

# Sorting Algorithms

---



**Nelson Padua-Perez**  
**Chau-Wen Tseng**

**Department of Computer Science**  
**University of Maryland, College Park**

## Overview

### ■ Comparison sort

- Bubble sort
  - Selection sort
  - Tree sort
  - Heap sort
  - Quick sort
  - Merge sort
- }  $O(n^2)$
- }  $O(n \log(n))$

### ■ Linear sort

- Counting sort
  - Bucket (bin) sort
  - Radix sort
- }  $O(n)$

# Sorting

## ■ Goal

- Arrange elements in **predetermined** order
  - Based on **key** for each element
- Derived from ability to **compare** two keys by size

## ■ Properties

- **Stable**  $\Rightarrow$  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

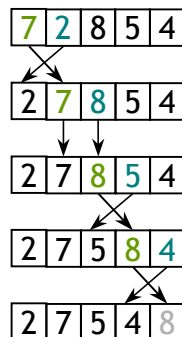
1. Iteratively sweep through shrinking portions of list
2. Swap element  $x$  with its right neighbor if  $x$  is larger

## ■ Performance

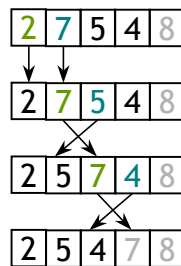
- $O(n^2)$  average / worst case

## Bubble Sort Example

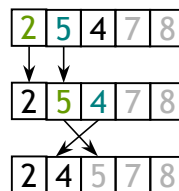
### Sweep 1



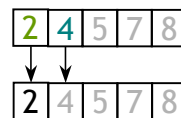
### Sweep 2



### Sweep 3



### Sweep 4



## 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]) {  
                int temp = a[inner];  
                a[inner] = a[inner + 1];  
                a[inner + 1] = temp;  
            }  
        }  
    }  
}
```

**Sweep through array** (points to the inner loop)

**Swap with right neighbor if larger** (points to the if condition)

## Selection Sort

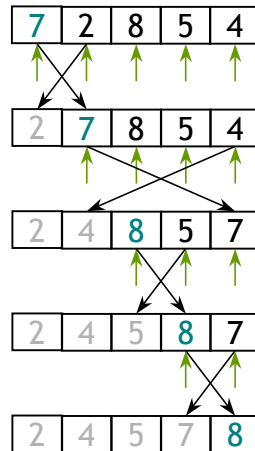
### Approach

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;
            }
        }
        int temp = a[outer];
        a[outer] = a[min];
        a[min] = temp;
    }
}
```

**Sweep through array** (points to the inner loop)

**Find smallest element** (points to the if statement)

**Swap with smallest element found** (points to the swap code)

## Tree Sort

### ■ Approach

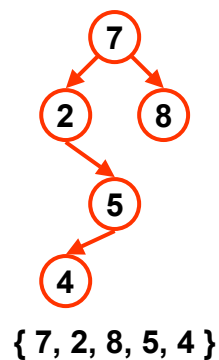
1. Insert elements in binary search tree
2. List elements using **inorder** traversal

### ■ 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



## Heap Sort

### ■ Approach

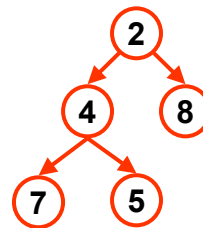
1. Insert elements in heap
2. Remove smallest element in heap, repeat
3. List elements in order of removal from heap

### ■ Performance

- $O(n \log(n))$  average / worst case

### ■ Example

Heap



{ 7, 2, 8, 5, 4 }

## Quick Sort

### ■ Approach

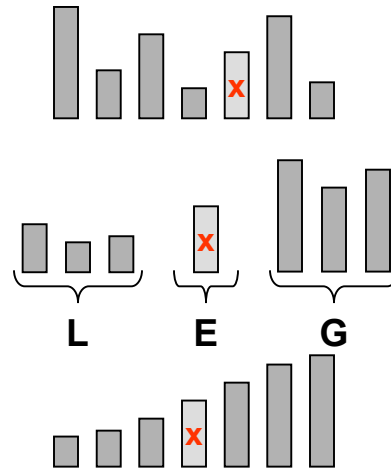
1. Select pivot value (near median of list)
  2. Partition elements (into 2 lists) using **pivot** value
  3. Recursively sort both resulting lists
  4. Concatenate resulting lists
- For efficiency pivot needs to partition list evenly

### ■ 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

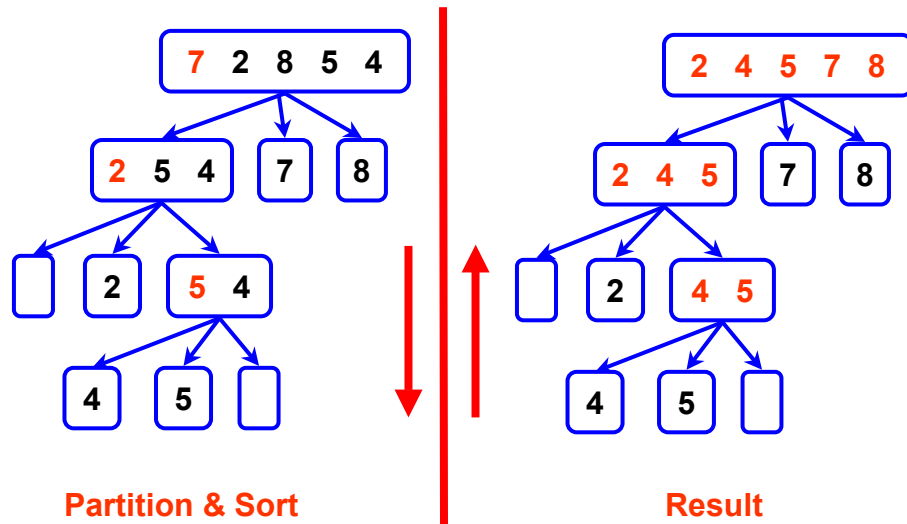
```
void quickSort(int[] a, int x, int y) {
    int pivotIndex;
    if ((y - x) > 0) {
        pivotIndex = partionList(a, x, y);
        quickSort(a, x, pivotIndex - 1);
        quickSort(a, pivotIndex+1, y);
    }
}

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

Lower end of array region to be sorted

Upper end of array region to be sorted

## 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

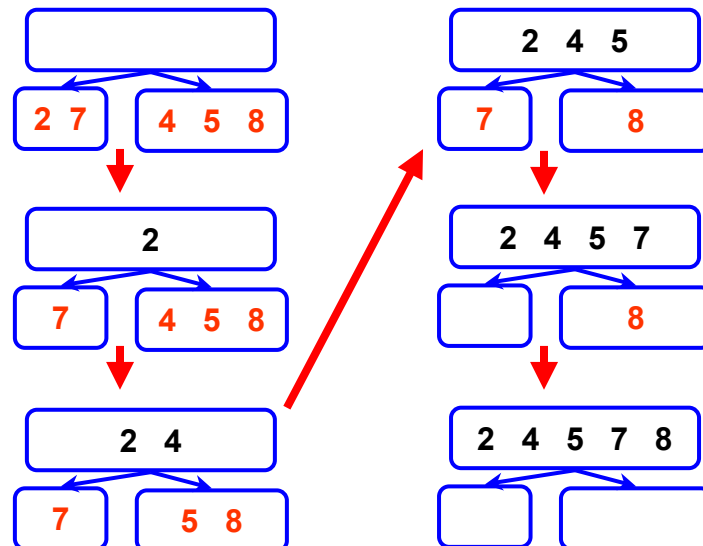
## ■ Approach

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 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 1<sup>st</sup> array region

Lower end of 1<sup>st</sup> array region

Upper end of 2<sup>nd</sup> 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

1. Sorts keys with values over range 0..k
2. Count number of occurrences of each key
3. Calculate # of keys  $\leq$  each key
4. Place keys in sorted location using # keys counted
  - If there are  $x$  keys  $\leq$  key  $y$
  - Put  $y$  in  $x^{\text{th}}$  position
  - Decrement  $x$  in case more instances of key  $y$

### ■ Properties

- $O(n + k)$  average / worst case

## Counting Sort Example

■ Original list

7	2	8	5	4
0	1	2	3	4

■ Count

0	0	1	0	1	1	0	1	1
0	1	2	3	4	5	6	7	8

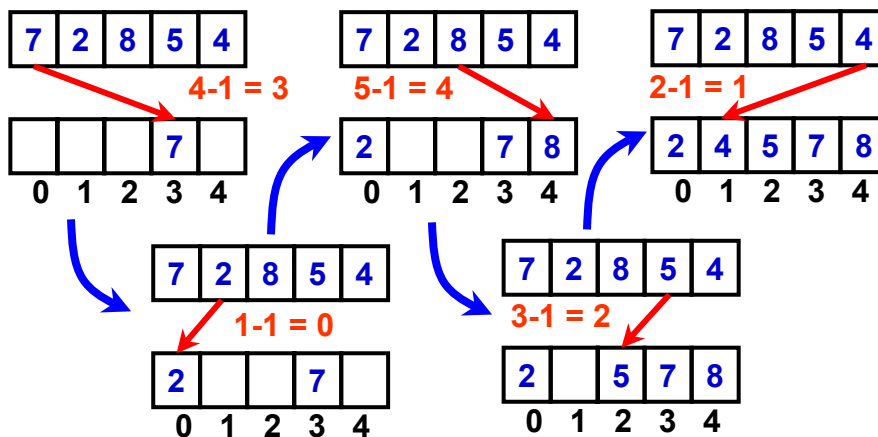
■ Calculate # keys  $\leq$  value

0	0	1	1	2	3	3	4	5
0	1	2	3	4	5	6	7	8

## Counting Sort Example

■ Assign locations

0	0	1	1	2	3	3	4	5
0	1	2	3	4	5	6	7	8



## Counting Sort Code

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

## Bucket (Bin) Sort

### ■ Approach

1. Divide key interval into **k** equal-sized subintervals
2. Place elements from each subinterval into **bucket**
3. Sort buckets (using other sorting algorithm)
4. Concatenate buckets in order

### ■ Properties

- Pick large **k** so can sort  $n / k$  elements in  $O(1)$  time
- $O(n)$  average case
- $O(n^2)$  worst case
  - If most elements placed in same bucket and sorting buckets with  $O(n^2)$  algorithm

## Bucket Sort Example

1. Original list
  - 623, 192, 144, 253, 152, 752, 552, 231
2. Bucket based on 1<sup>st</sup> 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  $C_1, C_2, \dots, C_d$ 
    - 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 3<sup>rd</sup> digit** (counting sort from 0-9)
  - 231, 192, 152, 752, 552, 623, 253, 144
3. **Sort on 2<sup>nd</sup> digit** (counting sort from 0-9)
  - 623, 231, 144, 152, 752, 552, 253, 192
4. **Sort on 1<sup>st</sup> digit** (counting sort from 0-9)
  - 144, 152, 192, 231, 253, 552, 623, 752

Compare with: counting sort from 192-752

## Sorting Properties

Name	Comparison 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