Sorting Algorithms

Nelson Padua-Perez

Bill Pugh

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

Sorting

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

Sorting

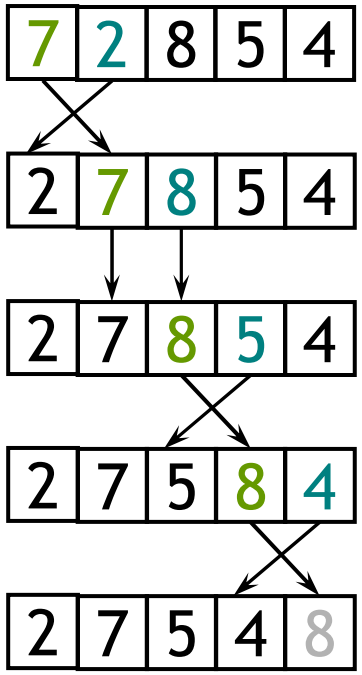
- Comparison sort
  - Only uses pairwise key comparisons
  - Proven lower bound of O(n log(n))
- Linear sort
  - Uses additional properties of keys

Bubble Sort

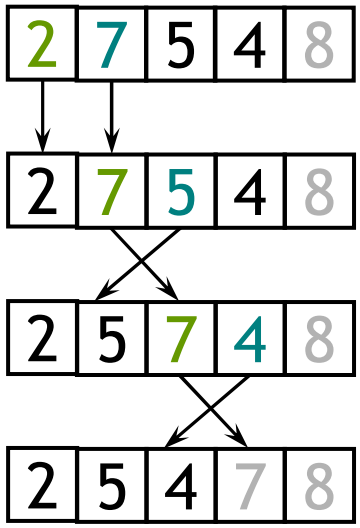
- **Approach**
  - Iteratively sweep through shrinking portions of list
  - Swap element x with its right neighbor if x is larger
- **Performance**
  - O(n^2) average / worst case

Bubble Sort Example

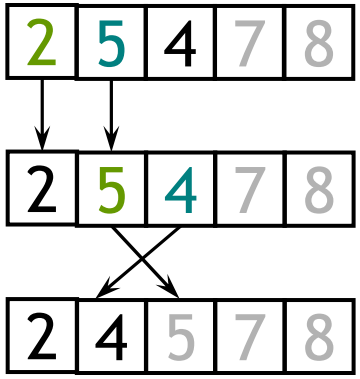
Sweep 1



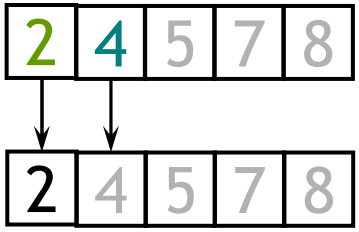
Sweep 2



Sweep 3



Sweep 4



Bubble Sort Code

```
void bubbleSort(int[] a) {

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

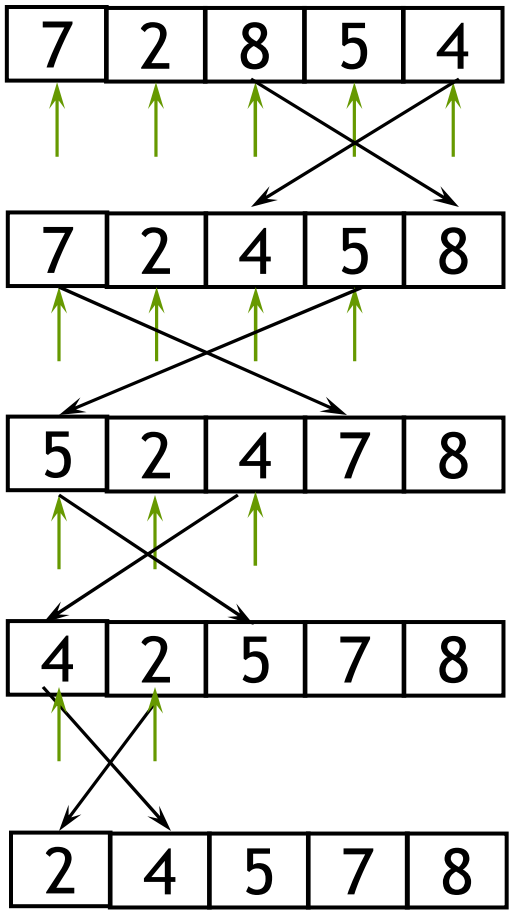
Sweep through array

Swap with right neighbor if larger

Selection Sort

- **Approach**
  - Iteratively sweep through shrinking portions of list
  - Select largest element found in each sweep
  - Swap largest element with end of current list
- **Performance**
  - O(n2) average / worst case

- **Example**



Selection Sort Code

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

Sweep through array

Find largest element

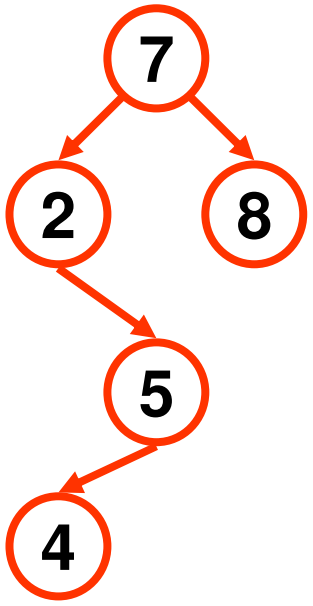
Swap with largest element found

Tree Sort

- Approach
  - Insert elements in binary search tree
  - List elements using inorder traversal
- Performance
  - Binary search tree
    - O(n log(n)) average case
    - O(n2) worst case
  - Balanced binary search tree
    - O(n log(n)) average / worst case

- Example

  Binary search tree



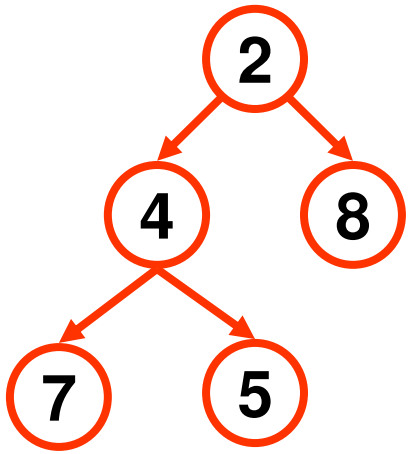
{ 7, 2, 8, 5, 4 }

Heap Sort

- **Approach**
  - Insert elements in heap
  - Remove smallest element in heap, repeat
  - List elements in order of removal from heap
- **Performance**
  - O(n log(n)) average / worst case

- **Example**

  Heap



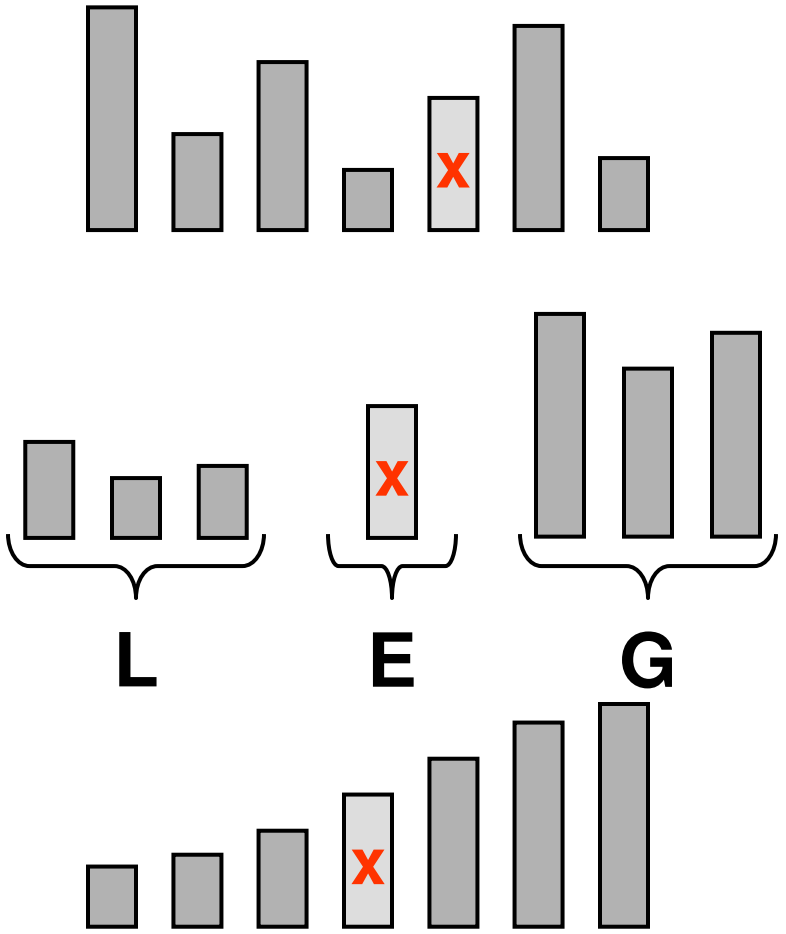
{ 7, 2, 8, 5, 4 }

Quick Sort

- Approach
  - Select pivot value (near median of list)
  - Partition elements (into 2 lists) using pivot value
  - Recursively sort both resulting lists
  - Concatenate resulting lists
  - For efficiency pivot needs to partition list evenly
- Performance
  - O(n log(n)) average case
  - O(n2) worst case

Quick Sort Algorithm

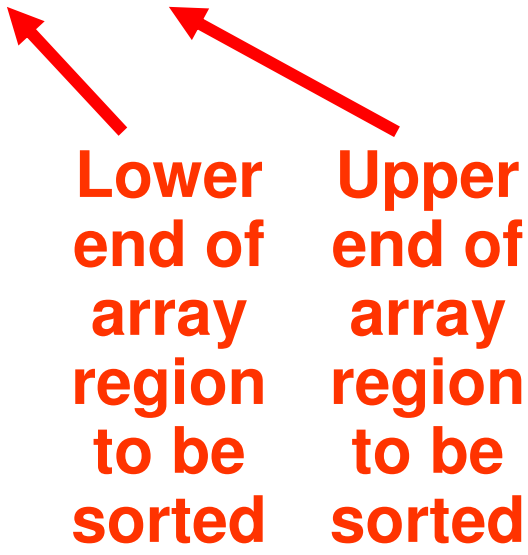
- If list below size K
  - Sort w/ other algorithm
- Else pick pivot x and partition S into
  - L elements <= x
  - E pivot element x
  - G elements > x
- Quicksort L & G
- Concatenate L, E & G
  - If not sorting in place



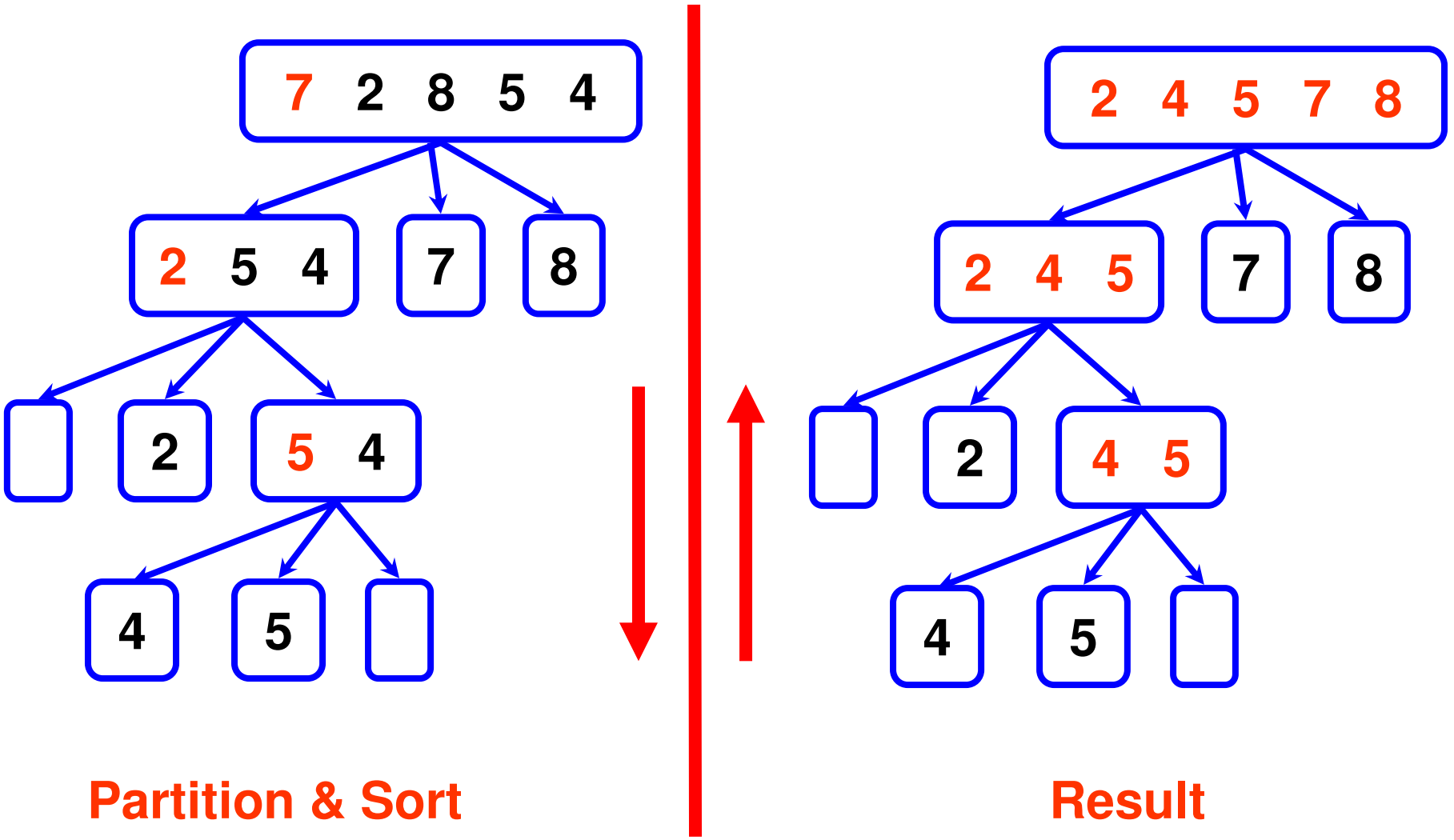
Quick Sort Code

```
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);
      }
}

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



Quick Sort Example



Quick Sort Code

```
static int partitionList(int[] a, int x, int y) {
    int left = x+1;
    int right = y;
    int pivot = a[x];
    while (true) {
     while (a[left] <= pivot && left < right)
       left++;
     while (a[right] > pivot)
       right--;
      if (left >= right) break;
      swap(a, left, right);
    }
    swap(a, x, right);
    return right;
  }
```

Use first element as pivot

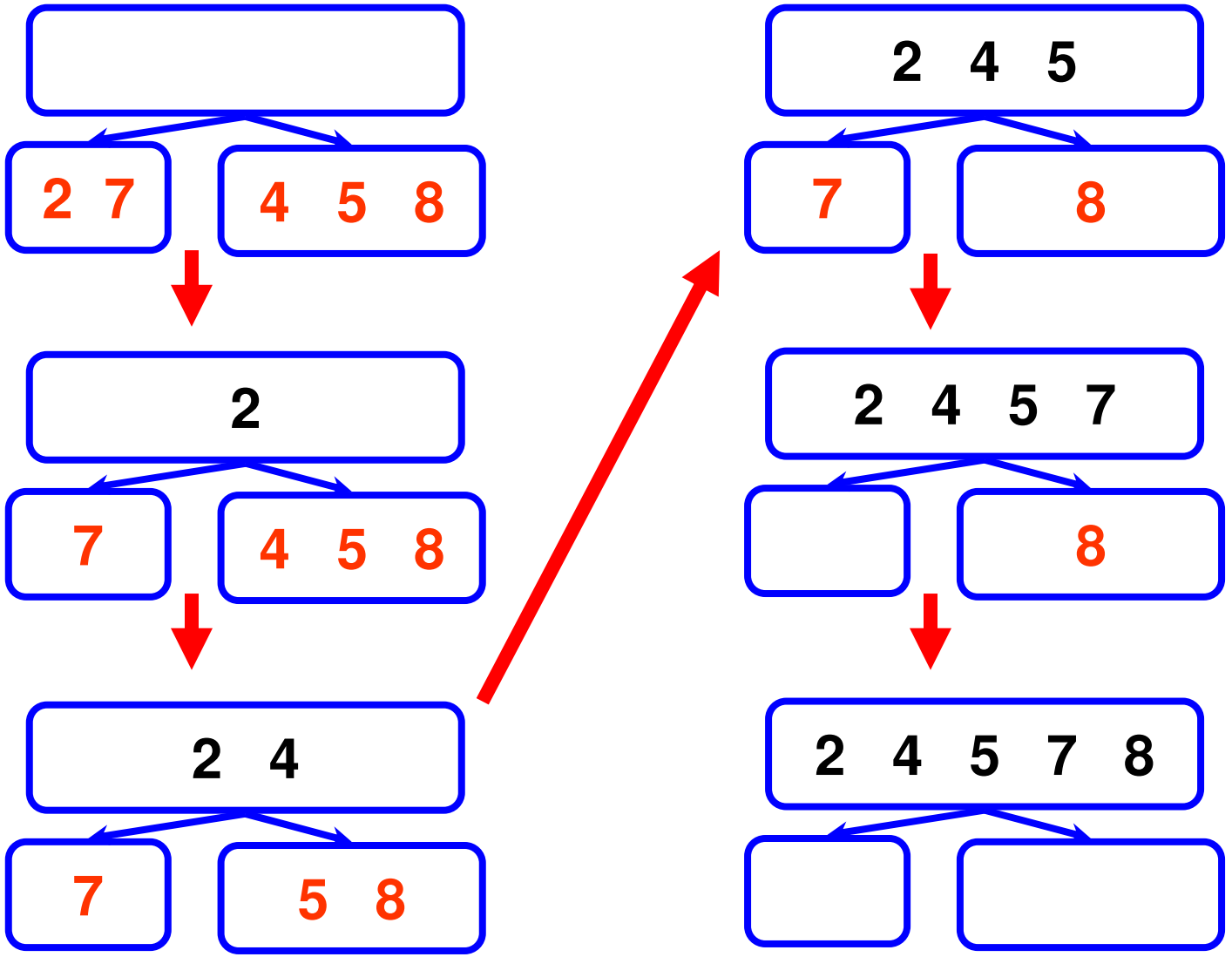
Partition elements in array relative to value of pivot

Place pivot in middle of partitioned array, return index of pivot

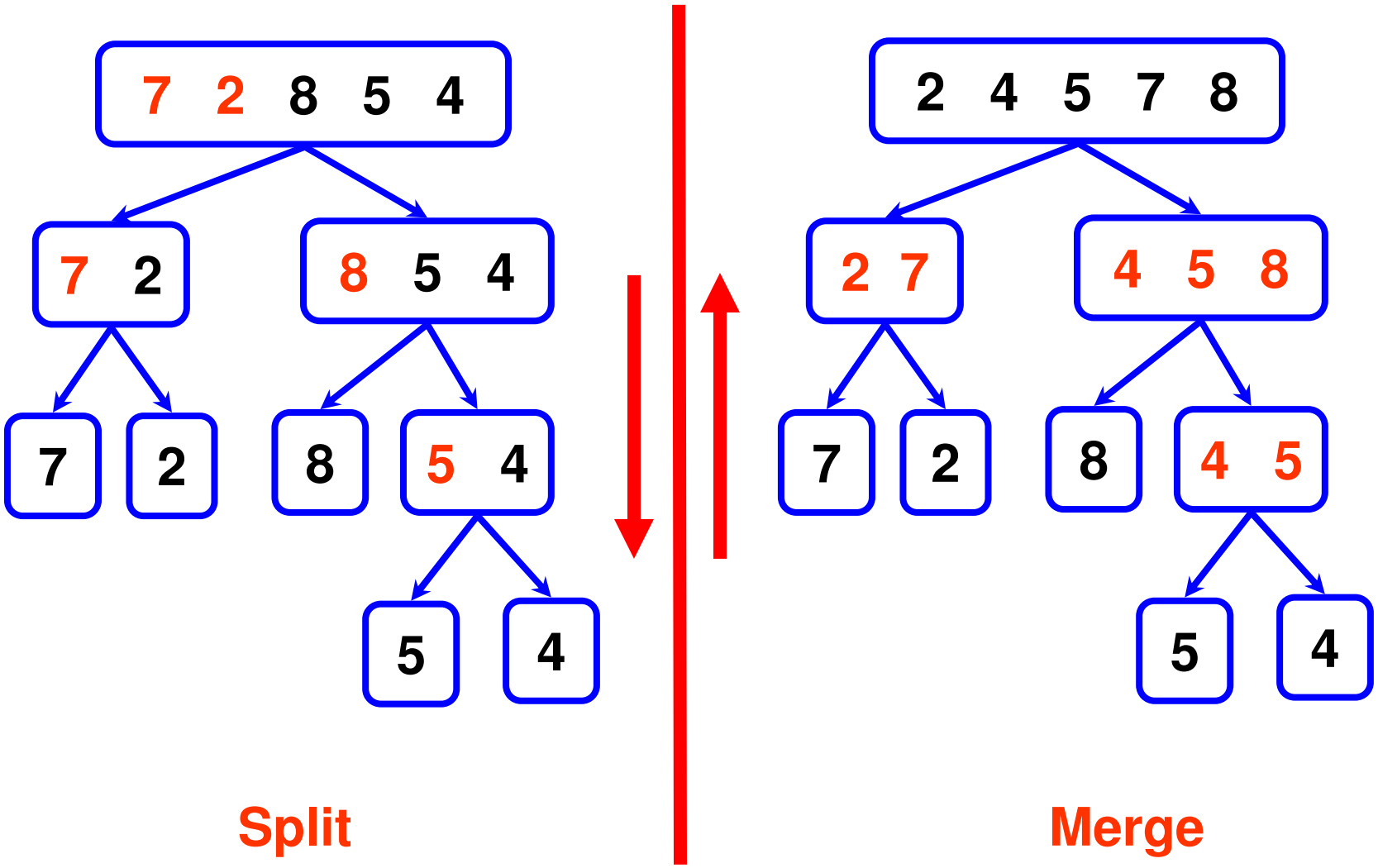
Merge Sort

- Approach
  - Partition list of elements into 2 lists
  - Recursively sort both lists
  - Given 2 sorted lists, merge into 1 sorted list
    - Examine head of both lists
    - Move smaller to end of new list
- Performance
  - O(n log(n)) average / worst case

Merge Example

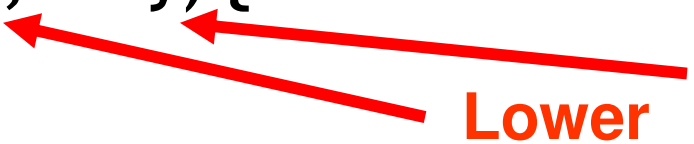


Merge Sort Example



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

Upper end of 2nd array region

Lower end of 1st array region

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

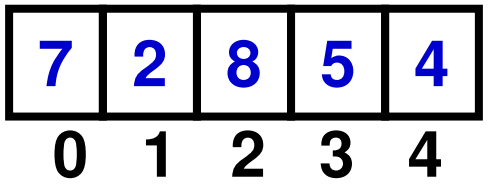
Copy merged array back

Counting Sort

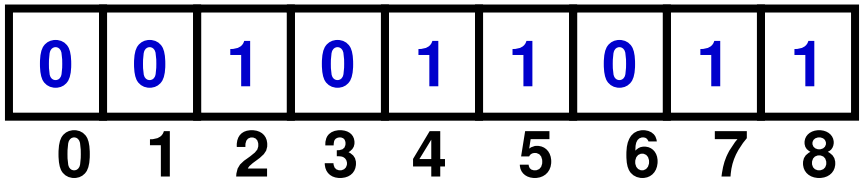
- Approach
  - Sorts keys with values over range 0..k
  - Count number of occurrences of each key
  - Calculate # of keys < each key
  - Place keys in sorted location using # keys counted
    - If there are x keys < y
    - Put y in xth position
    - increment position in which additional copies of y will be stored x
- Properties
  - O(n + k) average / worst case

Counting Sort Example

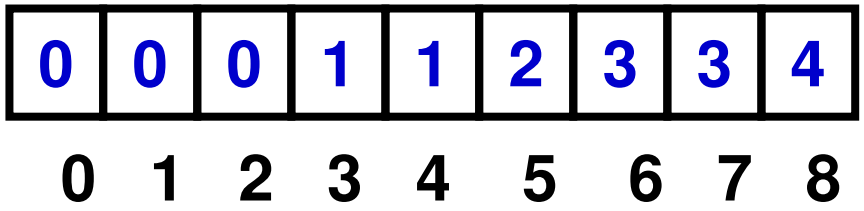
- Original list



- Count

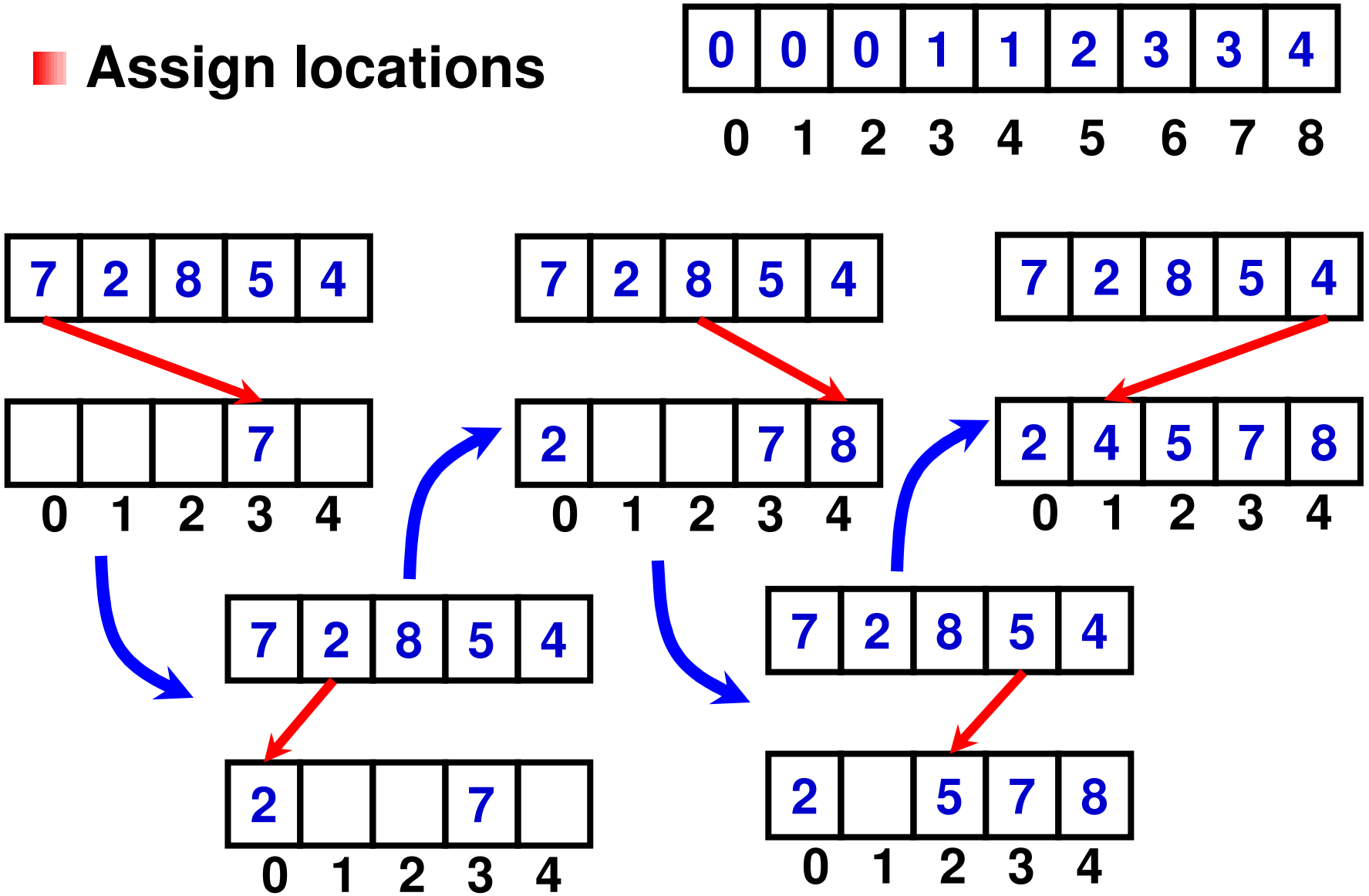


- Calculate # keys < value



Counting Sort Example

- Assign locations



Counting Sort Code

```
void countSort(int[] a, int k) {  // keys have value 0…k
    int[] b = new int[a.length]; int[] c = new int[k+1];
    for (int i = 0; i < a.size(); i++)  // count # keys
        c[a[i]]++;
    int count = 0;
    for (int i = 0; i ≤ k; i++) {// calculate # keys < value i
        int tmp = count+c[i];
        c[i] = count;
        count = tmp;
        }
    for (int i = 0; i < a.length; i++)
        b[c[a[i]]++] = a[i];          // move key to location
    for (i = 0; i < a.size(); i++)  // copy sorted list back to a
        a[i] = b[i];
}
```

Bucket (Bin) Sort

- Approach
  - Divide key interval into k equal-sized subintervals
  - Place elements from each subinterval into bucket
  - Sort buckets (using other sorting algorithm)
  - Concatenate buckets in order
- Properties
  - Pick large k so can sort n / k elements in O(1) time
  - O(n) average case
  - O(n2) worst case
    - If most elements placed in same bucket and sorting buckets with O(n2) algorithm

Bucket Sort Example

1. **Original list**
   - 623, 192, 144, 253, 152, 752, 552, 231
2. **Bucket based on 1st digit, then sort bucket**
   - 192, 144, 152 ⇒ 144, 152, 192
   - 253, 231 ⇒ 231, 253
   - 552 ⇒ 552
   - 623 ⇒ 623
   - 752 ⇒ 752
3. **Concatenate buckets**
   - 144, 152, 192 | 231, 253 | 552 | 623 | 752

Radix Sort

- **Approach**
  1. Decompose key C into components C_1, C_2, … C_d
     - Component d is least significant
     - Each component has values over range 0..k
  2. For each key component i = d down to 1
     - Apply linear sort based on component C_i (sort must be stable)
  - Example key components
    - Letters (string), digits (number)
- **Properties**
  - $O(d \times (n+k)) \approx O(n)$ average / worst case

Radix Sort Example

1. **Original list**
   - 623, 192, 144, 253, 152, 752, 552, 231
2. **Sort on 3rd digit** (counting sort from 0-9)
   - 231, 192, 152, 752, 552, 623, 253, 144
3. **Sort on 2nd digit** (counting sort from 0-9)
   - 623, 231, 144, 152, 752, 552, 253, 192
4. **Sort on 1st digit** (counting sort from 0-9)
   - 144, 152, 192, 231, 253, 552, 623, 752

Compare with: counting sort from 192-752

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))		√
Counting		O(n)	O(n)		√
Bucket		O(n)	$O(n^2)$		√
Radix		O(n)	O(n)		√

Sorting Summary

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