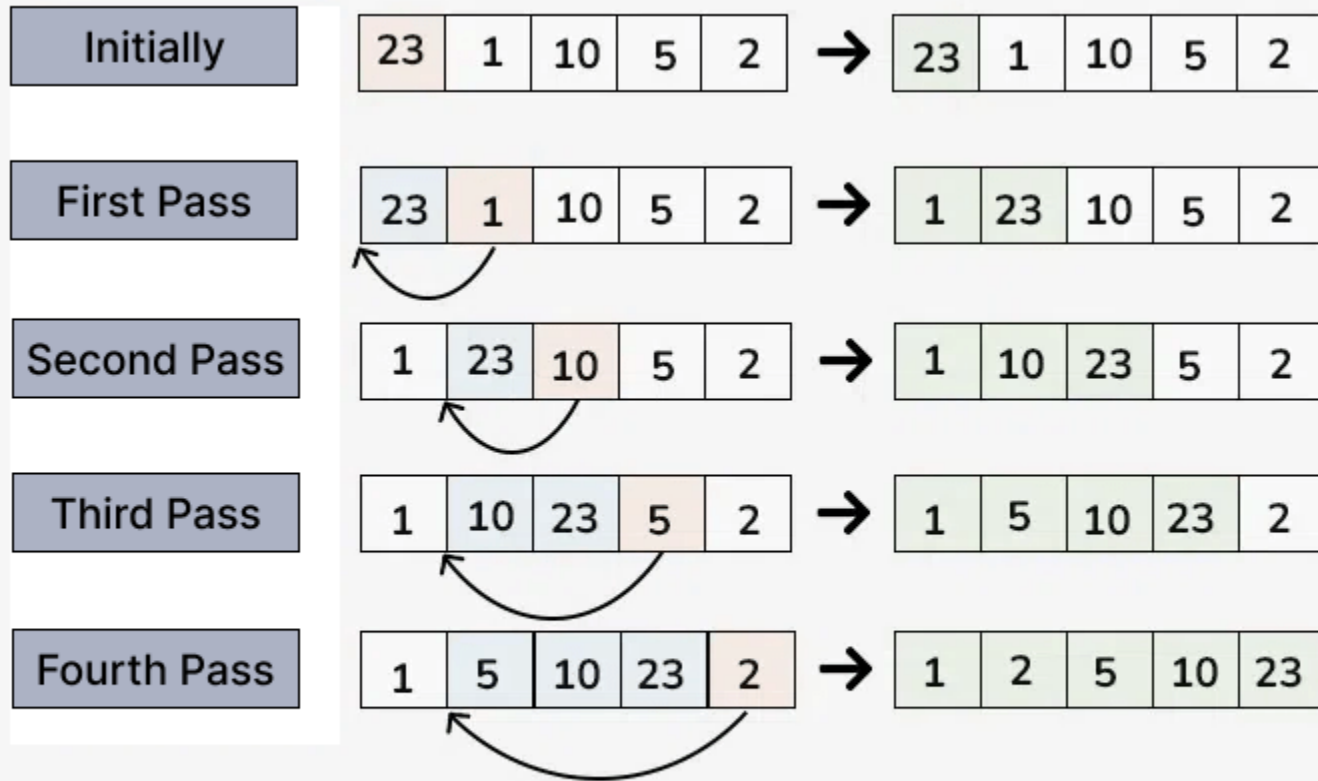


Selection Sort Algorithm

Insertion Sort

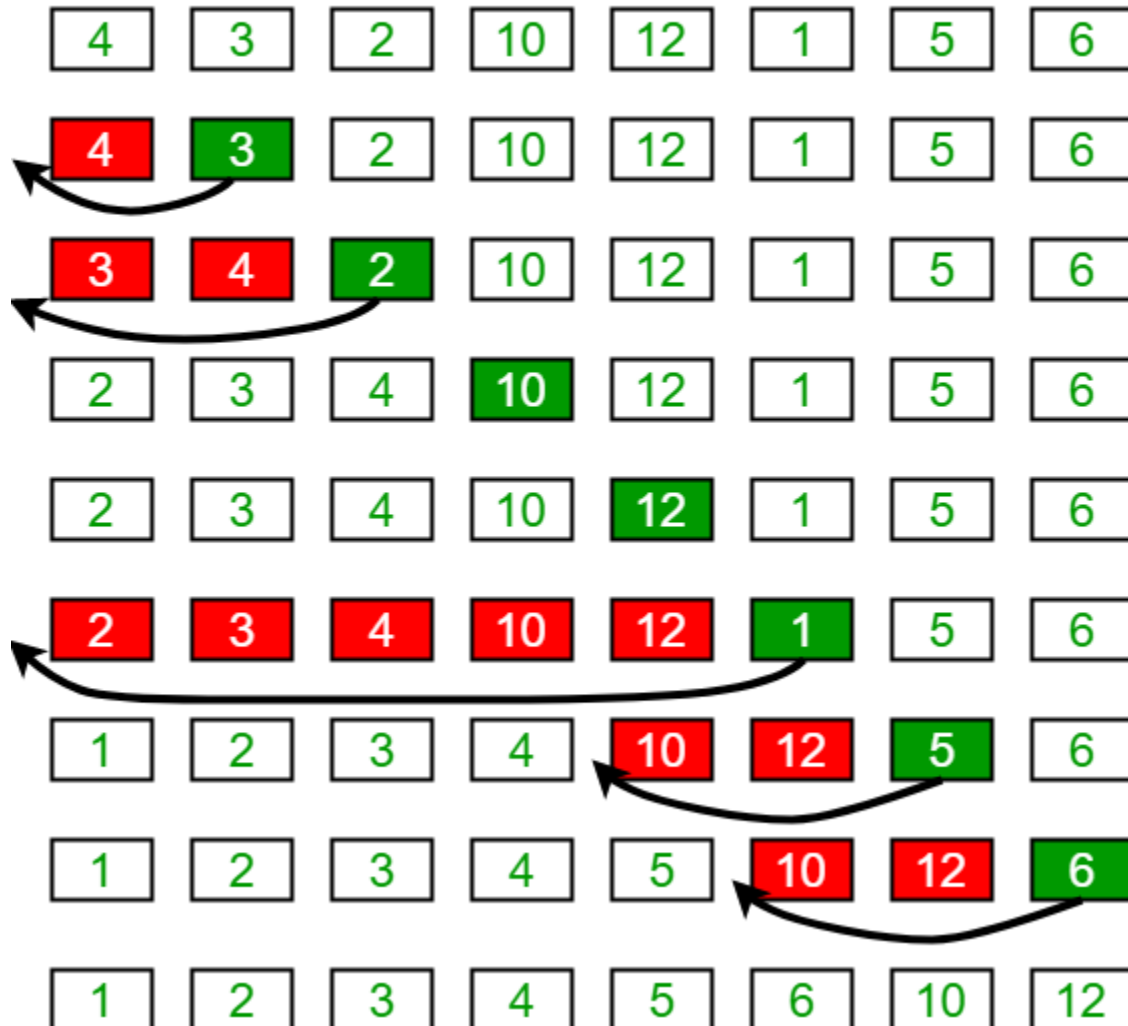
- Insertion sort works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.
 - We start with second element of the array as first element in the array is assumed to be sorted.
 - Compare second element with the first element and check if the second element is smaller then swap them.
 - Move to the third element and compare it with the first two elements and put at its correct position
 - Repeat until the entire array is sorted.
- Time complexity: $O(n^2)$, as there are two nested loops:
 - Outer loop to select each element in the unsorted group one by one, with $O(n)$ complexity
 - Inner loop to insert that element into the sorted group, with $O(n)$ complexity
- Insertion Sort | GeeksforGeeks
 - <https://www.geeksforgeeks.org/insertion-sort-algorithm/>
 - <https://www.geeksforgeeks.org/time-and-space-complexity-of-insertion-sort-algorithm/>
 - <https://www.youtube.com/watch?v=OGzPmgsI-pQ>

Insertion Sort Example 1



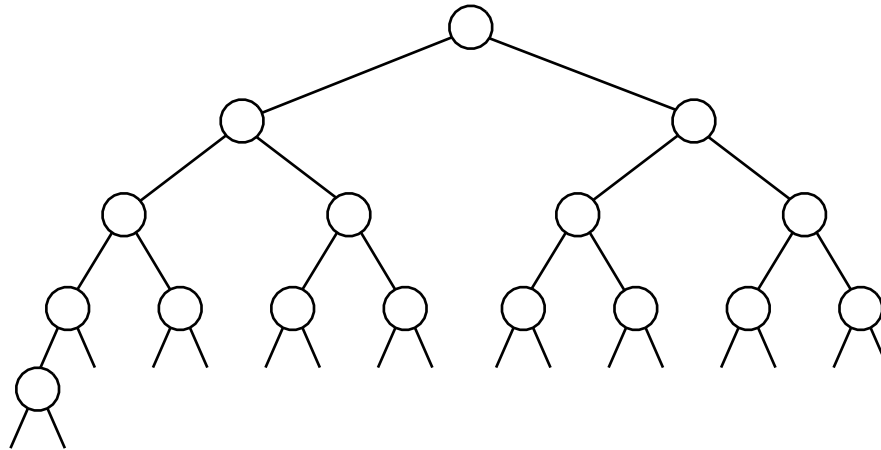
Insertion Sort Example 2

Insertion Sort Execution Example



Heapsort: Binary Heap

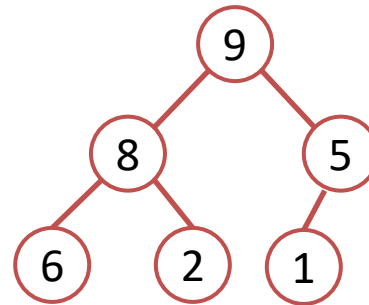
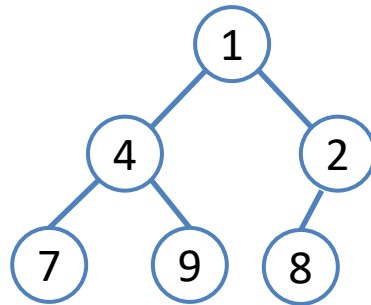
- In a **heap** the highest (or lowest) priority element is always stored at the root, hence the name "heap". A heap is useful data structure when you need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a **priority queue** and **heapsort**.
- A **binary 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.



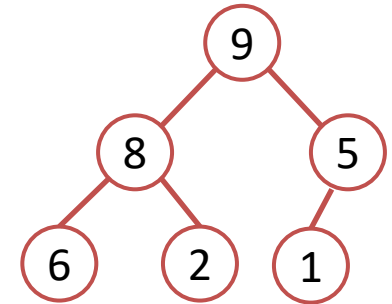
in this lecture
↙

- 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 has $O(\log n)$ height.

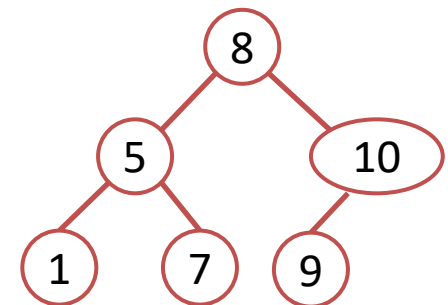
Min Heap Animations | Data Structure | Visual How
<https://www.youtube.com/watch?v=AFPzC2RJOMk>
Max Heap Animations | Data Structure | Visual How
<https://www.youtube.com/watch?v=uU0iWaVxMgc>

Binary Heap vs. Binary Search Tree

- Binary Heap is different from Binary Search Tree (e.g., red-black tree)
- Binary Heap: the max-heap property
 - Value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.
- Binary Search Tree: Ordered, or sorted, binary trees
 - Items to the left of a given node are smaller.
 - Items to the right of a given node are larger.
- Both structures offer $O(\log n)$ time complexity for certain operations, they are used in different scenarios.
 - Heapsort is used for efficient sorting and simple priority queue implementations
 - Red-black trees are for maintaining ordered data with frequent updates and searches
 - Red-black trees can also be used for sorting, by insertions followed by in-order traversal, with $O(n \log(n))$ complexity, but is less efficient in terms of memory and execution time than efficient sorting algorithms.



Binary Heap

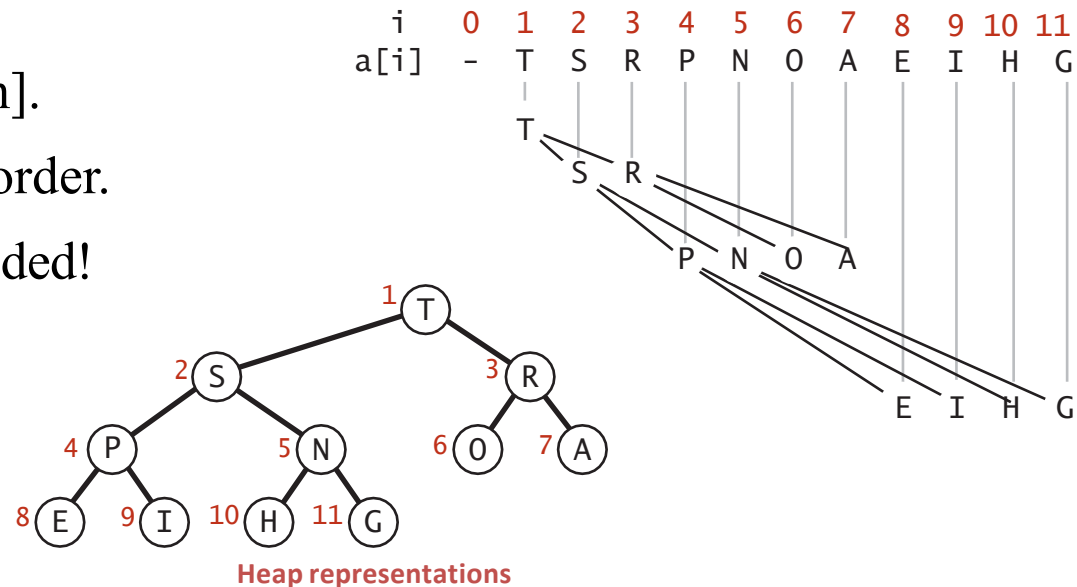


Binary Search Tree

Binary Heap: Array Representation

Array representation.

- Indices range in $[1, n]$.
- Take nodes in **level** order.
- No explicit links needed!



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

- Leaf nodes have indices $[\text{floor}(n/2)+1, n]$
 - $[\text{floor}(11/2)+1, n] = [6, 11]$
- Non-leaf nodes have indices $[1, \text{floor}(n/2)]$
 - $[1, \text{floor}(11/2)] = [1, 5]$

Proposition. Can use array indices to move through tree.

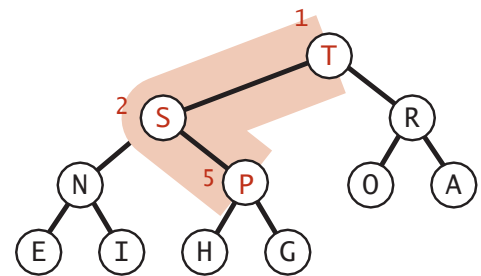
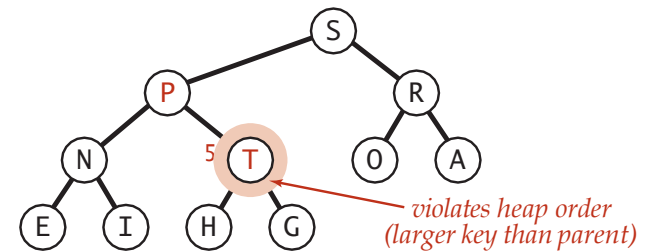
- Parent of node at k is at $\text{floor}(k/2)$.
 - Parent of node at 8 or 9 is at $\text{floor}(8/2) = \text{floor}(9/2) = 4$.
- Children of node at k are at $2k$ and $2k+1$.
 - Children of node at 4 is at $2*4=8$ and $2*4+1=9$

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;
    }
}
```

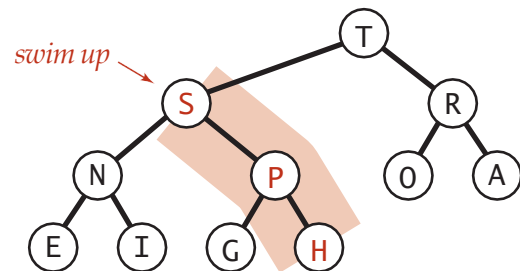
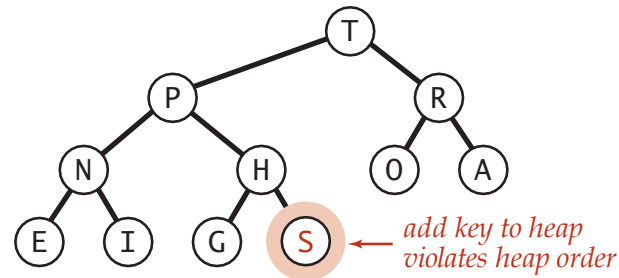
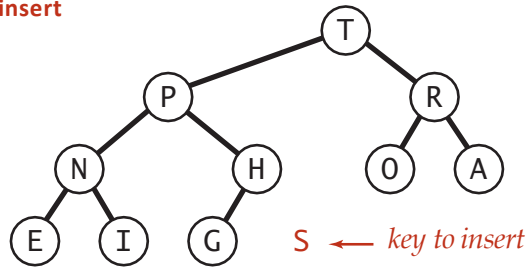
parent of node at k is at k/2



Binary Heap Operations: Insert

- **Insert.** Add node as leaf, then **swim** it up.
- **Cost.** $O(\log n)$ compares since tree height is $O(\log n)$.

insert



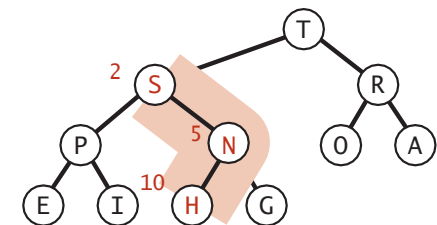
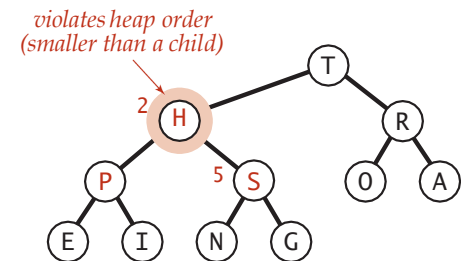
```
public void insert(Key x)
{
    pq[++n] = x;
    swim(n);
}
```


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.
- (Called `max_heapify` in video)

```
private void sink(int k, int n)
{
    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;
    }
}
```

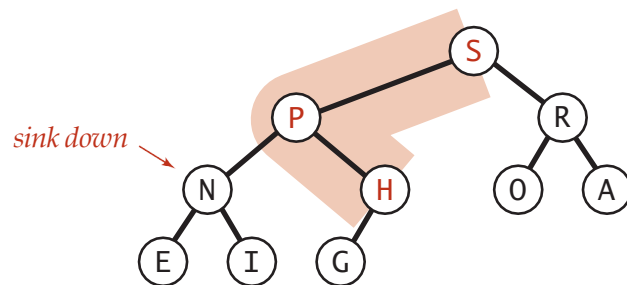
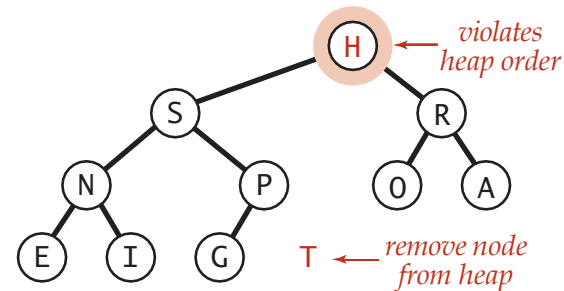
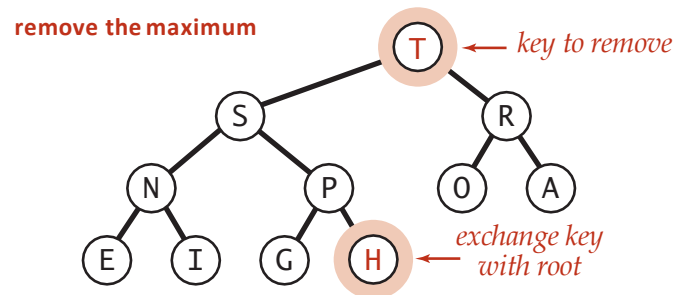
children of node at k
are $2*k$ and $2*k+1$



Top-down `max_heapify`(sink)

Binary Heap Operations: DeleteMax

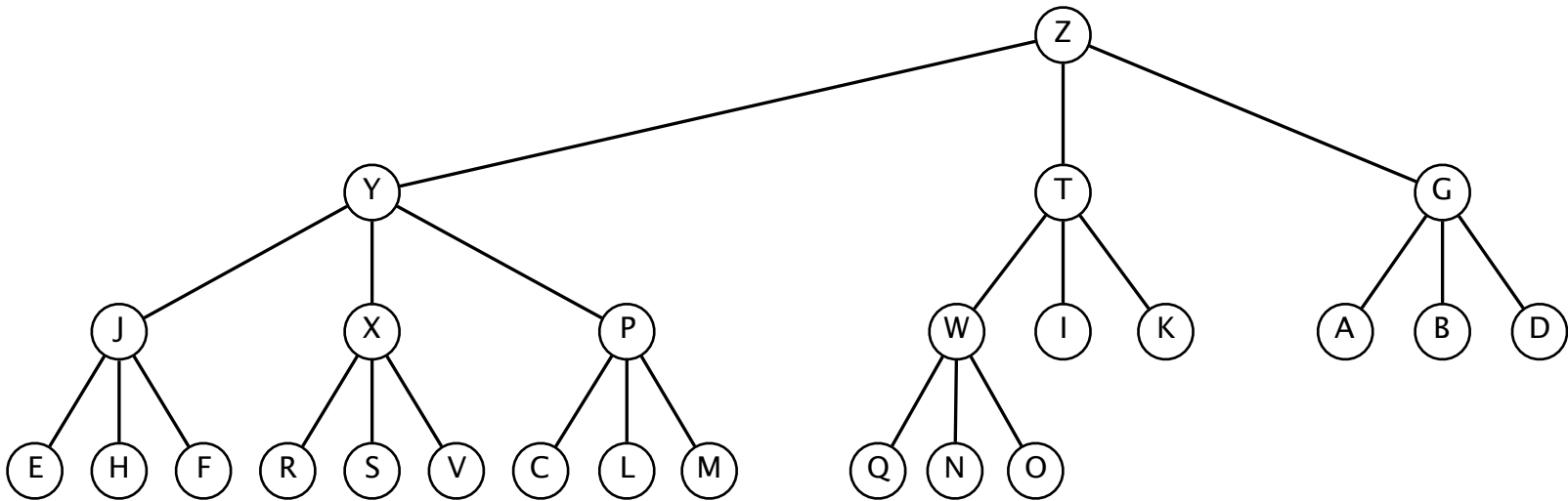
- **Delete max.** Exchange root with node at end, then **sink** it down.
- **Cost.** $O(\log n)$ compares.



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

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 $\sim \log_d n$.

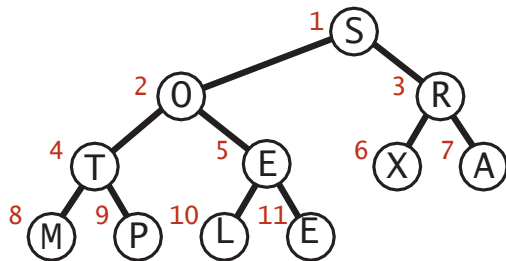


3-way heap

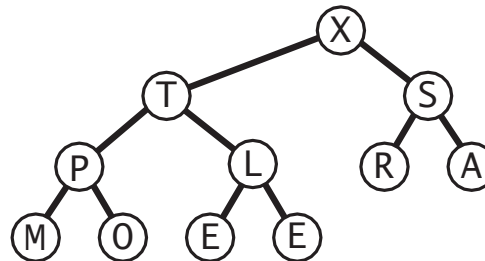
Heapsort Algorithm

- 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.

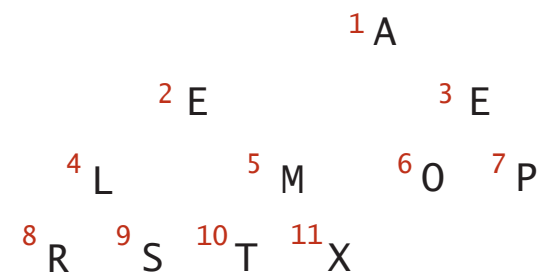
keys in arbitrary order



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

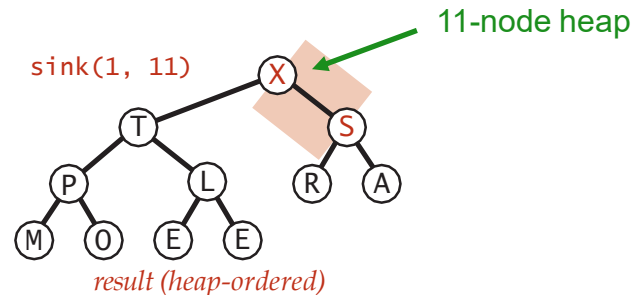
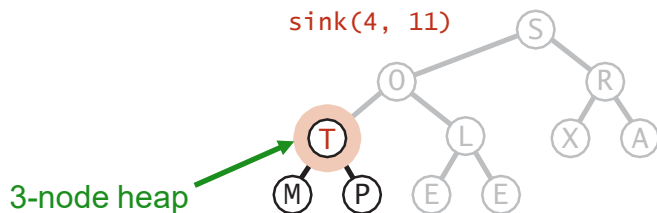
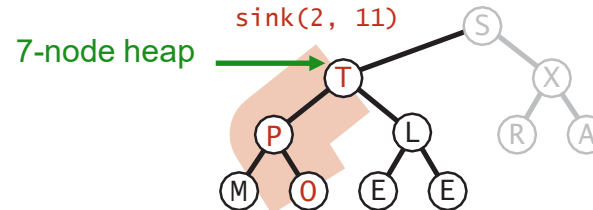
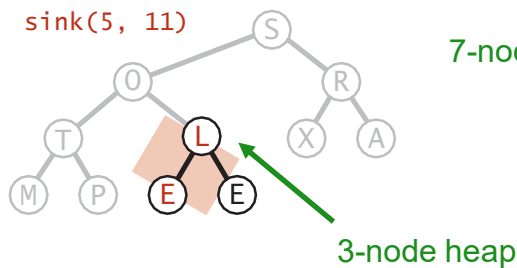
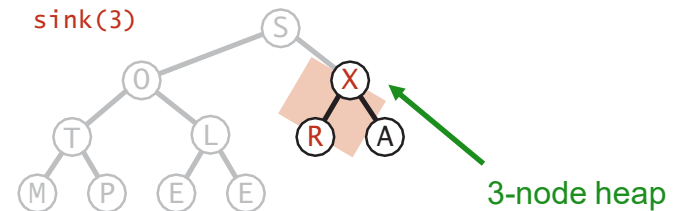
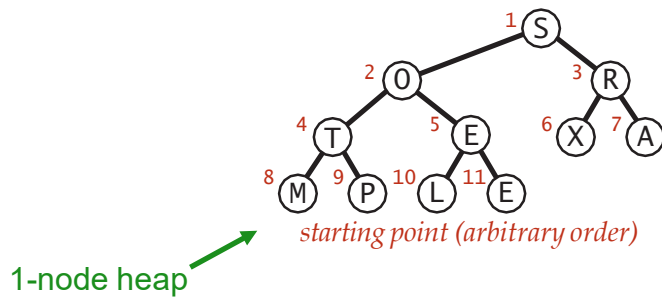
1	2	3	4	5	6	7	8	9	10	11
X	T	S	P	L	R	A	M	O	E	E

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

Heapsort: Heap Construction

First pass. Build heap using bottom-up method.

```
for (int k = floor(n/2); k >= 1; k--)  
    //call sink(k) on all non-leaf  
    nodes k from bottom up  
    sink(k, n);
```

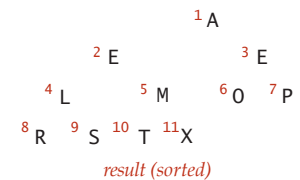
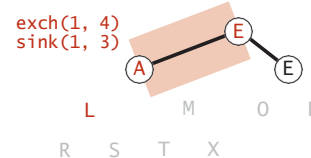
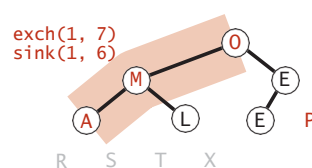
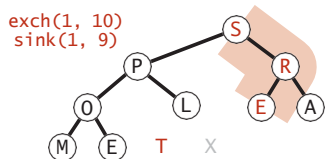
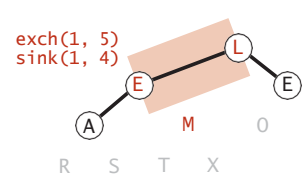
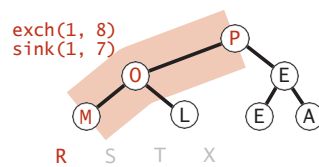
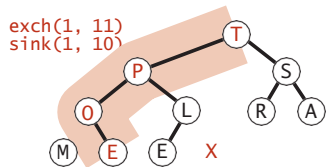
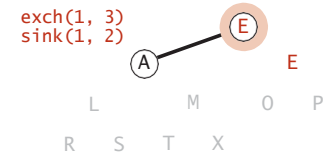
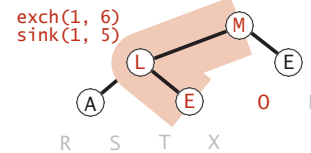
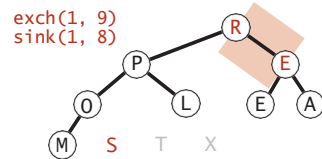
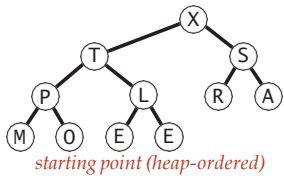


Heapsort: Sortdown

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.
 - We use the first element as Pivot in the examples.
 - #### 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 Time Complexity

Worst case. Number of compares is quadratic.

- $n + (n - 1) + (n - 2) + \dots + 1 \rightarrow O(n^2)$.
- More likely that your computer is struck by lightning bolt.

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

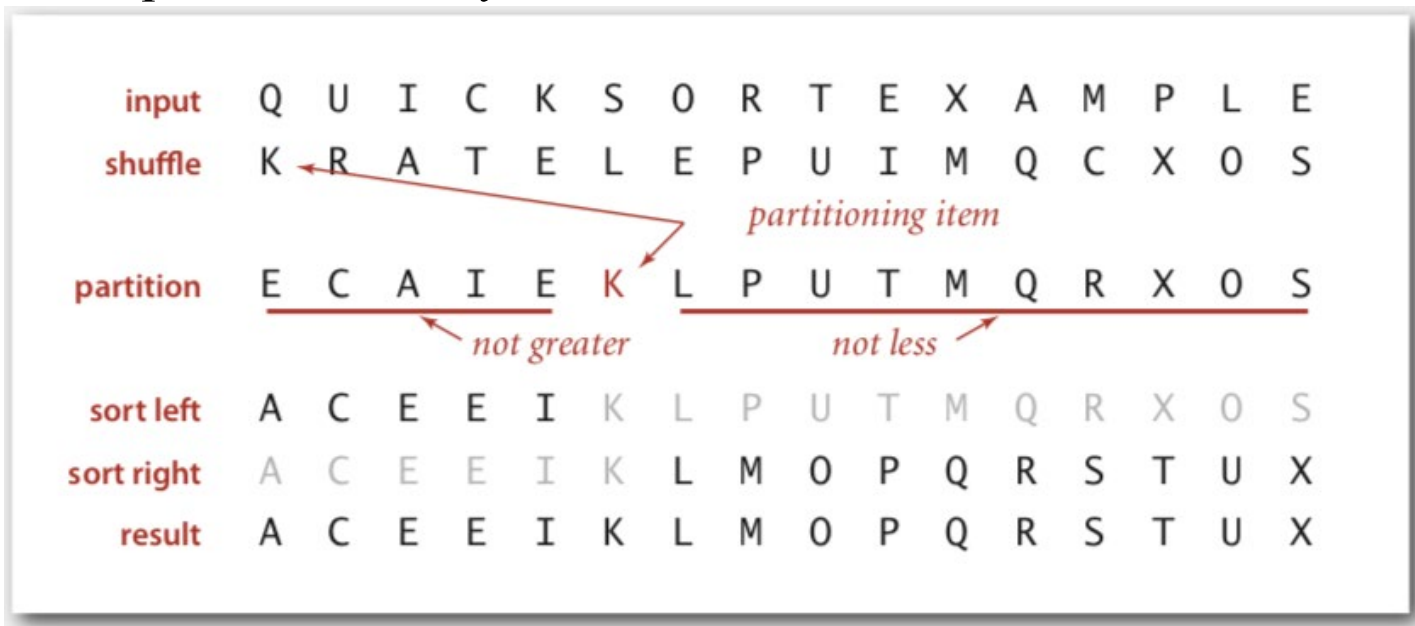
- 39% more compares than mergesort.
- **But** faster than mergesort in practice because of less data movement.

Random shuffle.

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

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.



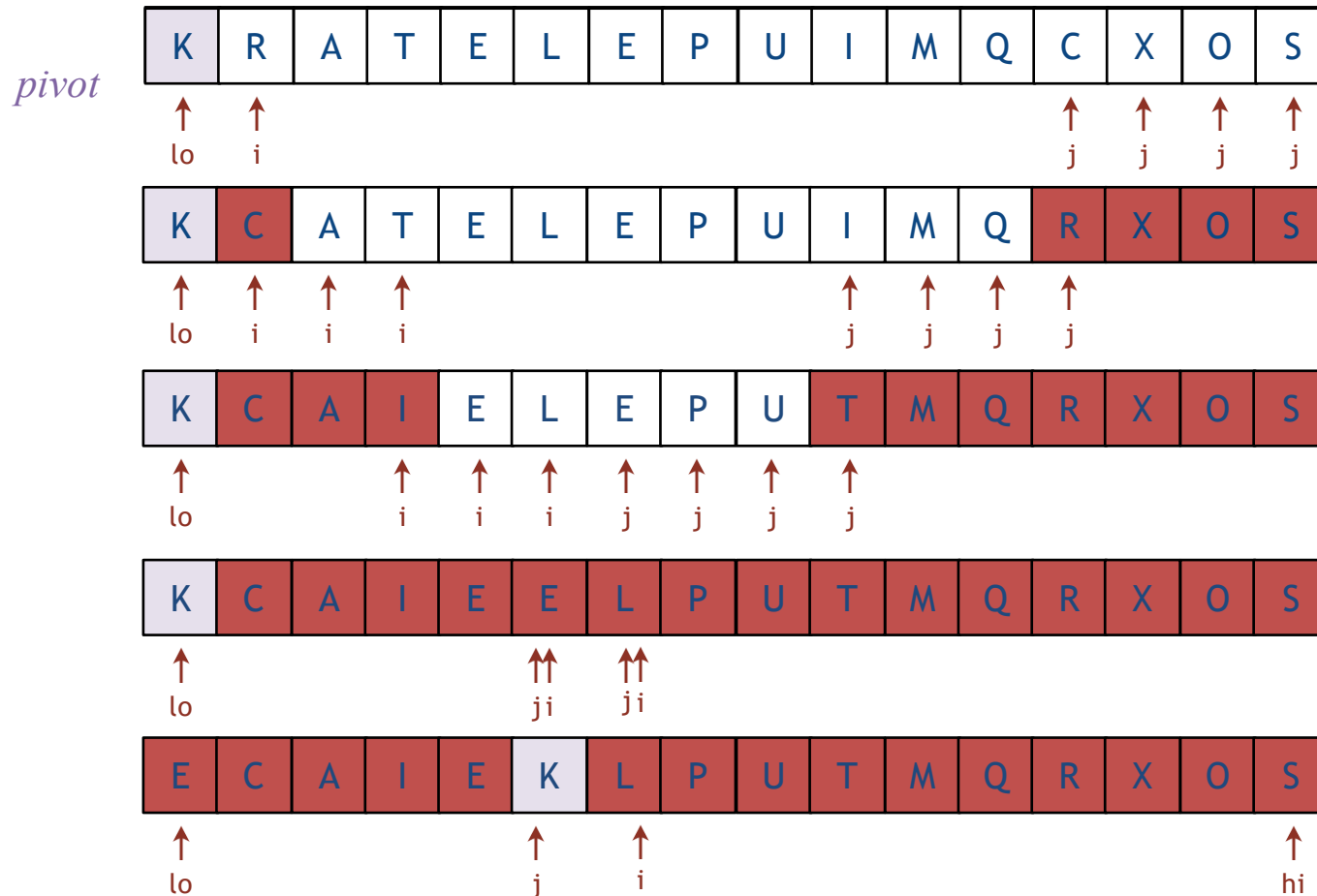
Partition Operation

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 ≈ 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);
}
```

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=ho05eggqPl4>

Merge Sort Animations | Data Structure | Visual How

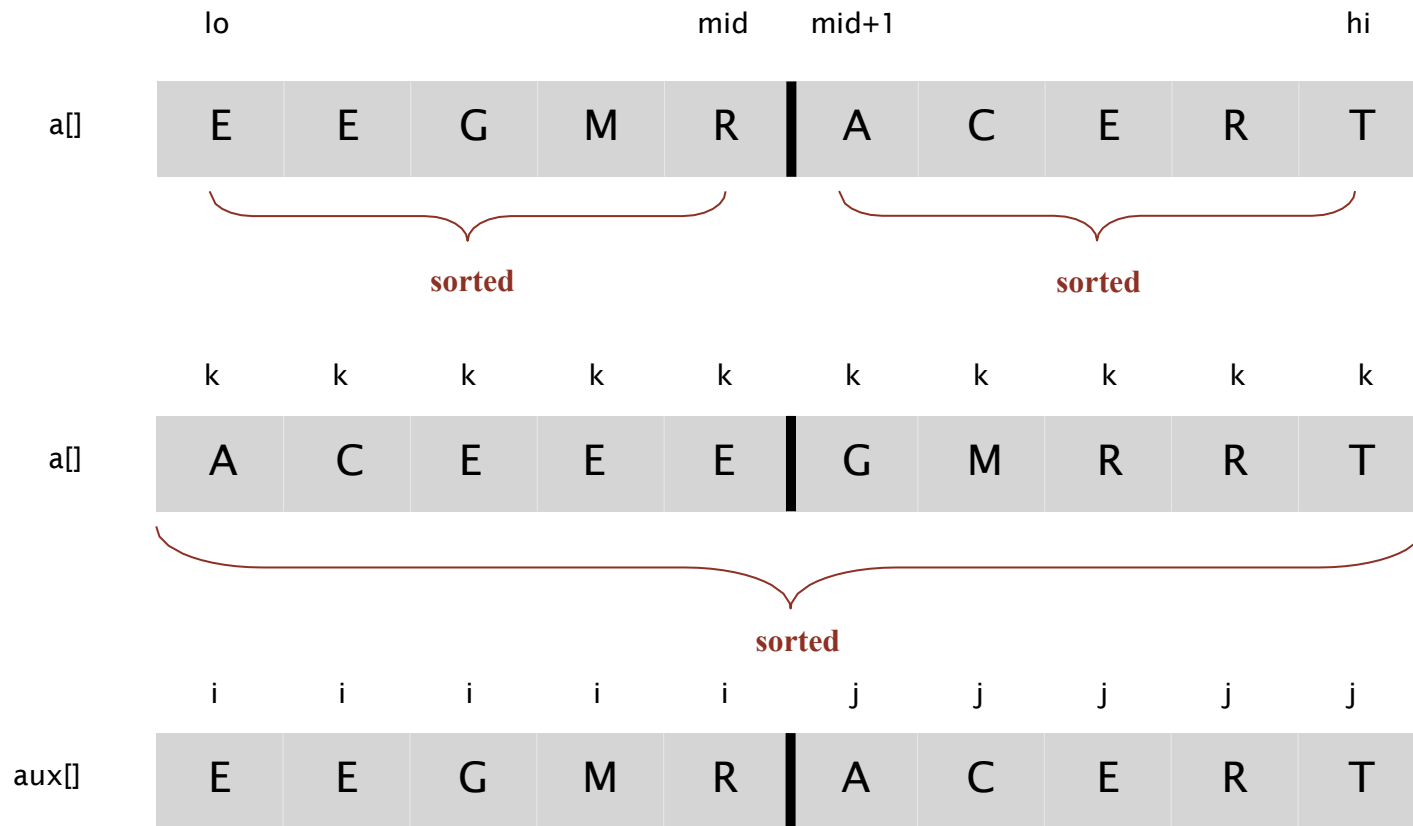
https://www.youtube.com/watch?v=spVhtO_lcGg

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

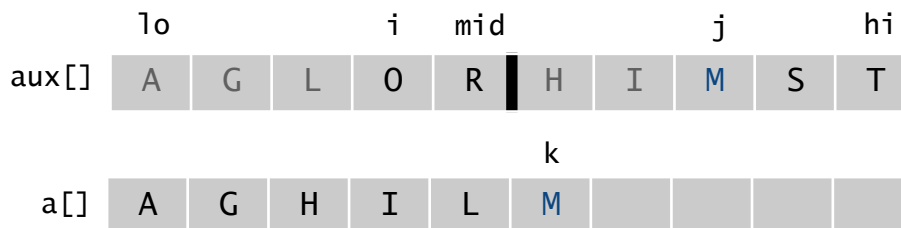
Merge Operation: Java Implementation

```
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];              copy

    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
}
```

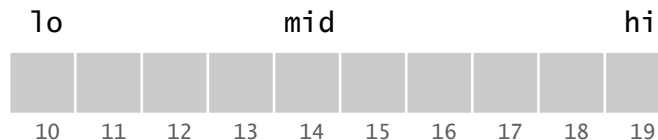


Mergesort: Java implementation

```
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, ^{lo} 0, 0, ^{hi} 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 ≈ 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
sz = 1	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
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

Stable Sorting Algorithm

- A stable sorting algorithm is one that maintains the relative order of elements with equal keys in the sorted output as they appeared in the input.
- Stability is important when multiple sorting operations are performed on data with multiple keys. For example, if you first sort a list of students by name and then by grade, a stable sort will ensure that students with the same grade remain sorted by name. This characteristic is crucial in scenarios where secondary attributes need to be preserved after sorting by primary attributes.
- Stable Sorting Algorithms include: Bubble Sort, Insertion Sort, Merge Sort, Radix Sort (next lecture)

Summary

	inplace?	stable?	Complexity (worst-case)	remarks
Bubble	✓	✓	$O(n^2)$	
Selection	✓		$O(n^2)$	
Insertion	✓	✓	$O(n^2)$	
Heap	✓		$O(n \lg n)$	$O(n \log n)$ guarantee
Quick	✓		$O(n^2)$	$O(n \log n)$ probabilistic guarantee; fastest in practice
Merge		✓	$O(n \lg n)$	$O(n \log n)$ guarantee;

Short Videos of Sorting Algorithms

- Sort Algos // Michael Sambol Michael Sambol
 - https://www.youtube.com/playlist?list=PL9xmBV_5YoZOSbGAXAPIq1BeUf4j20pl
 - Merge Sort, Quick Sort, Bubble Sort, Insertion Sort, Selection Sort, Heap Sort
- Insertion Sort in 2 min (Python)
 - <https://www.youtube.com/watch?v=ZS2ruPQm9QQ>
- Selection Sort in 2 min (Python)
 - <https://www.youtube.com/watch?v=O6qX5xHXWpI>
- Bubble Sort in 2 min (Python)
 - <https://www.youtube.com/watch?v=2DNItBUngrA>
- Visual How
 - <https://www.youtube.com/@visualhow/videos>
- 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

Video Tutorials

- Heap Sort
 - Heaps // Michael Sambol
 - https://www.youtube.com/playlist?list=PL9xmBV_5YoZNsyqgPW-DNwUeT8F8uhWc6
 - Binary Min Heap
 - <https://www.youtube.com/playlist?list=PLvTjg4siRgU197GA1yFNRWUgsPZnvjuyL>
 - HEAP SORT | Sorting Algorithms | DSA | GeeksforGeeks
 - https://www.youtube.com/watch?v=MtQL_lI5KhQ
- Quick Sort
 - Quick sort in 4 minutes
 - <https://www.youtube.com/watch?v=Hoixgm4-P4M&t=19s>
 - Quicksort Algorithm: A Step-by-Step Visualization
 - <https://www.youtube.com/watch?v=bZkzH5x0SKU>
 - Visualization of Quick sort (HD)
 - <https://www.youtube.com/watch?v=aXXWXz5rF64>
- Merge Sort
 - Merge sort in 3 minutes
 - <https://www.youtube.com/watch?v=4VqmGXwpLqc>