Sorting

Instructor: Dr. Sunho Lim (Ph.D., Assistant Professor)

Lecture 17

sunho.lim@ttu.edu

Adapted partially from Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning; and Algorithms and Data Structures, Douglas Wilhelm Harder, Mmath

Introduction

- Sorting
  - improve the **efficiency of accessing data**, e.g., increasing or decreasing order, or alphabetical order
  - common measure metrics
    - number of comparisons
    - number of data movements
    - may be difficult to determine exactly → approximations
    - may differ depending on the original state of the data set (e.g., sorted, unsorted, partially sorted)

Elementary Sorting Algorithms: Insertion Sort

- Pseudocode for the insertion sort:

```
insertionsort(data[],n)
  for i = 1 to n-1
    move all elements data[j] greater than data[i] by
one position;
    place data[i] in its proper position;
```

- The array of values → divide into two sets,
  - sorted values vs. unsorted
- Initially, the element with **index 0**
  - sorted set
- The element with **index 1**,
  - the first element of the unsorted set
- Each repetition,
  - pick up the first element in the unsorted set and insert it into the correct position

Elementary Sorting Algorithms: Insertion Sort (cont.)

- Consider an array of integers

39	9	45	63	18	81	108	54	72	36

39	9	45	63	18	81	108	54	72	36	A[0] is the only element in sorted list
9	39	45	63	18	81	108	54	72	36	(Pass 1)
9	39	45	63	18	81	108	54	72	36	(Pass 2)
9	39	45	63	18	81	108	54	72	36	(Pass 3)
9	18	39	45	63	81	108	54	72	36	(Pass 4)
9	18	39	45	63	81	108	54	72	36	(Pass 5)
9	18	39	45	63	81	108	54	72	36	(Pass 6)
9	18	39	45	54	63	81	108	72	36	(Pass 7)
9	18	39	45	54	63	72	81	108	36	(Pass 8)
9	18	36	39	45	54	63	72	81	108	(Pass 9)

Elementary Sorting Algorithms: Insertion Sort (cont.)

- The best case,
  - the array is already **sorted**
  - the first element in the unsorted set compares **only with the last element** of the sorted set
  - O(n)
- The worst case,
  - the array is sorted in the **reverse order**
  - the first element in the unsorted set compares with **almost every element** in the sorted set
  - O(n^2)

Elementary Sorting Algorithms: Selection Sort

- Localize the exchange of array elements
  - finding an unsorted item and putting it in its final location
- Find the **smallest value** in the array
  - place it in the first position
- Find the **second smallest value** in the array
  - place it in the second position
- Repeat until the entire array is sorted

Elementary Sorting Algorithms: Selection Sort (cont.)

- The pseudocode for the algorithm reflects its simplicity:

```
selectionsort(data[],n)
  for i = 0 to n-2
    select the smallest element among data[i], . . . ,
data[n-1];
    swap it with data[i];
```

- Array divided into two sets:
  - elements in the sorted set
  - elements in the unsorted set
- The last value for *i* is *n* – 2
  - since if all items have been looked at and placed except for the last
  - the n-th element has to be the largest

Elementary Sorting Algorithms: Selection Sort (cont.)

-

39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

Elementary Sorting Algorithms: Selection Sort (cont.)

- complexity
  - pass 1:
    - select the element with the smallest value for all n elements
    - **n – 1 comparisons**
  - pass 2
    - Swap the smallest value with the element in the first position, and so on
    - **(n – 1) + (n – 2) + … + 2 + 1**
    - n(n – 1) /2 = O(n^2)

Elementary Sorting Algorithms: Bubble Sort

- On each pass,
  - **compare** pairs of adjacent items and **swap** them if they are in the wrong order
  - repeatedly moving the **largest element** to the **highest index position** of the array
- Continue till the list of unsorted elements exhaust
- The pseudocode of bubble sort:

```
bubblesort(data[],n)
  for i = 0 to n-2
    for j = n-1 down to i+1
      swap items in positions j and j-1 if they are out of
order;
```


Elementary Sorting Algorithms: Bubble Sort (cont.)

- For example,

```
A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:
 (a) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (b) Compare 52 and 29. Since 52 > 29, swapping is done.
     30, 29, 52, 87, 63, 27, 19, 54
 (c) Compare 52 and 87. Since 52 < 87, no swapping is done.
 (d) Compare 87 and 63. Since 87 > 63, swapping is done.
     30, 29, 52, 63, 87, 27, 19, 54
 (e) Compare 87 and 27. Since 87 > 27, swapping is done.
     30, 29, 52, 63, 27, 87, 19, 54
 (f) Compare 87 and 19. Since 87 > 19, swapping is done.
     30, 29, 52, 63, 27, 19, 87, 54
 (g) Compare 87 and 54. Since 87 > 54, swapping is done.
     30, 29, 52, 63, 27, 19, 54, 87
```


Elementary Sorting Algorithms: Bubble Sort (cont.)

- For example, (cont.)

```
A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 2:
 (a) Compare 30 and 29. Since 30 > 29, swapping is done.
     29, 30, 52, 63, 27, 19, 54, 87
 (b) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (c) Compare 52 and 63. Since 52 < 63, no swapping is done.
 (d) Compare 63 and 27. Since 63 > 27, swapping is done.
     29, 30, 52, 27, 63, 19, 54, 87
 (e) Compare 63 and 19. Since 63 > 19, swapping is done.
     29, 30, 52, 27, 19, 63, 54, 87
 (f) Compare 63 and 54. Since 63 > 54, swapping is done.
     29, 30, 52, 27, 19, 54, 63, 87
```

- and so on

Elementary Sorting Algorithms: Bubble Sort (cont.)

- Complexity
  - In the first pass,
    - n – 1 comparisons
  - In the second pass,
    - n – 2 comparisons, and so on
  - (n – 1) + (n – 2) + … + 2 + 1
  - = n(n – 1)/2
  - O(n^2)