## 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  
  \[ O(n^2) \]

- **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 ⇒ 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

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**

<table>
<thead>
<tr>
<th>Sweep 1</th>
<th>Sweep 2</th>
<th>Sweep 3</th>
<th>Sweep 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 2 8 5 4</td>
<td>2 7 5 4 8</td>
<td>2 5 4 7 8</td>
<td>2 4 5 7 8</td>
</tr>
<tr>
<td>2 7 8 5 4</td>
<td>2 7 5 4 8</td>
<td>2 5 4 7 8</td>
<td>2 4 5 7 8</td>
</tr>
<tr>
<td>2 7 8 5 4</td>
<td>2 5 7 4 8</td>
<td>2 4 5 7 8</td>
<td>2 4 5 7 8</td>
</tr>
<tr>
<td>2 7 5 8 4</td>
<td>2 5 4 7 8</td>
<td>2 4 5 7 8</td>
<td>2 4 5 7 8</td>
</tr>
<tr>
<td>2 7 5 4 8</td>
<td>2 5 4 7 8</td>
<td>2 4 5 7 8</td>
<td>2 4 5 7 8</td>
</tr>
</tbody>
</table>
**Bubble Sort Code**

```c
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;
}
```

**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**

```
Initial: 7 2 8 5 4
Sweep 1: 2 7 8 5 4
Sweep 2: 2 4 8 5 7
Sweep 3: 2 4 5 8 7
```

```
Final: 2 4 5 7 8
```
Selection Sort Code

```java
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**

```
{ 7, 2, 8, 5, 4 }
```

```
\begin{tikzpicture}
    \node (root) {7};
    \node (2) [below left of=root] {2};
    \node (8) [right of=2] {8};
    \node (5) [below left of=8] {5};
    \node (4) [below left of=5] {4};
    \draw (root) -- (2);
    \draw (root) -- (8);
    \draw (2) -- (5);
    \draw (2) -- (4);
\end{tikzpicture}
```
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
  
  ![Heap](image)

  { 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**

```c
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);
    }
}
```

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

Quick Sort Code

```c
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;
}
```
**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**

```
2   4
7
5   8
```

```
2   4   5
7
8
```

```
2   4   5   7
8
```

```
2   4   5   7   8
```
Merge Sort Example

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
}

Merge Sort Code
Merge Sort Code

```c
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];
}
```

Counting Sort

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

**Properties**
- O(n + k) average / worst case
Counting Sort Example

- Original list
  \[\begin{array}{cccc}
  7 & 2 & 8 & 5 \\
  0 & 1 & 2 & 3 \\
  \end{array}\]

- Count
  \[\begin{array}{cccccccc}
  0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
  \end{array}\]

- Calculate # keys ≤ value
  \[\begin{array}{cccccccc}
  0 & 0 & 1 & 1 & 2 & 3 & 3 & 4 & 5 \\
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
  \end{array}\]

Counting Sort Example

- Assign locations
  \[\begin{array}{cccccccc}
  0 & 0 & 1 & 1 & 2 & 3 & 3 & 4 & 5 \\
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
  \end{array}\]
Counting Sort Code

```c
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 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 C₁, C₂, … Cₖ
   - 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 Cᵢ (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

<table>
<thead>
<tr>
<th>Name</th>
<th>Comparison Sort</th>
<th>Avg Case Complexity</th>
<th>Worst Case Complexity</th>
<th>In Place</th>
<th>Can be Stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>✓</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Selection</td>
<td>✓</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Tree</td>
<td>✓</td>
<td>O(n log(n))</td>
<td>O(n^2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td>✓</td>
<td>O(n log(n))</td>
<td>O(n log(n))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quick</td>
<td>✓</td>
<td>O(n log(n))</td>
<td>O(n^2)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Merge</td>
<td>✓</td>
<td>O(n log(n))</td>
<td>O(n log(n))</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Counting</td>
<td></td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Bucket</td>
<td></td>
<td>O(n)</td>
<td>O(n^2)</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Radix</td>
<td></td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Sorting Summary

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