

# Data Structure & Algorithms

Lec 19: Sorting Algorithms

Fall 2017 - University of Windsor  
Dr. Sherif Saad



# Outlines

- Introduction to Sorting
- Bubble Sort
- Selection Sort
- Insertions Sort
- Mergesort
- Quicksort
- Radix sort

# Sorting

- What is sorting?
  - It is the process that organizes a collection of data into either ascending or descending order.
- Types of Sorting
  - An **internal sort** requires that the collection of data fit entirely in the computer's main memory.
  - **External sort** when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.

# Sorting Algorithms

- There are many sorting algorithms such as:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Radix Sort
- There are more than 100 sorting algorithms. The most recent sorting algorithm is gapped insertion sort invented in 2006

# Sorting

- Why sorting?
  - The performance of many algorithms could improve a lot when data are sorted and their implementation become much easier.
- Stability
  - A sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.
  - Stable Sort Algorithms: Insertion sort, Merge Sort, Bubble Sort,
  - Unstable Sort Algorithms: Heap Sort, Quick Sort, etc.

# Analyzing Sorting Algorithms

- In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).
- To analyze a sorting algorithm we count the **number of key comparisons** and the **number of moves**.
- Ignoring other operations does not affect our final result.

# Bubble Sort

- One of the **simplest** sorting algorithms, but **not** one of the most **efficient**.
- It works by successively comparing adjacent elements, interchanging them if they are in the wrong order.
- We perform the basic operation, that is, interchanging a larger element with a smaller one following it, starting at the beginning of the list, for a full pass. We iterate this procedure until the sort is complete.

# Bubble Sort

- The list is divided into two sublists: sorted and unsorted. which are divided by an imaginary wall.
- The smallest element is bubbled from the unsorted list and moved to the sorted sublist.
- After that, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time an element moves from the unsorted part to the sorted part one sort pass is completed.
- Given a list of  $n$  elements, bubble sort requires up to  $n-1$  passes to sort the data.



# Bubble Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 7 | 5 | 9 | 6 | 2 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j],a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Bubble Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 7 | 5 | 9 | 6 | 2 |

Traverse the array from  $i = 0$  to  $i < n-1$

$i = 0$

$j = 4$

sorted = true

$a[j-1] > a[j] \rightarrow \text{swap}(a[j], a[j-1])$

sorted = false

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 7 | 5 | 9 | 2 | 6 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Bubble Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 7 | 5 | 9 | 2 | 6 |

Traverse the array from  $i = 0$  to  $i < n-1$

$i = 0$

$j = 3$

sorted = false

$a[j-1] > a[j] \rightarrow \text{swap}(a[j], a[j-1])$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 7 | 5 | 2 | 9 | 6 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Bubble Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 7 | 5 | 2 | 9 | 6 |

Traverse the array from  $i = 0$  to  $i < n-1$

$i = 0$

$j = 2$

sorted = false

$a[j-1] > a[j] \rightarrow \text{swap}(a[j], a[j-1])$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 7 | 2 | 5 | 9 | 6 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Bubble Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 7 | 2 | 5 | 9 | 6 |

Traverse the array from  $i = 0$  to  $i < n-1$

$i = 0$

$j = 1$

sorted = false

$a[j-1] > a[j] \rightarrow \text{swap}(a[j], a[j-1])$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 7 | 5 | 9 | 6 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Bubble Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 7 | 5 | 9 | 6 |

Traverse the array from  $i = 0$  to  $i < n-1$

$i = 1$

$j = 4$

sorted = true

$a[j-1] > a[j] \rightarrow \text{swap}(a[j], a[j-1])$

sorted = false

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 7 | 5 | 6 | 9 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Bubble Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 7 | 5 | 6 | 9 |

Traverse the array from  $i = 0$  to  $i < n-1$

$i = 1$

$j = 3$

sorted = false

$a[j-1] < a[j] \rightarrow$  do nothing

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 7 | 5 | 6 | 9 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Bubble Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 7 | 5 | 6 | 9 |

Traverse the array from  $i = 0$  to  $i < n-1$

$i = 1$

$j = 2$

sorted = false

$a[j-1] > a[j] \rightarrow \text{swap}(a[j], a[j-1])$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 5 | 7 | 6 | 9 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```



# Bubble Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 5 | 7 | 6 | 9 |

Traverse the array from  $i = 0$  to  $i < n-1$

$i = 2$

$j = 4$

sorted = true

$a[j-1] > a[j] \rightarrow$  do nothing

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 5 | 7 | 6 | 9 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Bubble Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 7 | 6 | 9 |

Traverse the array from  $i = 0$  to  $i < n-1$

$i = 2$

$j = 3$

sorted = true

$a[j-1] > a[j] \rightarrow \text{swap}(a[j], a[j-1])$

sorted = false

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 6 | 7 | 9 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Bubble Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 5 | 6 | 7 | 9 |

Traverse the array from  $i = 0$  to  $i < n-1$

$i = 3$

$j = 4$

sorted = true

$a[j-1] > a[j] \rightarrow \text{swap}(a[j], a[j-1])$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 5 | 6 | 7 | 9 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Bubble Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 6 | 7 | 9 |

Traverse the array from  $i = 0$  to  $i < n-1$

$i = 4$

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 6 | 7 | 9 |

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n-1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Bubble Sort Analysis

- **Best Case:**  $O(n)$ 
  - Array is already sorted in ascending order.
  - The number of moves: 0  $\rightarrow O(1)$
  - The number of key comparisons:  $(n-1) \rightarrow O(n)$
- **Worst Case:**  $O(n^2)$ 
  - Array is in reverse order
  - Outer loop is executed  $n-1$  times,
  - The number of moves:  $3 \cdot (1+2+\dots+n-1) = 3 \cdot n \cdot (n-1)/2 \rightarrow O(n^2)$
  - The number of key comparisons:  $(1+2+\dots+n-1) = n \cdot (n-1)/2 \rightarrow O(n^2)$

# Bubble Sort Analysis

- Average Case:  $O(n^2)$ 
  - We have to look at all possible initial data organizations.
- The run time complexity of bubble sort is  $O(n^2)$

Is it stable sorting algorithm or not?

# Bubble Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 6 | 7 | 9 |

**Best Case:  $O(n)$**

```
1. void bubbleSort(Item a[], int n)
2. {
3.     bool sorted = false;
4.     for (int i = 0; (i < n - 1) && !sorted; i++){
5.         sorted = true;
6.         for (int j = n - 1; j > i; j--){
7.             if (a[j-1] > a[j]){
8.                 swap(a[j], a[j-1]);
9.                 sorted = false;
10.            }
11.        }
12.    }
```

# Selection Sort

- The list is divided into two sublists: sorted and unsorted. which are divided by an imaginary wall.
- We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.
- After each selection and swapping, the imaginary wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.



# Selection Sort

- Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.
- A list of  $n$  elements requires  $n-1$  passes to completely rearrange the data.

Is selection sort a stable sorting algorithm?

# Selection Sort

```
1. void selectionSort( Item a[ ], int n) {  
2.   for (int i = 0; i < n-1; i++) {  
3.     for (int j = i+1; j < n; j++)  
4.       if (a[j] < a[i])  
5.         swap(a[i], a[j]);  
6.   }  
7. }
```

# Selection Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 7 | 5 | 2 | 9 | 6 |

Original List

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 7 | 9 | 6 |

After pass 1

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 7 | 9 | 6 |

After pass 2

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 6 | 9 | 7 |

After pass 3

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 6 | 7 | 9 |

After pass 4

# Selection Sort Analysis

- The **outer** for loop executes  **$n-1$**  times.
- We invoke swap function once at each iteration.
  - Total Swaps:  **$n-1$**
  - Total Moves:  **$3 * (n-1)$**
- The inner for loop executes the size of the unsorted part minus 1
- In each iteration of the inner loop we make one key comparison. The of key comparisons =  **$1+2+...+n-1 = n*(n-1)/2$**

# Selection Sort Analysis

- The **best case**, the **worst case**, and the **average case** of the selection sort algorithm are same. → all of them are  **$O(n^2)$**
- This means that the behavior of the selection sort algorithm does not depend on the **initial organization of data**.
- A selection sort could be a good choice if data moves are costly but key comparisons are not costly (small keys, large records). Since the number of moves is  $O(n)$  while key comparisons require  $O(n^2)$

# Selection Sort Analysis

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 6 | 7 | 9 |

**Best = Average = Worst Case =  $O(n^2)$**

```
1. void selectionSort( Item a[ ], int n) {  
2.   for (int i = 0; i < n-1; i++) {  
3.     for (int j = i+1; j < n; j++)  
4.       if (a[j] < a[i])  
5.         swap(a[i], a[j]);  
6.   }  
7. }
```

# Insertion Sort

- The list is divided into two sublists: sorted and unsorted. which are divided by an imaginary wall.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- A list of  $n$  elements will take at most  $n-1$  passes to sort the data.
- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.

# Insertion Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 7 | 5 | 9 | 6 | 2 |

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```



# Insertion Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 7 | 5 | 9 | 6 | 2 |

Traverse the array from  $i = 1$  to  $i < n$

$i = 1$

$j = 1$

$tmp = a[i] = a[1] = 5$

$tmp < a[j-1]$  ( $5 < 7$ ),  $a[j] = a[j-1]$  and  $a[j] = tmp$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 7 | 9 | 6 | 2 |

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```

# Insertion Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 7 | 9 | 6 | 2 |

Traverse the array from  $i = 1$  to  $i < n$

$i = 2$

$j = 2$

$tmp = a[i] = a[2] = 9$

$tmp < a[j-1]$  ( $9 < 7$ ), do nothing

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 7 | 9 | 6 | 2 |

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```

# Insertion Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 7 | 9 | 6 | 2 |

Traverse the array from  $i = 1$  to  $i < n$

$i = 3$

$j = 3$

$tmp = a[i] = a[3] = 6$

$tmp < a[j-1]$  ( $6 < 9$ ),  $a[j] = a[j-1]$  and  $a[j] = tmp$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 7 | 6 | 9 | 2 |

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```

# Insertion Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 7 | 6 | 9 | 2 |

Traverse the array from  $i = 1$  to  $i < n$

$i = 3$

$j = 2$

$tmp = 6$

$tmp < a[j-1]$  ( $6 < 7$ ),  $a[j] = a[j-1]$  and  $a[j] = tmp$

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 9 | 2 |

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```

# Insertion Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 9 | 2 |

Traverse the array from  $i = 1$  to  $i < n$

$i = 3$

$j = 1$

$tmp = 6$

$tmp < a[j-1]$  ( $6 < 5$ ), do nothing

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 9 | 2 |

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```

# Insertion Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 9 | 2 |

Traverse the array from  $i = 1$  to  $i < n$

$i = 3$

$j = 3$

$tmp = a[i] = a[3] = 9$

$tmp < a[j-1]$  ( $9 < 7$ ), do nothing

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 7 | 6 | 9 | 2 |

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```

# Insertion Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 9 | 2 |

Traverse the array from  $i = 1$  to  $i < n$

$i = 4$

$j = 4$

$tmp = a[i] = a[4] = 2$

$tmp < a[j-1]$  ( $2 < 9$ ),  $a[j] = a[j-1]$  and  $a[j] = tmp$

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 2 | 9 |

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```

# Insertion Sort

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 6 | 7 | 2 | 9 |

Traverse the array from  $i = 1$  to  $i < n$

$i = 4$

$j = 3$

$tmp = 2$

$tmp < a[j-1]$  ( $2 < 6$ ),  $a[j] = a[j-1]$  and  $a[j] = tmp$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 6 | 2 | 7 | 9 |

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```



# Insertion Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 2 | 7 | 9 |

Traverse the array from  $i = 1$  to  $i < n$

$i = 4$

$j = 2$

$tmp = 2$

$tmp < a[j-1]$  ( $2 < 7$ ),  $a[j] = a[j-1]$  and  $a[j] = tmp$

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 2 | 6 | 7 | 9 |

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```

# Insertion Sort

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 2 | 6 | 7 | 9 |

Traverse the array from  $i = 1$  to  $i < n$

$i = 4$

$j = 1$

$tmp = 2$

$tmp < a[j-1]$  ( $2 < 7$ ),  $a[j] = a[j-1]$  and  $a[j] = tmp$

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 6 | 7 | 9 |

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```

# Insertion Sort Analysis

- **Best Case:**  $O(n)$ 
  - Array is already sorted in ascending order. Inner loop will not be executed
  - The number of moves:  $2(n-1) = O(n)$
  - The number of key comparisons:  $(n-1) = O(n)$
- **Worst Case:**  $O(n^2)$ 
  - Array in reverse order
  - Inner loop executed  $i-1$  times, where  $i = 2, 3, \dots, n$
  - Number of moves is  $2(n-1) + (1+2+\dots+n-1) = 2(n-1) + n(n-1)/2 = O(n^2)$
  - Number of key comparisons is  $(1+2+\dots+n-1) = n(n-1)/2 = O(n^2)$
- **Average Case :**  $O(n^2)$

# Insertion Sort Analysis

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 6 | 7 | 9 |

**Best Case:**  $O(n)$

```
1. void insertionSort(Item a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         Item tmp = a[i];
5.         for (int j=i; j>0 && tmp < a[j-1]; j--)
6.             a[j] = a[j-1];
7.         a[j] = tmp;
8.     }
9. }
```

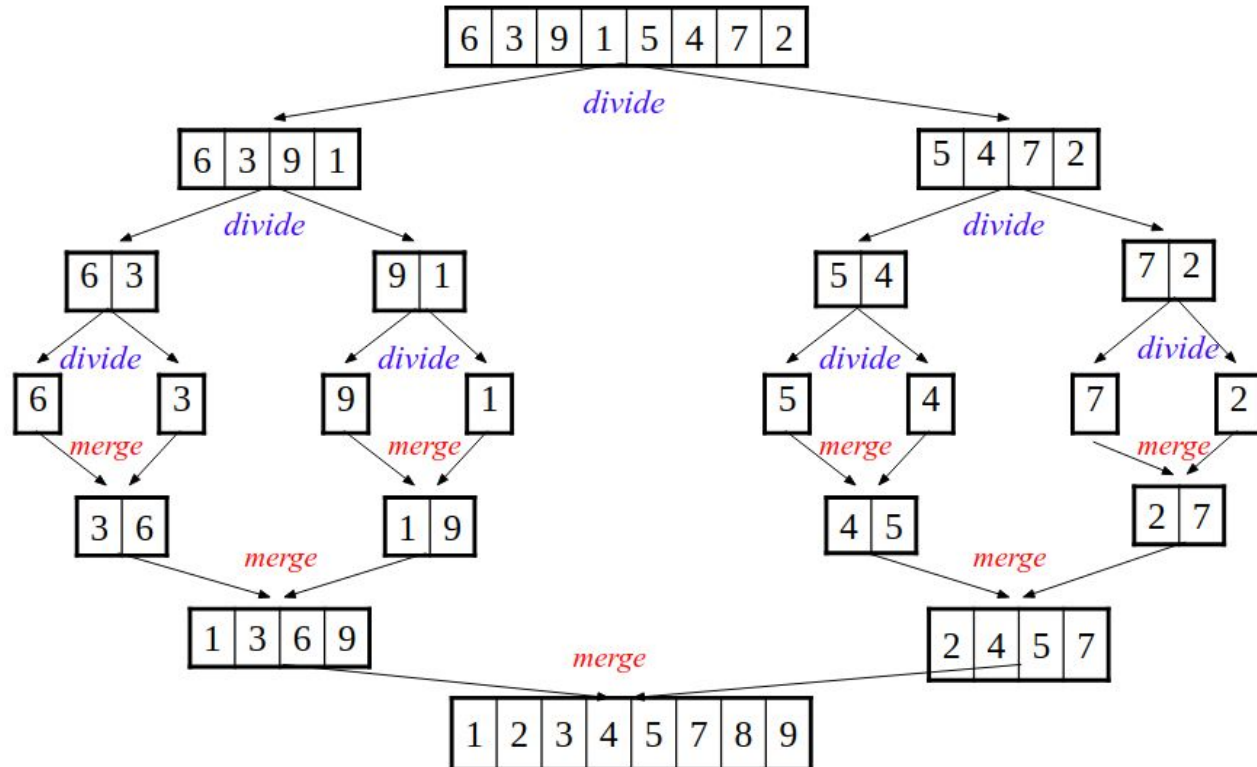
# Merge Sort

- It applies **divide-and-conquer** algorithm design technique.
- It is an efficient sorting algorithms comparing to other algorithms such as selections, insertion, and bubble sort.
- It is a recursive algorithm
  - It divides the list into smaller sublists
  - Sort each sublist separately
  - Then merge the sorted sublist into one sorted array

# Merge Sort

```
1. void mergesort(Item theArray[], int first, int last) {  
2.     if (first < last) {  
3.         int mid = (first + last)/2;    // index of midpoint  
4.         mergesort(theArray, first, mid);  
5.         mergesort(theArray, mid+1, last);  
6.         // merge the sub lists  
7.         merge(theArray, first, mid, last);  
8.     }  
9. }
```

# Merge Sort



# Merge Sort

theArray: 

|   |   |   |   |   |
|---|---|---|---|---|
| 8 | 1 | 4 | 3 | 2 |
|---|---|---|---|---|

Divide the array in half

|   |   |   |
|---|---|---|
| 1 | 4 | 8 |
|---|---|---|

|   |   |
|---|---|
| 2 | 3 |
|---|---|

Sort the halves

Temporary array  
tempArray:

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 8 |
|---|---|---|---|---|

Merge the halves:

- a.  $1 < 2$ , so move 1 from left half to tempArray
- b.  $4 > 2$ , so move 2 from right half to tempArray
- c.  $4 > 3$ , so move 3 from right half to tempArray
- d. Right half is finished, so move rest of left half to tempArray

theArray:

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 8 |
|---|---|---|---|---|

Copy temporary array back into  
original array

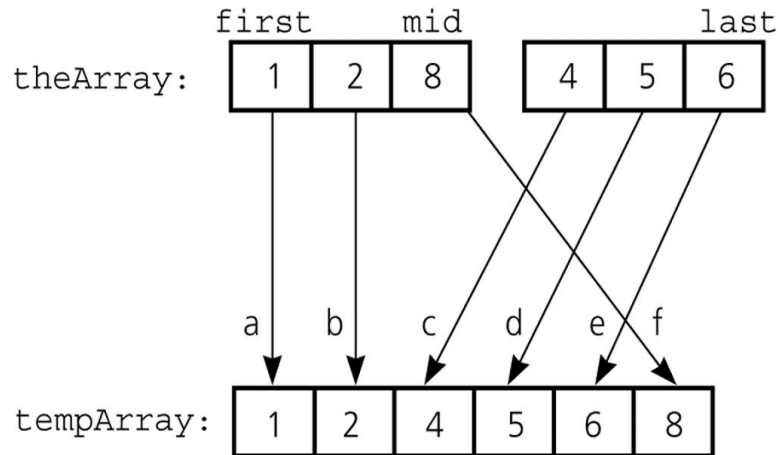


# Merge Sort Analysis

- Let us assume we are merging two sorted arrays of size  $m$
- **Best-case:**
  - All the elements in the first array are smaller (or larger) than all the elements in the second array.
  - The number of moves:  $2m + 2m$
  - The number of key comparisons:  $m$
- **Worst-case:**
  - The number of moves:  $2m + 2m$
  - The number of key comparisons:  $2m-1$

# Merge Sort Analysis

Example of the **worst-case** instance of the merge step in *mergesort*

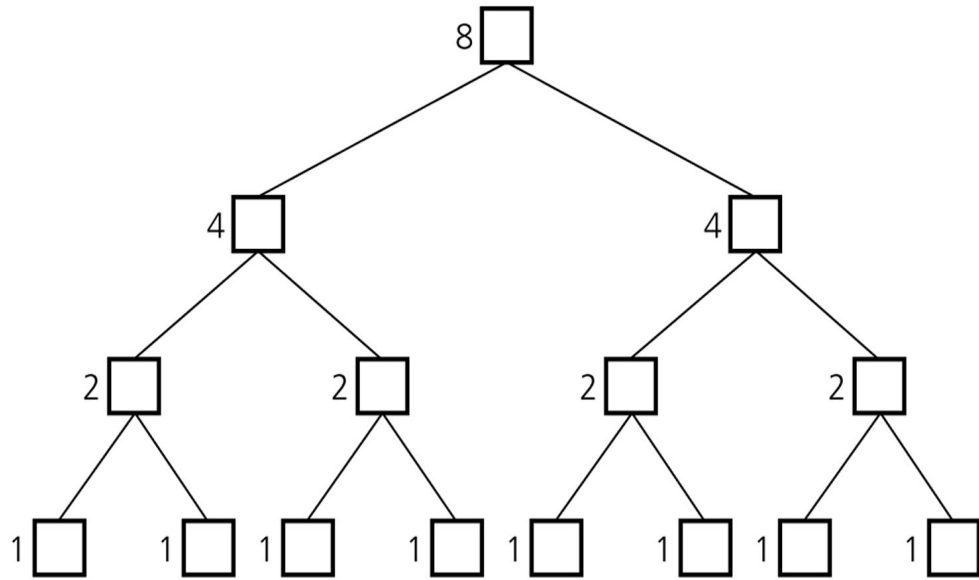


Merge the halves:

- a.  $1 < 4$ , so move 1 from theArray[first..mid] to tempArray
- b.  $2 < 4$ , so move 2 from theArray[first..mid] to tempArray
- c.  $8 > 4$ , so move 4 from theArray[mid+1..last] to tempArray
- d.  $8 > 5$ , so move 5 from theArray[mid+1..last] to tempArray
- e.  $8 > 6$ , so move 6 from theArray[mid+1..last] to tempArray
- f. theArray[mid+1..last] is finished, so move 8 to tempArray

# Mergesort Analysis

Assume we have an array of size 8, the number of recursive calls to mergesort



Level 0: mergesort 8 items

Level 1: 2 calls to mergesort with 4 items each

Level 2: 4 calls to mergesort with 2 items each

Level 3: 8 calls to mergesort with 1 item each

# Mergesort Analysis

- The number of key comparisons,  $T(n) = n * \log_2 n - n + 1$
- Mergesort is extremely efficient algorithm with respect to time.
- Both worst case and average cases are  **$O(n * \log_2 n)$**
- But, mergesort requires **an extra array** whose size equals to the size of the original array.

# Quicksort

- Like mergesort, Quicksort is also based on the divide-and-conquer paradigm.
- But it uses this technique in a somewhat opposite manner, as all the hard work is done before the recursive calls.
- It works as follows:
  - First, it partitions an array into two parts,
  - Then, it sorts the parts independently,
  - Finally, it combines the sorted subsequences by a simple concatenation.

# Quicksort

1. **Divide:** to partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the **pivot**. Then we partition the elements so that all those with values less than the pivot come in one sublist and all those with greater values come in another.
2. **Recursion:** Recursively sort the sublists separately.
3. **Conquer:** Put the sorted sublists together.

# Quicksort

There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

# Quicksort

```
1. quickSort(arr[], low, high)  {  
2.     if (low < high){  
3.         // select pi as the pivot point  
4.         pi = partition(arr, low, high);  
5.         quickSort(arr, low, pi - 1); // Before pi  
6.         quickSort(arr, pi + 1, high); // After pi  
7.     }  
8. }
```



# Quicksort

```
1. partition (arr[ ], low, high){  
2.     pivot = arr[high];  
3.     i = (low - 1) // Index of smaller element  
4.     for (j = low; j <= high- 1; j++){  
5.         // If current element is smaller than or  
6.         // equal to pivot  
7.         if (arr[j] <= pivot){  
8.             i++; // increment index of smaller element  
9.             swap arr[i] and arr[j]  
10.        }  
11.    }  
12.    swap arr[i + 1] and arr[high])  
13.    return (i + 1)  
14. }
```

# Quicksort


| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
|----|----|----|----|----|----|----|
| 10 | 80 | 30 | 90 | 40 | 50 | 70 |

low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, **i = -1**

# Quicksort

| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
|----|----|----|----|----|----|----|
| 10 | 80 | 30 | 90 | 40 | 50 | 70 |



Traverse elements from  $j = \text{low}$  to  $\text{high}-1$

$j = 0$


Since  $\text{arr}[j] \leq \text{pivot}$ , do  $i++$  and ***swap(arr[i], arr[j])***

$i = 0$

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 10 | 80 | 30 | 90 | 40 | 50 | 70 |
|----|----|----|----|----|----|----|

# Quicksort

| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
|----|----|----|----|----|----|----|
| 10 | 80 | 30 | 90 | 40 | 50 | 70 |



Traverse elements from  $j = \text{low}$  to  $\text{high}-1$

$j = 1$


Since  $\text{arr}[j] > \text{pivot}$ , nothing will change, and

$i = 0$

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 10 | 80 | 30 | 90 | 40 | 50 | 70 |
|----|----|----|----|----|----|----|

# Quicksort

| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
|----|----|----|----|----|----|----|
| 10 | 80 | 30 | 90 | 40 | 50 | 70 |



Traverse elements from  $j = \text{low}$  to  $\text{high}-1$

$j = 2$


Since  $\text{arr}[j] \leq \text{pivot}$ , do  $i++$  and ***swap(arr[i], arr[j])***

$i = 1$

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 10 | 30 | 80 | 90 | 40 | 50 | 70 |
|----|----|----|----|----|----|----|

# Quicksort

| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
|----|----|----|----|----|----|----|
| 10 | 30 | 80 | 90 | 40 | 50 | 70 |



Traverse elements from  $j$  = low to high-1

$j = 3$


Since  $arr[j] > pivot$ , nothing will change

$i = 1$

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 10 | 30 | 80 | 90 | 40 | 50 | 70 |
|----|----|----|----|----|----|----|

# Quicksort

| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
|----|----|----|----|----|----|----|
| 10 | 30 | 80 | 90 | 40 | 50 | 70 |



Traverse elements from  $j$  = low to high-1

$j = 4$


Since  $arr[j] \leq pivot$ , do  $i++$  and ***swap(arr[i], arr[j])***

$i = 2$

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 10 | 30 | 40 | 90 | 80 | 50 | 70 |
|----|----|----|----|----|----|----|

# Quick Sort

| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
|----|----|----|----|----|----|----|
| 10 | 30 | 40 | 90 | 80 | 50 | 70 |



Traverse elements from  $j = \text{low}$  to  $\text{high}-1$

$j = 5$

Since  $\text{arr}[j] \leq \text{pivot}$ , do  $i++$  and ***swap(arr[i], arr[j])***

$i = 3$

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 10 | 30 | 40 | 50 | 80 | 90 | 70 |
|----|----|----|----|----|----|----|



# Quicksort


j

j = 5

i = 3

| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
|----|----|----|----|----|----|----|
| 10 | 30 | 40 | 50 | 80 | 90 | 70 |

=



high-1

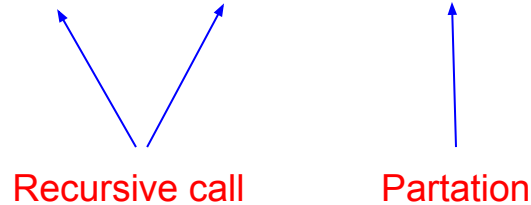
```
12. swap arr[i + 1] and arr[high])
13. return (i + 1)
```

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 10 | 30 | 40 | 50 | 70 | 90 | 80 |
|----|----|----|----|----|----|----|

# Quicksort Analysis

The Quicksort run time complexity could be expressed as:

$$T(n) = T(m) + T(n-m-1) + \theta(n)$$



Here, **n** is the total number of elements (keys) in the array and **m** is the number of keys less than the pivot

# Quicksort Analysis

**Worst Case:** when we pick greatest or smallest element as pivot.

For example in case the last element is picked as the pivot. The worst case exist when the array is sorted in decreasing order

| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
|----|----|----|----|----|----|----|
| 90 | 80 | 70 | 50 | 40 | 30 | 10 |

↑  
**pivot**

$$T(n) = T(0) + T(n-1) + O(n)$$

$$T(n) = O(n^2)$$

# Quicksort Analysis

**Best Case:** when we pick middle element as pivot.

For example in case the last element is picked as the pivot. The best case exists when this last element is the middle element

| 0  | 1  | 2  | 3  | 4  | 5  | 6  |
|----|----|----|----|----|----|----|
| 90 | 40 | 70 | 10 | 80 | 30 | 50 |



**pivot**

$$T(n) = T(n/2) + T(n - n/2 - 1) + O(n)$$

$$T(n) = 2T(n/2) + O(n)$$

$$T(n) = O(n \log n)$$

# Radix Sort

- Treats each data item as a character string or a string of digits.
- How it works?
  - Take the least significant digit of each element
  - Sort the list of elements based on that digit, but keep the order of the elements with the same digit ( why??)
  - Repeat the sort with each more significant digit

# Radix Sort

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| mom | dad | god | fat | bad | cat | mad | pat | bar | him |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| dad | god | bad | mad | mom | him | bar | fat | cat | pat |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| dad | bad | mad | bar | fat | cat | pat | him | god | mom |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| bad | bar | cat | dad | fat | god | him | mad | mom | pat |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

# Radix Sort

|    |     |    |     |   |    |     |
|----|-----|----|-----|---|----|-----|
| 52 | 537 | 31 | 211 | 7 | 22 | 100 |
|----|-----|----|-----|---|----|-----|

|     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|
| 052 | 537 | 031 | 211 | 007 | 022 | 100 |
|-----|-----|-----|-----|-----|-----|-----|

|     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|
| 100 | 031 | 211 | 052 | 022 | 537 | 007 |
|-----|-----|-----|-----|-----|-----|-----|

|     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|
| 100 | 007 | 211 | 022 | 031 | 537 | 052 |
|-----|-----|-----|-----|-----|-----|-----|

|     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|
| 007 | 022 | 031 | 052 | 100 | 211 | 537 |
|-----|-----|-----|-----|-----|-----|-----|

# Radix Sort Analysis

The run time of radix sort is:

$$T(n) = 2 * n * m$$

$$T(n) = O(nm)$$

Where  $n$  is the number of elements and  $m$  is the number of digits or characters per element.



# Comparison of Sorting Algorithms

| Algorithm      | Worst Case    | Average Case  | Stability |
|----------------|---------------|---------------|-----------|
| Selection Sort | $O(n^2)$      | $O(n^2)$      | No        |
| Bubble Sort    | $O(n^2)$      | $O(n^2)$      | Yes       |
| Insertion Sort | $O(n^2)$      | $O(n^2)$      | Yes       |
| Mergesort      | $O(n \log n)$ | $O(n \log n)$ | Yes       |
| Quicksort      | $O(n^2)$      | $O(n \log n)$ | No        |
| Radix Sort     | $O(mn)$       | $O(mn)$       | Yes       |
| Tree Sort      | $O(n^2)$      | $O(n \log n)$ | No        |
| Heapsort       | $O(n \log n)$ | $O(n \log n)$ | No        |