Sorting

Lecture 19

Instructor: Dr. Cong Pu, Ph.D.

cong.pu@okstate.edu

Adapted partially from Data Structures and Algorithms in Java, M.T. Goodrich, R.Tamassia and M. H. Goldwasser, Sixth Edition, Wiley; Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning

Introduction

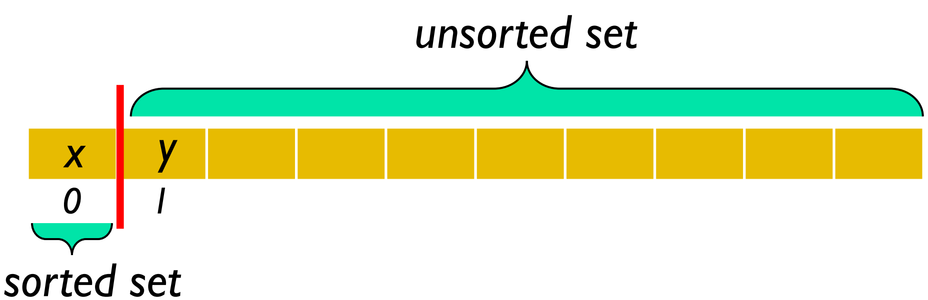
- ***Sorting***
  - improve the **efficiency of accessing (or handling) data**, e.g., increasing or decreasing order; or alphabetical order
    - e.g, find a name in the telephone directory
      - alphabetically ordered
  - *criteria* to order data: number or alphabetic character (ASCII)
  - common/critical *properties of souring algorithms* (machine-independent)
    - *number of comparisons* } *compared* and *moved*
    - *number of data movements* }
    - may be difficult to determine exactly → approximations
    - may differ depending on the original state of the data set (e.g., best case, worst case, and average case)

Elementary Sorting Algorithms: Insertion Sort

- ***Insertion sort***:

```
insertionsort(data[],n)
    for i = 1 to n-1    // unsorted set
        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



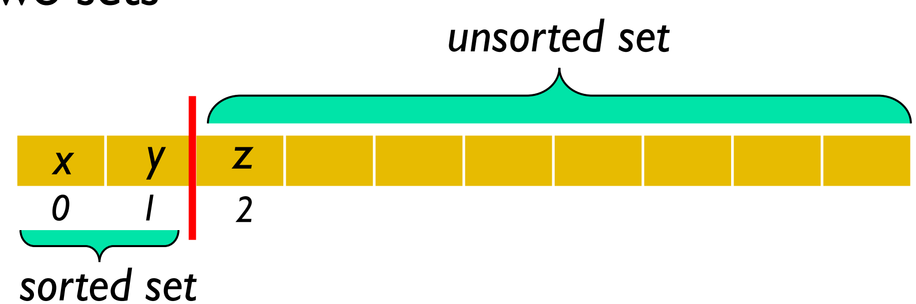
- Each repetition
  - pick up the ***first*** element in the ***unsorted set*** and insert it into the correct position of sorted set

Elementary Sorting Algorithms: Insertion Sort

- ***Insertion sort***:

```
insertionsort(data[],n)
    for i = 1 to n-1    // unsorted set
        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 of sorted set

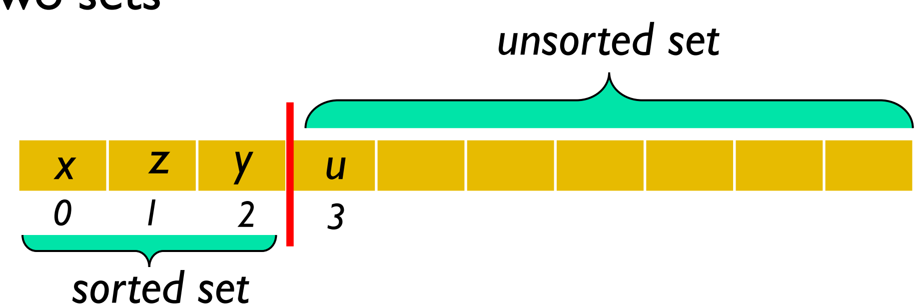


Elementary Sorting Algorithms: Insertion Sort

- ***Insertion sort***:

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

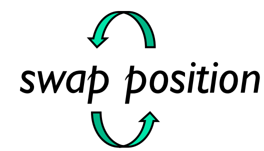
- The array of values → divide into two sets
  - sorted values vs. unsorted
- Initially, the element with **index 0**
  - sorted set
- The element with **index 1**
  - the first element of the unsorted set
- Each repetition
  - pick up the ***first*** element in the ***unsorted set*** and insert it into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

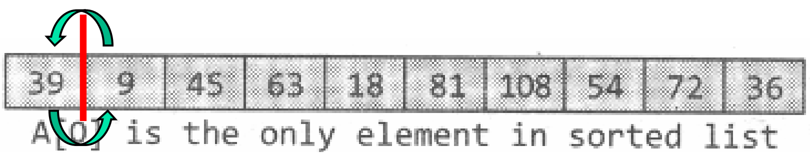


Elementary Sorting Algorithms: Insertion Sort (cont.)

- Consider an array of integers

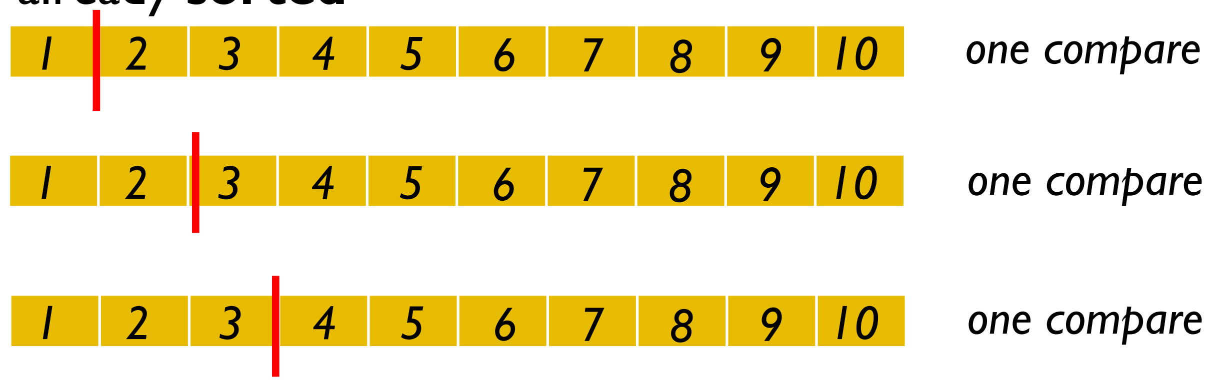
39	9	45	63	18	81	108	54	72	36





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)

Elementary Sorting Algorithms: Insertion Sort (cont.)

- The worst case,
  - the array is sorted in the **reverse order**

10	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*one compare*
9	*10*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*two compare*
8	*9*	*10*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*three compare*

  - the first element in the unsorted set compares with **almost every element** in the sorted set
  - O(n^2)

Elementary Sorting Algorithms: Selection Sort

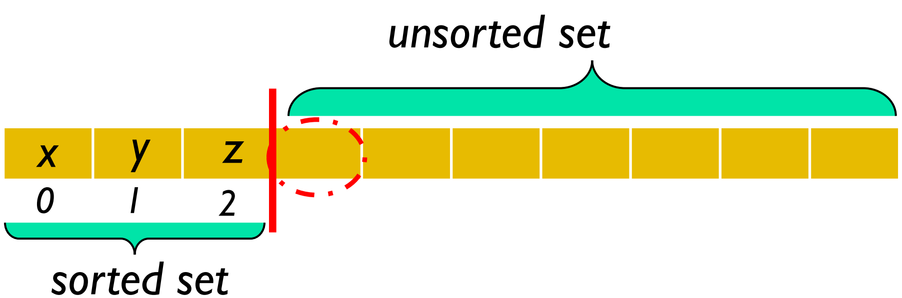
- ***Selection sort***
  - localize the exchange of array elements
    - finding a misplaced item and putting it in its final location
  - find the ***first* smallest value** in the array
    - place it in the *first* position
  - find the ***second* smallest value** in the array
    - place it in the *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 $n - 2$
  - if all items have been looked at and placed except for the last item
    - the last element has to be the largest

Elementary Sorting Algorithms: Selection Sort (cont.)

39	9	81	45	90	27	72	18

the location of smallest element in unsorted set

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]

Elementary Sorting Algorithms: Selection Sort (cont.)

- Complexity
  - pass 1:
    - select the element with the first smallest value for all n elements
    - **n – 1 comparisons**
  - pass 2
    - select the element with the second smallest value for all n - 1 elements
    - **n – 2 comparisons**
    - **(n – 1) + (n – 2) + … + 2 + 1**
    - n(n – 1) /2 = O(n^2)

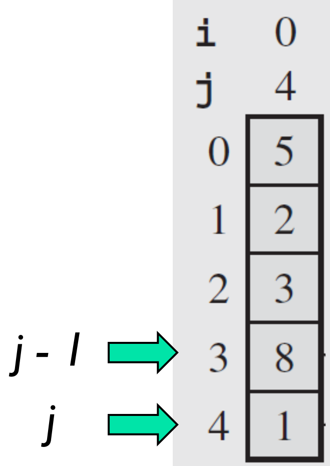
Elementary Sorting Algorithms: Bubble Sort

- ***Bubble sort***
  - during each pass, **compare** pairs of adjacent items and **swap** them if they are in the wrong order
  - repeatedly moving the **largest (smallest) element** to the **highest (lowest) 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 elements in positions j and j-1 if they are out of order;
```

Elementary Sorting Algorithms: Bubble Sort (cont.)

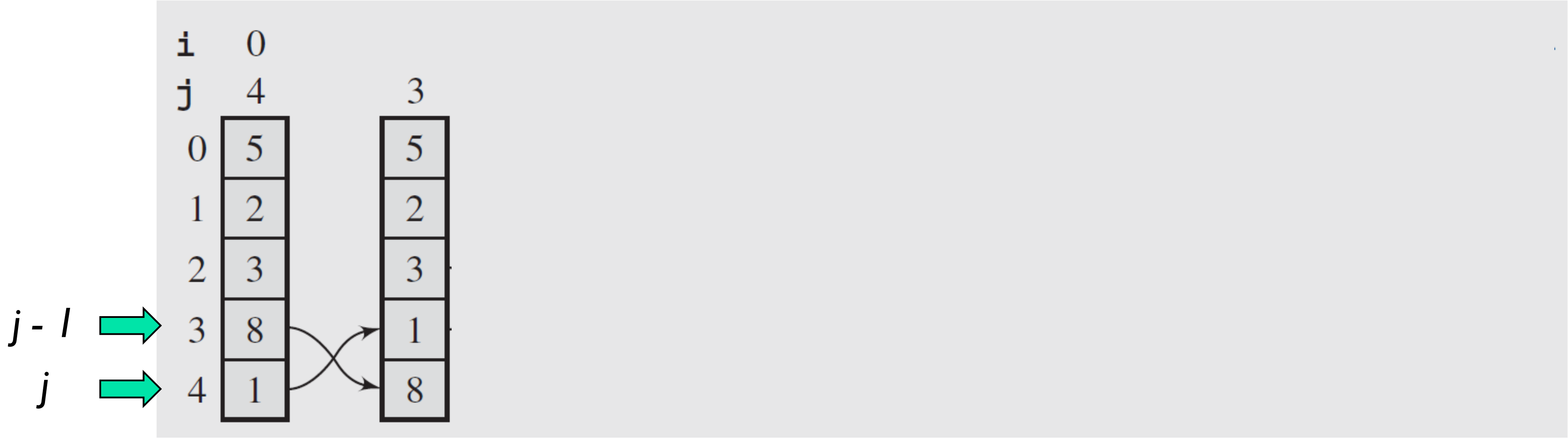
- For example,



Approach: repeatedly moving the smallest element to the lowest index position of the array.

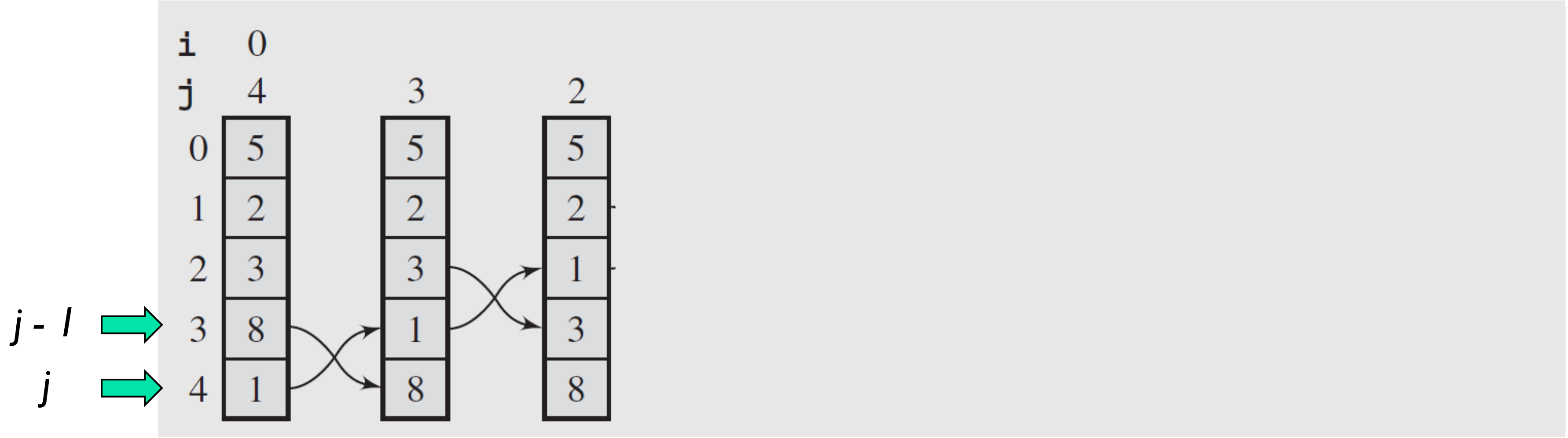
Elementary Sorting Algorithms: Bubble Sort (cont.)

- For example,



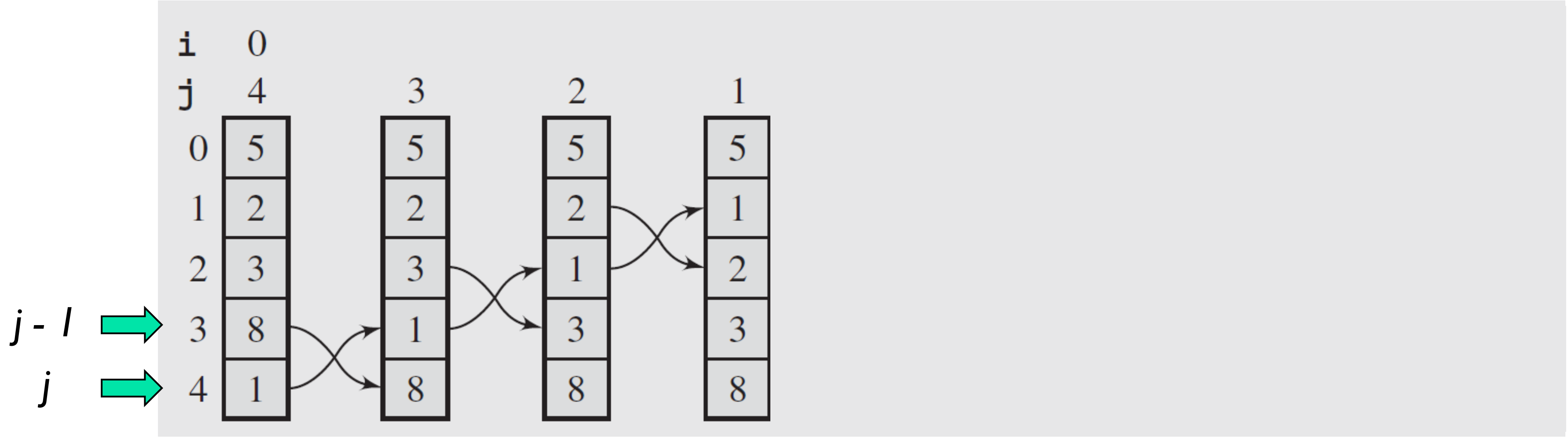
Elementary Sorting Algorithms: Bubble Sort (cont.)

- For example,



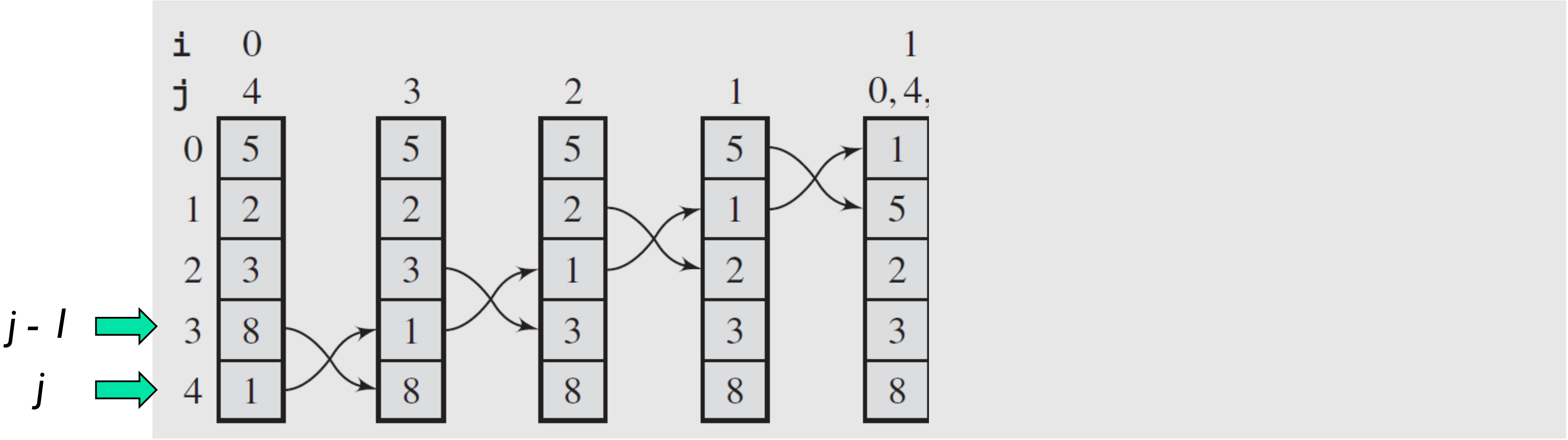
Elementary Sorting Algorithms: Bubble Sort (cont.)

- For example,



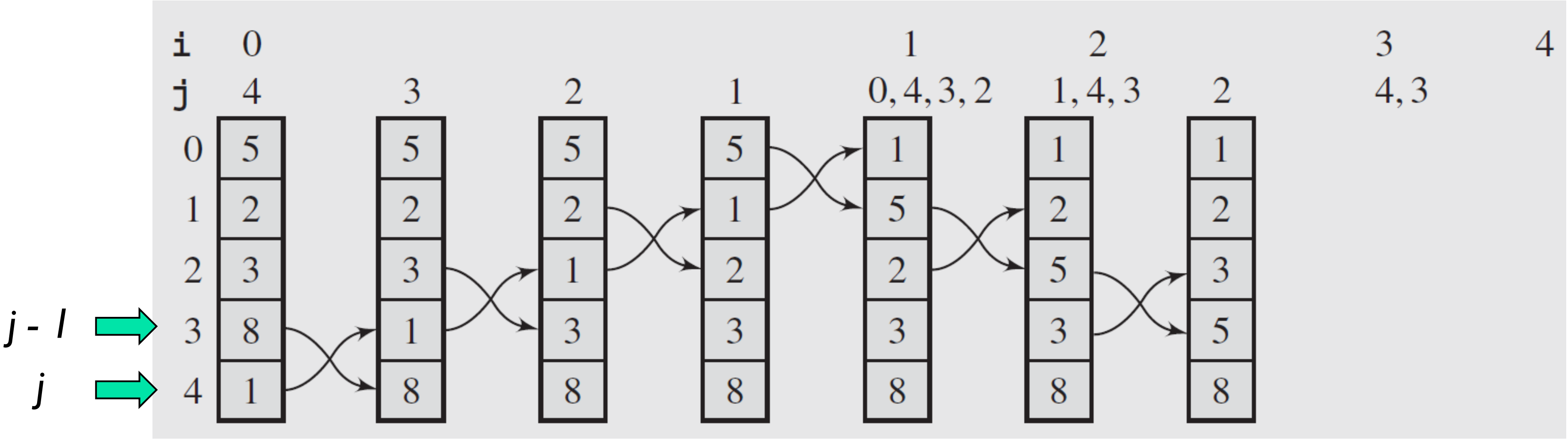
Elementary Sorting Algorithms: Bubble Sort (cont.)

- For example,



Elementary Sorting Algorithms: Bubble Sort (cont.)

- For example,

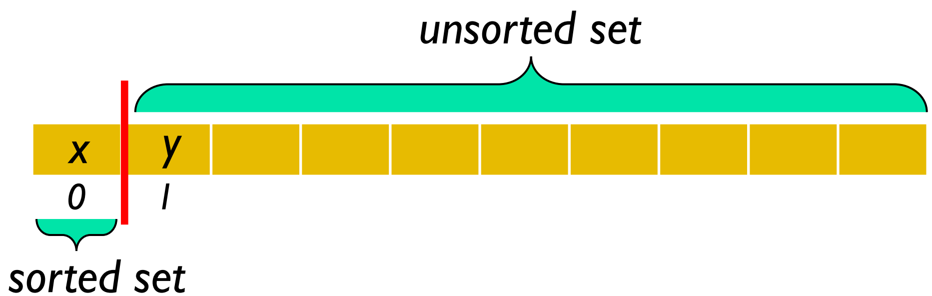


Elementary Sorting Algorithms: Bubble Sort (cont.)

- Complexity
  - in the first pass,
    - $n - 1$ comparisons
  - in the second pass,
    - $n - 2$ comparisons, and so on
  - $(n - 1) + (n - 2) + \ldots + 2 + 1 = n(n - 1)/2 = O(n^2)$

Efficient Sorting Algorithms: Shell Sort

- In insertion sort,
  - work well when the input element is "almost sorted"
  - if not sorted? inefficient to move the elements



Efficient Sorting Algorithms: Shell Sort

- ***Shell sort***
  - improve over *insertion sort*
    - number of data movements
  - how?
    - sort parts (partial array) of the original array first and then;
    - if they are at least partially ordered or already sorted;
    - getting closer to the best case of an ordered array than initially.

Efficient Sorting Algorithms: Shell Sort (cont.)

- The pseudo code of shell sort:

```
divide data into h subarrays;
for i = 1 to h
    sort subarray data_i;
sort array data;
```

one time or several times?

- if *h* is **too small**
  - the subarrays could be too large and the resulting sort would be inefficient
- if *h* is **too big**
  - too many subarrays
- use several different subdivisions
  - apply the same process separately to each subdivision

Efficient Sorting Algorithms: Shell Sort (cont.)

- The pseudo code of shell sort (cont.):

```
determine numbers h_t ... h_1 of ways of dividing array data into subarrays;
for (h=h_t; t > 1; t--, h=h_t)
    divide data into h subarrays;
    for i = 1 to h
        sort subarray data_i;
sort array data;
```

  - called,
    - diminishing increment sort, shell sort, or shell's method

Efficient Sorting Algorithms: Shell Sort (cont.)

- Perform the shell sort
  - arrange the elements in the form of a table
  - sort the columns
  - repeat with smaller number of long columns

63, 19, 7, 90, 81, 36, 54, 45, 72, 27, 22, 9, 41, 59, 33

h = 8

63	19	7	90	81	36	54	45
72	27	22	9	41	59	33	

Result:

63	19	7	9	41	36	33	45
72	27	22	90	81	59	54	

63, 19, 7, 9, 41, 36, 33, 45, 72, 27, 22, 90, 81, 59, 54

h = 5

63	19	7	9	41
36	33	45	72	27
22	90	81	59	54

Result:

22	19	7	9	27
36	33	45	59	41
63	90	81	72	54

22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54

Efficient Sorting Algorithms: Shell Sort (cont.)

- Perform the shell sort (cont.)
  - arrange the elements in the form of a table
  - sort the columns
  - repeat with smaller number of long columns

22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54

h = 3

22	19	7
9	27	36
33	45	59
41	63	90
81	72	54

Result:

9	19	7
22	27	36
33	45	54
41	63	59
81	72	90

9, 19, 7, 22, 27, 36, 33, 45, 54, 41, 63, 59, 81, 72, 90

Efficient Sorting Algorithms: Shell Sort (cont.)

- Perform the shell sort (cont.)
  - arrange the elements in the form of a table
  - sort the columns
  - repeat with smaller number of long columns

h = *1*	*Result:*
9	7
19	9
7	19
22	22
27	27
36	33
33	36
45	41
54	45
41	54
63	59
59	63
81	72
72	81
90	90

7, 9, 19, 22, 27, 33, 36, 41, 45, 54, 59, 63, 72, 81, 90