# CMSC 132: Object-Oriented Programming II

## Sorting

Department of Computer Science
University of Maryland, College Park

# Overview

- Comparison sort
  - Bubble sort
  - Selection sort
  - Tree sort
  - Heap sort
  - Quick sort
  - Merge sort
- Linear sort
  - Counting sort
  - Bucket (bin) sort
  - Radix sort

$O(n^2)$

$O(n \log(n))$

$O(n)$

# Sorting

- **Goal**
  - **Arrange elements in predetermined order**
    - **Based on key for each element**
  - **Derived from ability to compare two keys by size**
- **Properties**
  - **Stable ⇒ relative order of equal keys unchanged**
    - **Stable: 3, 1, 4, 3, 3, 2 → 1, 2, 3, 3, 3, 4**
    - **Unstable: 3, 1, 4, 3, 3, 2 → 1, 2, 3, 3, 3, 4**
  - **In-place ⇒ uses only constant additional space**
  - **External ⇒ can efficiently sort large # of keys**

# Sorting

- Comparison sort
  - Only uses pairwise key comparisons
  - Proven lower bound of O( n log(n) )

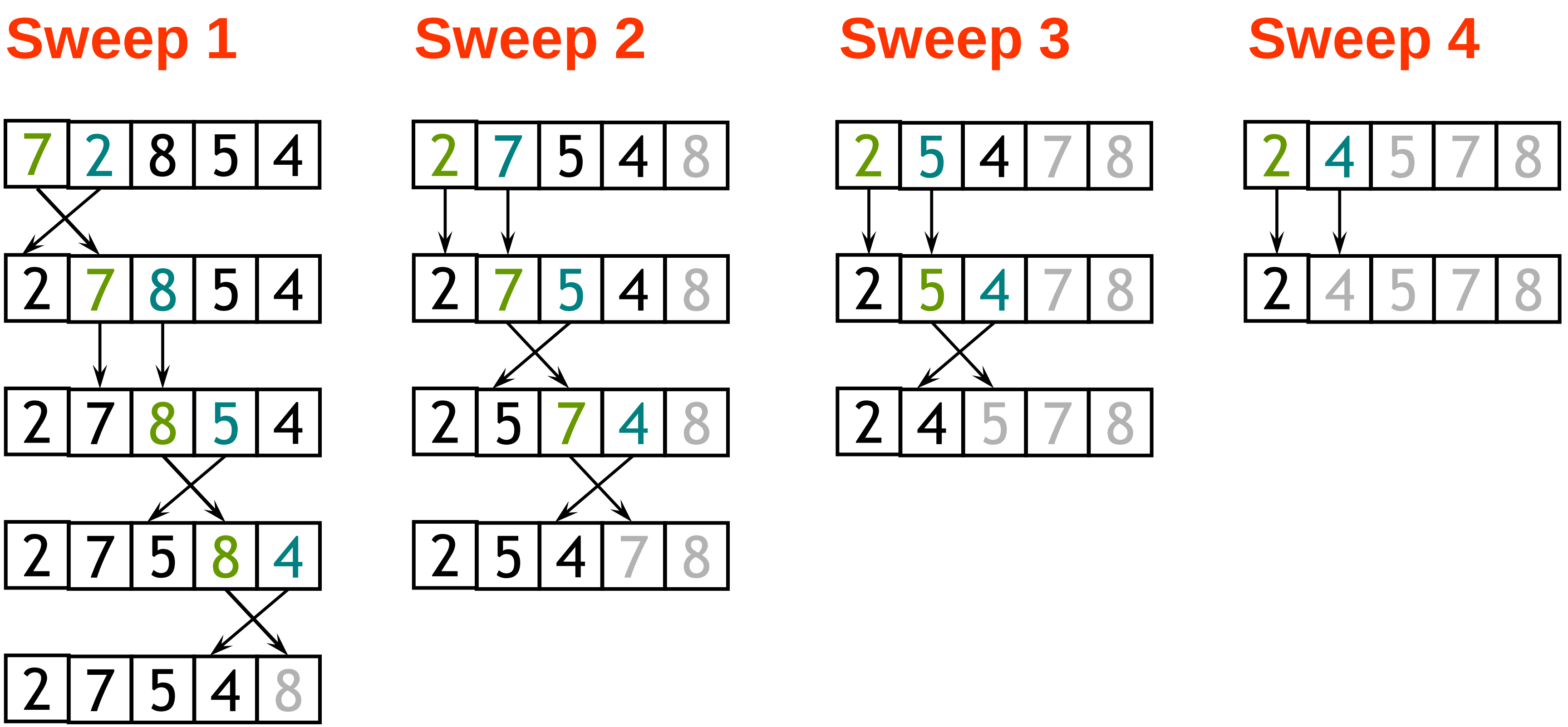
- Linear sort
  - Uses additional properties of keys

# Bubble Sort

- **Approach**
  1. **Iteratively sweep through shrinking portions of list**
  2. **Swap element x with its right neighbor if x is larger**

- **Performance**
  - **$O( n^2 )$ average / worst case**

# Bubble Sort Example



# Bubble Sort Code

```
void bubbleSort(int[ ] a) {
    int outer, inner;
    for (outer = a.length - 1; outer > 0; outer--)
        for (inner = 0; inner < outer; inner++)
          if (a[inner] > a[inner + 1])
              swap(a, inner, inner+1);
}

void swap(int a[ ], int x, int y) {
    int temp = a[x];
    a[x] = a[y];
    a[y] = temp;
}
```

Sweep through array

Swap with right neighbor if larger

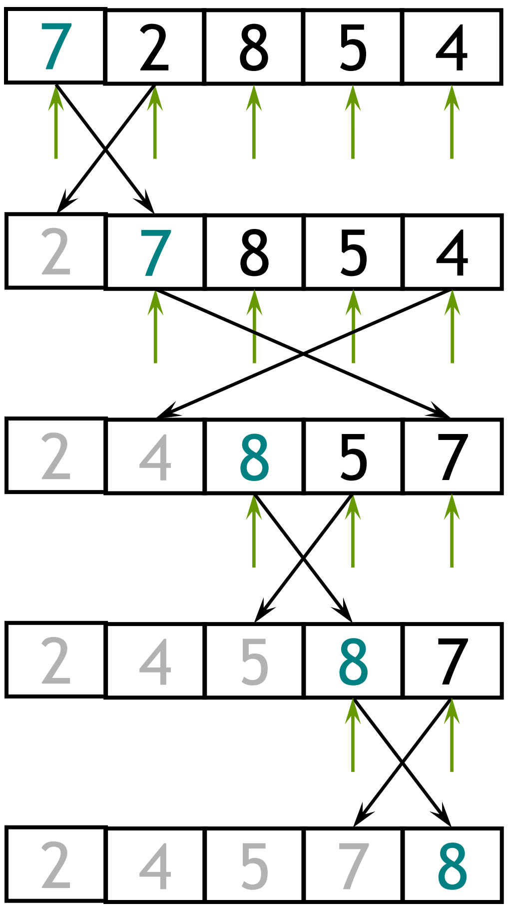
Swap array elements at positions x & y

# Selection Sort

- **Approach**
  1. Iteratively sweep through shrinking portions of list
  2. Select smallest element found in each sweep
  3. Swap smallest element with front of current list

- **Performance**
  - O( $n^2$ ) average / worst case

- **Example**



# Selection Sort Code

```
void selectionSort(int[] a) {
        int outer, inner, min;
        for (outer = 0; outer < a.length - 1; outer++) {
            min = outer;
            for (inner = outer + 1; inner < a.length; inner++) {
                if (a[inner] < a[min]) {
                    min = inner;
                }
            }
            swap(a, outer, min);
        }
}
```

Sweep through array

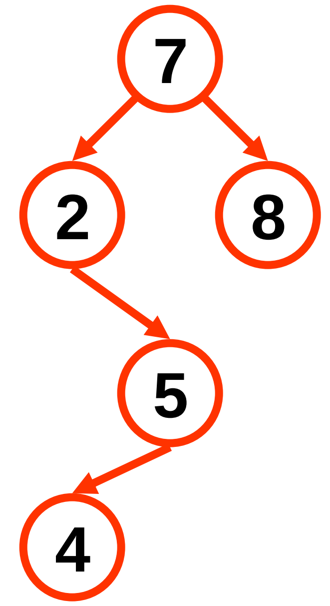
Find smallest element

Swap with smallest element found

# Tree Sort

- **Approach**
  1. Insert elements in binary search tree
  2. List elements using inorder traversal
- **Performance**
  - Binary search tree
    - O( n log(n) ) average case
    - O( $n^2$ ) worst case
  - Balanced binary search tree
    - O( n log(n) ) average / worst case
- **Example**

  Binary search tree




{ 7, 2, 8, 5, 4 }

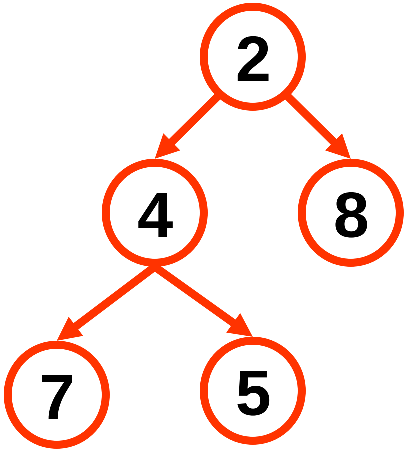
# Heap Sort

- **Approach**
  1. Insert elements in heap
  2. Remove smallest element in heap, repeat
  3. List elements in order of removal from heap

- **Performance**
  - O( n log(n) ) average / worst case

- **Example**

  Heap




{ 7, 2, 8, 5, 4 }

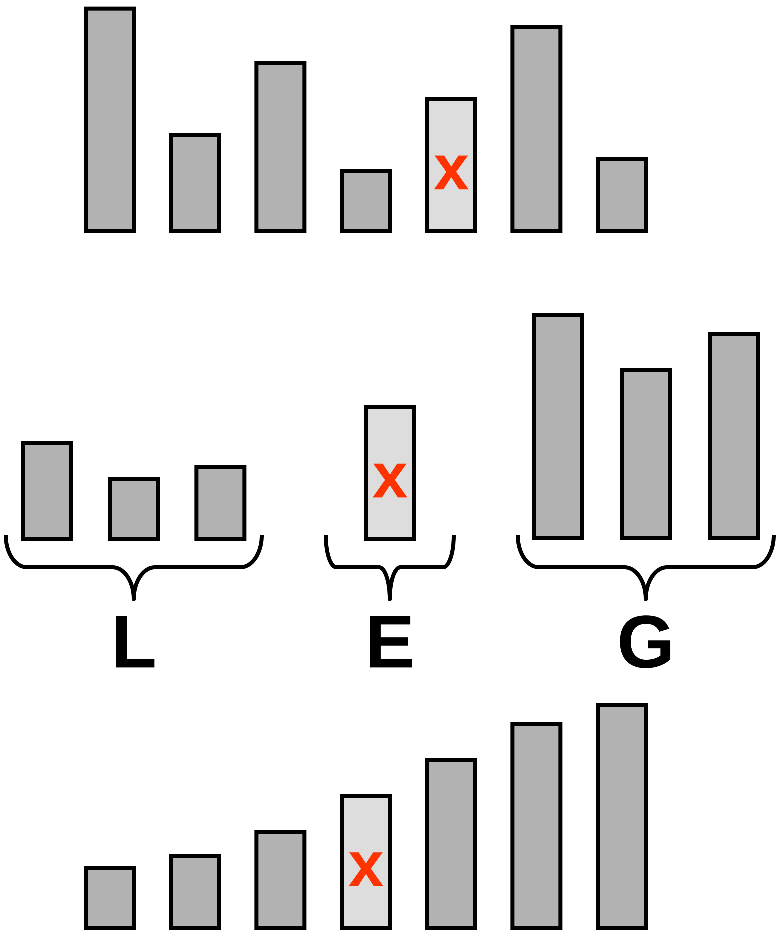
# Quick Sort

- **Approach**
  1. **Select pivot value (near median of list)**
  2. **Partition elements (into 2 lists) using pivot value**
  3. **Recursively sort both resulting lists**
  4. **Concatenate resulting lists**
  - **For efficiency pivot needs to partition list evenly**

- **Performance**
  - **O( n log(n) ) average case**
  - **O( $n^2$ ) worst case**

# Quick Sort Algorithm

1. If list below size K
   - Sort w/ other algorithm
2. Else pick pivot x and partition S into
   - L elements < x
   - E elements = x
   - G elements > x
3. Quicksort L & G
4. Concatenate L, E & G
   - If not sorting in place



# Quick Sort Code

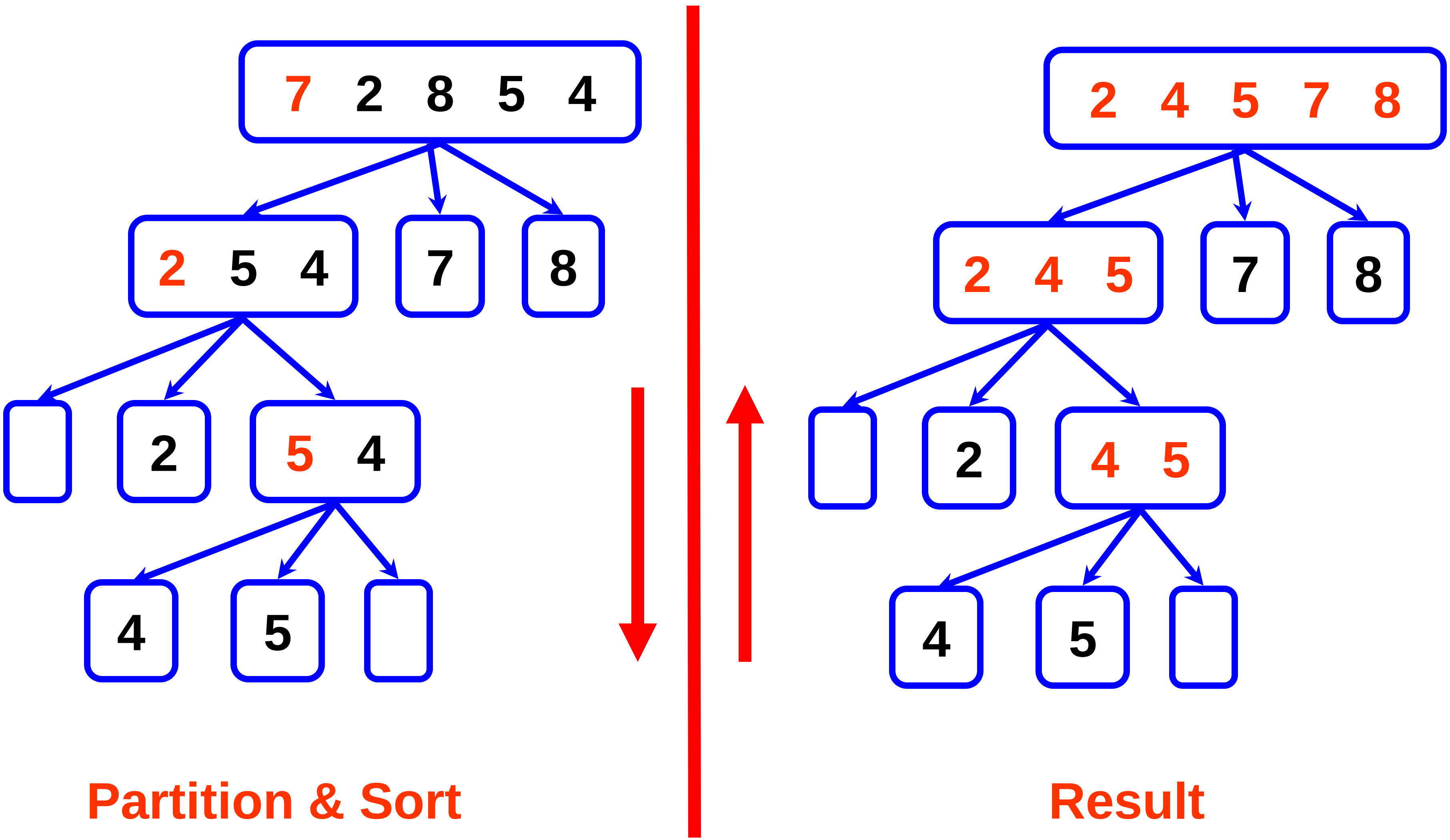
```
void quickSort(int[] a, int x, int y) {
      int pivotIndex;
      if ((y - x) > 0) {
         pivotIndex = partionList(a, x, y);
         quickSort(a, x, pivotIndex - 1);
         quickSort(a, pivotIndex+1, y);
      }
}
```

Lower end of array region to be sorted

Upper end of array region to be sorted

```
int partionList(int[] a, int x, int y) {
       … // partitions list and returns index of pivot
}
```

# Quick Sort Example



# Quick Sort Code

```
int partitionList(int[] a, int x, int y) {
      int pivot = a[x];
      int left = x;
      int right = y;
      while (left < right) {
        while ((a[left] < pivot) && (left < right))
            left++;
        while (a[right] > pivot)
            right--;
         if (left < right)
            swap(a, left, right);
       }
       swap(a, x, right);
       return right;
}
```

Use first element as pivot

Partition elements in array relative to value of pivot

Place pivot in middle of partitioned array

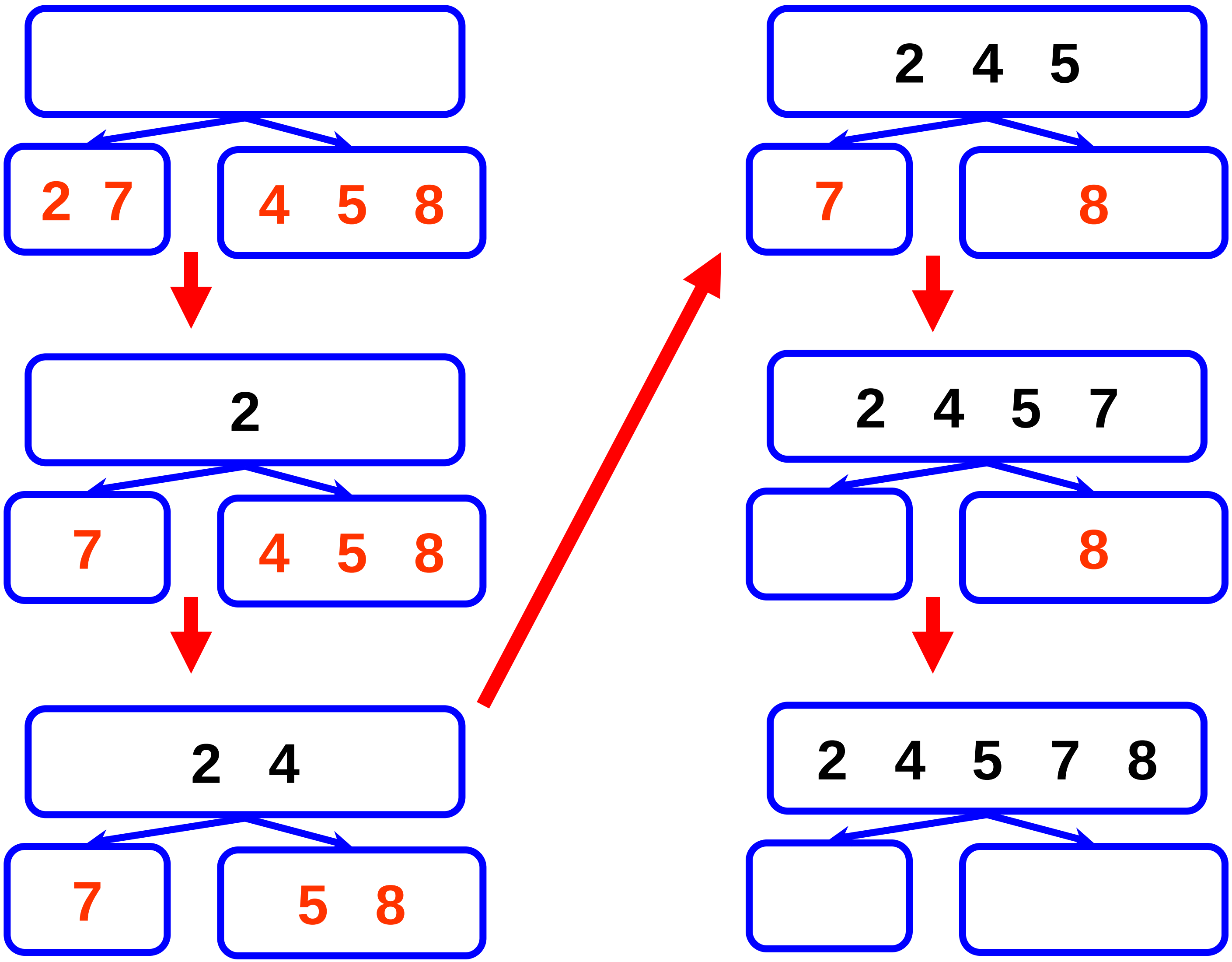
return index of pivot

# Merge Sort

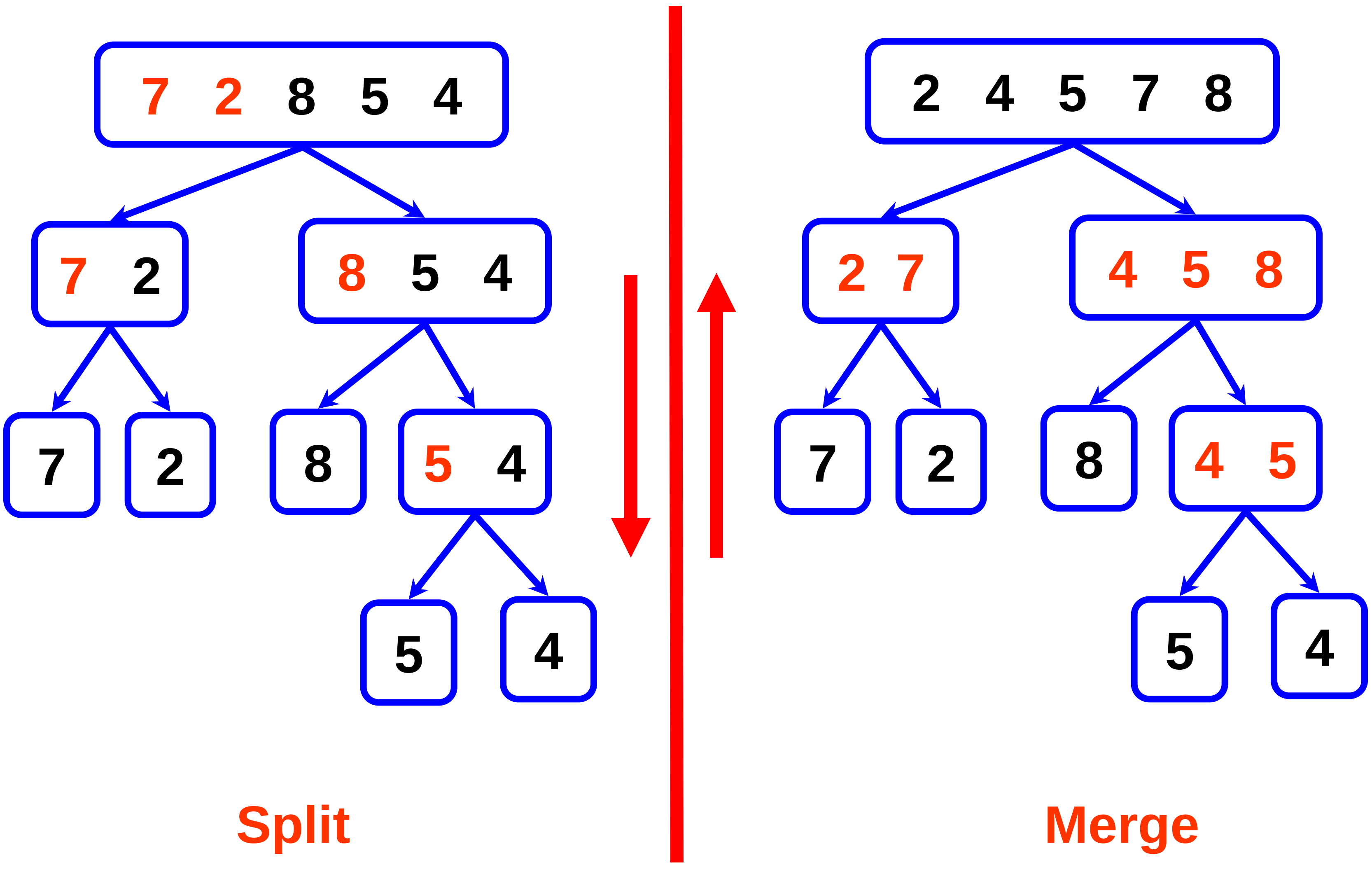
- **Approach**
  1. **Partition list of elements into 2 lists**
  2. **Recursively sort both lists**
  3. **Given 2 sorted lists, merge into 1 sorted list**
     a) **Examine head of both lists**
     b) **Move smaller to end of new list**

- **Performance**
  - **O( n log(n) ) average / worst case**

# Merge Example



# Merge Sort Example



# Merge Sort Code

```
void mergeSort(int[] a, int x, int y) {
      int mid = (x + y) / 2;
      if (y == x) return;
      mergeSort(a, x, mid);
      mergeSort(a, mid+1, y);
      merge(a, x, y, mid);
}
void merge(int[] a, int x, int y, int mid) {
      … // merges 2 adjacent sorted lists in array
}
```

Lower end of array region to be sorted

Upper end of array region to be sorted

# Merge Sort Code

```
void merge (int[] a, int x, int y, int mid) {
    int size = y - x;
    int left = x;
    int right = mid+1;
    int[] tmp; int j;
    for (j = 0; j < size; j++) {
        if (left > mid) tmp[j] = a[right++];
        else if (right > y) || (a[left] < a[right])
            tmp[j] = a[left++];
        else tmp[j] = a[right++];
    }
    for (j = 0; j < size; j++)
        a[x+j] = tmp[j];
}
```

Upper end of 1st array region

Lower end of 1st array region

Upper end of 2nd array region

Copy smaller of two elements at head of 2 array regions to tmp buffer, then move on

Copy merged array back

# Sorting Properties

| Name | Compari-son Sort | Avg Case Complexity | Worst Case Complexity | In Place | Can be Stable |
|---|---|---|---|---|---|
| Bubble | √ | $O(n^2)$ | $O(n^2)$ | √ | √ |
| Selection | √ | $O(n^2)$ | $O(n^2)$ | √ | √ |
| Tree | √ | O(n log(n)) | $O(n^2)$ | | |
| Heap | √ | O(n log(n)) | O(n log(n)) | | |
| Quick | √ | O(n log(n)) | $O(n^2)$ | √ | |
| Merge | √ | O(n log(n)) | O(n log(n)) | | √ |

# Sorting Summary

- Many different sorting algorithms
- Complexity and behavior varies
- Size and characteristics of data affect algorithm