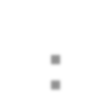
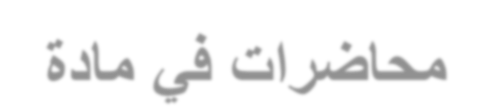
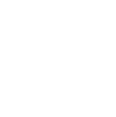
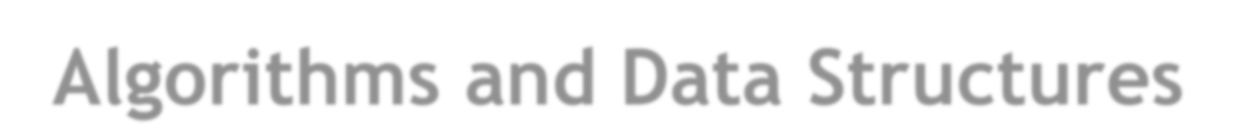
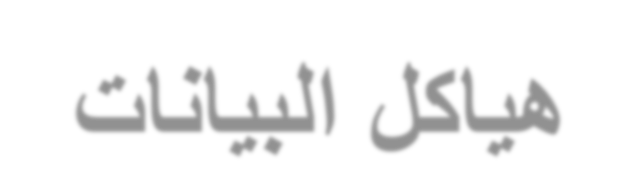
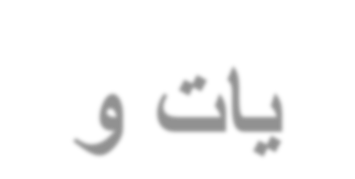
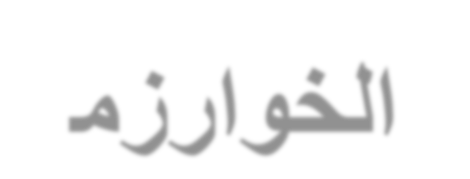
219



**Lecture # 10**

Sorting

Outline

• **Sorting:**

 Introduction.

• **Simple sorting algorithms:**

 Bubble Sort.

 Selection Sort.

 Insertion Sort.

Sorting

• **Sorting** is the process of rearranging a given set of objects in a specific order (ascending or descending).

• There are many commonly-used sorting algorithms.

 Some of them are good for sorting small sets of data, while others are good for sorting large sets of data.

 Some are simple, others are a bit more complicated.

Sorting (cont…)

• In this article, you'll learn about:

 Simple sorting algorithms which are good for sorting small sets of data (but slow) such as:

 **Bubble Sort.**

 **Selection Sort.**

 **Insertion Sort.**

 A bit more complicated sorting algorithms which are good for sorting large sets of data such as:

 **Merge Sort.**

 **Quick Sort.**

Sorting (cont…)

• The goal is to come up with better and more efficient sorts.

• Because sorting a large number of elements can be extremely time

consuming, a good sorting algorithm is very desirable.

• **Measuring performance** for different sort algorithms:

**1) Space/storage requirements:**

 Additional/temporary/intermediate arrays required?

**2) Algorithm running time:**

 Number of compares, moves, swaps of data items

required.

(1) Bubble Sort

• ***Bubble sort*** orders a list of values by repeatedly comparing neighboring (adjacent) elements and swapping their positions if necessary.

• **Bubble sort algorithm:**

 Scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the end.

 Scan the list again, bubbling up the second highest value.

 Repeat until all elements have been placed in their proper order.

"Bubbling" largest element

• Traverse a collection of elements:

 Move from the front to the end.

 "Bubble" the largest value to the end using pair-wise comparisons and swapping.

Bubble Sort Example 1

**Original:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **77** | **42** | **35** | **12** | **101** | **5** |

**1 2 3 4 5 6**

**1 2 3 4 5 6**

**Sw**

**77**

**ap42**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **42** | **77** | **35** | **12** | **101** | **5** |

Bubble Sort Example 1 (cont…)

**1 2 3 4 5 6**

**77 Sw**

**ap35**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **42** | **35** | **77** | **12** | **101** | **5** |

Bubble Sort Example 1 (cont…)

**1 2 3 4 5 6**

**77 Sw**

**ap12**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **42** | **35** | **12** | **77** | **101** | **5** |

Bubble Sort Example 1 (cont…)

**1 2 3 4 5 6**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **42** | **35** | **12** | **77** | **101** | **5** |

**No need to swap**

Bubble Sort Example 1 (cont…)

**1 2 3 4 5 6**

**101Sw**

**ap 5**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **42** | **35** | **12** | **77** | **5** | **101** |

Bubble Sort Example 1 (cont…)

**1 2 3 4 5 6**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **42** | **35** | **12** | **77** | **5** | **101** |

**Largest value correctly placed**

Bubble Sort Example 2

Original: 25 57 48 37 12 92 86 33

Pass 1: 25 57 48 37 12 92 86 33

25 48 57 37 12 92 86 33

25 48 37 57 12 92 86 33

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 25 | 48 | 37 | 12 | 57 | 92 86 33 |
| 25 | 48 | 37 | 12 | 57 | 86 92 33 |
| 25 | 48 | 37 | 12 | 57 | 86 33 92 |

Bubble Sort Example 2 (cont…)

Pass 2: 25 48 37 12 57 86 33 92

25 37 48 12 57 86 33 92

25 37 12 48 57 86 33 92

25 37 12 48 57 33 86 92

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Pass | 3: | 25 | 37 12 | 48 | 57 33 | 86 | 92 |
|  |  | 25 | 12 37 | 48 | 57 33 | 86 | 92 |
|  |  | 25 | 12 37 | 48 | 33 57 | 86 | 92 |

Bubble Sort Example 2 (cont…)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Pass | 4: | 25 12 | 37 48 33 | 57 | 86 | 92 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 12 | 25 | 37 48 33 | 57 | 86 | 92 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | 12 | 25 | 37 33 | 48 | 57 | 86 | 92 |
| Pass | 5: | 12 | 25 | 37 33 | 48 | 57 | 86 | 92 |
|  |  | 12 | 25 | 33 37 | 48 | 57 | 86 | 92 |
| Pass | 6: | 12 | 25 | 33 37 | 48 | 57 | 86 | 92 |

Bubble Sort code

**void swap(int &x, int &y)**

**{**

**int temp = x;**

**x = y;**

**y = temp;**

**}**

**void BubbleSort(int a[])**

**{**

**for (int i = 0; i < size; i++)**

**{**

**for (int j = 1; j < size - i; j++)**

**{**

**// swap adjacent out-of-order elements if (a[j-1] > a[j])**

**swap(a[j-1], a[j]);**

**}**

**}**

**}**

Bubble Sort runtime

• Running time is the number of comparisons for input size ***n***:

*n* 1

*n* *i*

*n* 1

 1 

 (*n*

 *i*)

*i* 0

*j* 1

*i* 0

*n* 1

  *i*

*i* 0

 (*n*  1)*n*

2

 O(*n* 2 )

 Number of actual swaps performed depends on the data.

 Out-of-order data performs many swaps.

Bubble Sort summary

• **Worst case:** Inverse sorting

 Passes: ***(n -* 1*)***

 Comparisons each pass: ***(n - k)*** , where ***k*** pass number.

 Total number of comparisons:

***(n -* 1*) + (n -* 2*) + (n -* 3*) + … +* 1 *= n2/2 - n/2***

***= O(n2)***

• **Best case:** already sorted

 **O(*n*)** : one pass over set, verifying sorting.

Bubble Sort summary (cont…)

• **Total number of exchanges:**

 **Best case:** None.

 **Worst case: *O(n2)***

 Lots of exchanges: a problem with large items.

(2) Selection Sort

• ***Selection sort*** orders a list of values by repeatedly putting a particular

value (smallest or largest) into its final position.

• **Selection sort algorithm:**

 Find the smallest value in the list.

 Switch it with the value in the first position.

 Find the next smallest value in the list.

 Switch it with the value in the second position.

 Repeat until all values are in their proper places.

Selection Sort Example 1

**3 9 6 1 2**

Scan right starting with 3. ..\_ ..\_

1 is the smallest. Exchange 1 and 3. T.\_

.T

Scan right starting with 9.

**1 9 6 3 2**

2 is the smallest. Exchange 9 and 2. **t t**

Scan right starting with 6.

**1 2 6** 3 **9**

3 is the smallest. Exchange 6 and 3. **t t**

Scan right starting with 6.

**1 2 3 6 9**

6 is the smallest. Exchange 6 and 6. **t**

**1 2 3 6 9**

Selection Sort Example 2

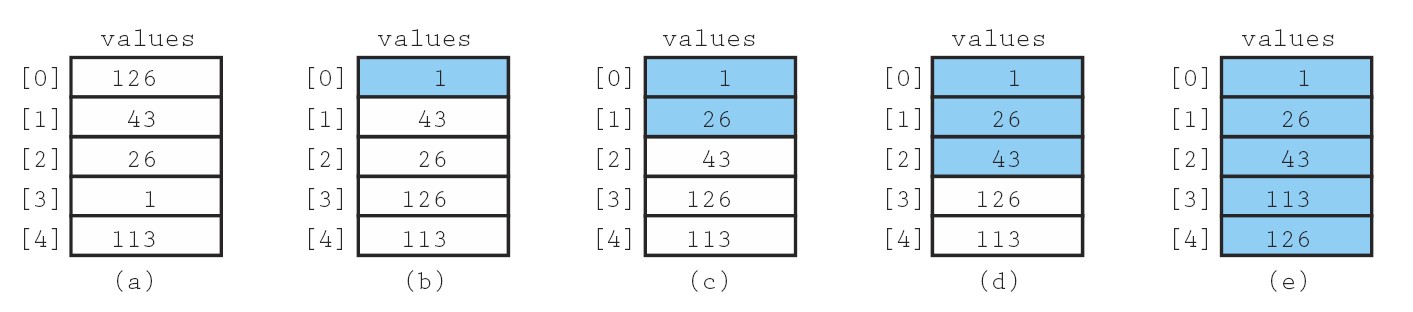


Figure 10.1: Example of Selection Sort (Sorted elements are shaded)

Selection Sort Example 3

 Find the **largest** value in the list, switch it with the value in the last position.

Initial: 25 57 48 37 12 92

**92**

86 33

Pass 1: 25 57 48 37 12 33 86

**86**

I 92

Pass 2: 25 57

**57**

48 37 12 33 I 86 92

Pass 3: 25 33 48

**48**

37 12 I 57 86 92

Pass 4: 25 33 12 37

**37**

I 48 57 86 92

Pass 5: 25 33

**33**

12 I 37 48 57 86 92

Pass 6:

**25**

25 12 I 33 37 48 57 86 92

Pass 7: 12 I 25 33 37 48 57 86 92

Selection Sort code

**void SelectionSort(int a[])**

**{**

**for (int i = 0; i < size; i++)**

**{**

**// find index of smallest element int min = i;**

**for (int j = i + 1; j < size; j++)**

**{**

**if (a[j] < a[min])**

**{**

**min = j;**

**}**

**}**

**// swap smallest element with a[i]**

**swap(a[i], a[min]);**

**}**

**}**

Selection Sort runtime

• Running time for input size ***n***:

 in practice, a bit faster than bubble sort.

*n* 1 *n*

*n* 1

 1 

 (*n*

 (*i*

 1)  1)

*i* 0

*j* *i* 1

*i* 0

*n* 1

  (*n*

*i* 0

*n* 1

  *i*

*i* 0

 *i*)

 (*n*  1)*n*

2

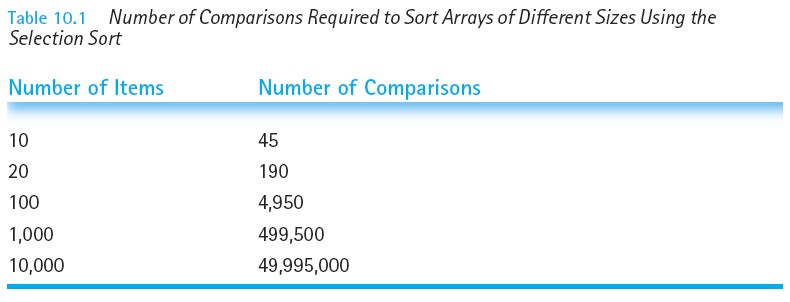
 O(*n* 2 )

Selection Sort runtime (cont…)

• Note that the particular arrangement of values in the array does not affect the amount of work done.

• Even if the array is in sorted order before the call to **SelectionSort**, the

function still makes ***n(n-1)/2*** comparisons.



Selection Sort summary

• **Best case:** Already sorted

 Passes: ***(n -* 1*)***

 Comparisons each pass: ***(n-k)*** where ***k*** pass number.

 Number of comparisons:

***(n -* 1*) + (n -* 2*) + (n -* 3*) + … +* 1 *= n2/2 - n/2***

***= O(n2)***

• **Worst case:** Same.

• **Number of exchanges:**

 Always ***(n -* 1*)*** (better than Bubble Sort).

(3) Insertion Sort

• ***Insertion sort*** orders a list of values by repeatedly inserting a particular

value into a sorted subset of the list.

• **Selection sort algorithm:**

 Consider the first item to be a sorted sublist of length 1.

 Insert the second item into the sorted sublist, shifting the first item if needed.

 Insert the third item into the sorted sublist, shifting the other items as needed.

 Repeat until all values have been inserted into their proper positions.

Insertion Sort (cont…)

• Simple sorting algorithm.

 ***(n -* 1*)*** passes over the array.

 At the end of **pass *i***, the elements that occupied **A[0] … A[*i*]**

originally are still in those spots and in sorted order.

Insertion Sort Example 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 15 | 8 | 1 | 17 | 10 | 12 | 5 |

**Original:**

0 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 15 | 8 | 1 | 17 | 10 | 12 | 5 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 15 | 1 | 17 | 10 | 12 | 5 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 15 | 8 | 1 | 17 | 10 | 12 | 5 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Pass 1:** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Pass 2:** |  |  |  |  |  |  |  |  |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Insertion Sort Example 1 (cont…)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 8 | 15 | 17 | 10 | 12 | 5 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 8 | 15 | 17 | 10 | 12 | 5 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 8 | 10 | 15 | 17 | 12 | 5 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Pass 3:** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Pass 4:** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Pass 5:** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Insertion Sort Example 1 (cont…)

**Pass 6:**

0 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 8 | 10 | 12 | 15 | 17 | 5 |

**Pass 7:**

0 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 5 | 8 | 10 | 12 | 15 | 17 |

Insertion Sort Example 2

3 is sorted.

Shift nothing. Insert 9.

**3 9 6 1** 2

3 and 9 are sorted.

**3 9 ----11•• 6 1 2**

Shift 9 to the right. Insert 6. **t**

**3 ----ti·· 6 ----ti·· 9 ----11•• 1** 2



+.... \_

3, 6, and 9 are sorted.

Shift 9, 6, and 3 to the right. Insert 1.

1, 3, 6, and 9 are sorted.

**1 3 ----ti·· 6 ----ti·· 9 ----11•• 2**

Shift 9, 6, and 3 to the right. Insert 2. **t**

**1 2 3 6 9**

Insertion Sort Example 3

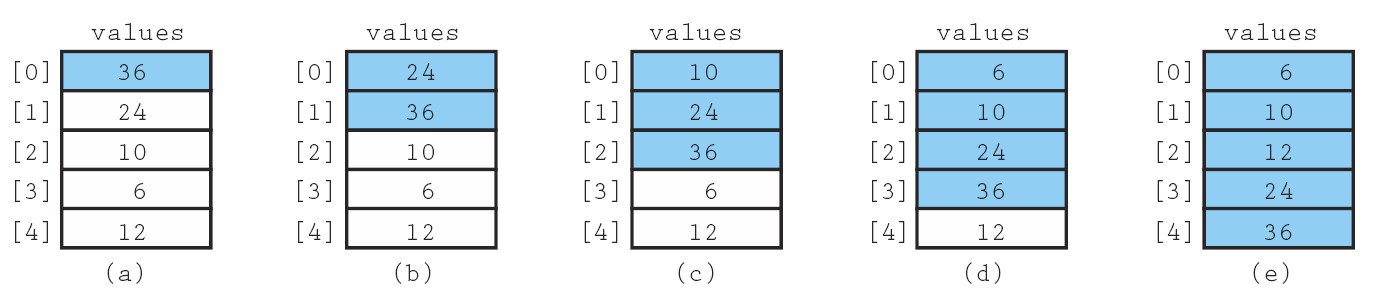


Figure 10.3: Example of Insertion Sort (Sorted elements are shaded)

Insertion Sort code

**void InsertionSort(int a[])**

**{**

**for (int i = 1; i < size; i++)**

**{**

**int temp = a[i];**

**// slide elements down to make room for a[i]**

**int j = i;**

**while (j > 0 && a[j - 1] > temp) {**

**a[j] = a[j - 1];**

**j--;**

**}**

**a[j] = temp;**

**}**

**}**

Insertion Sort runtime

• **Worst case:** reverse-ordered elements in array.

*n* 1

 *i*

*i* 1

 1  2  3  ...  (*n*  1) 

(*n*  1)*n*

2

 *O*(*n*2 )

• **Best case:** array is in sorted ascending order.

*n* 1

1 

*i* 1

*n*  1  *O*(*n*)

• **Average case:** each element is about half way in order.

*n* 1

*i*



 1 (1  2  3...  (*n* 1)) 

(*n*  1)*n*

*i* 1

2 2 4

 *O*(*n*2 )

Insertion Sort summary

• **Best case:** Already sorted  ***O(n)*** comparisons.

• **Worst case: *O(n2)*** comparisons.

• **Number of exchanges:**

 **Best case: *O(n)***.

 **Worst case: *O(n2)***.

• In practice, best for small sets (<30 items).

• Very efficient on nearly-sorted inputs.

Comparing Simple Sorts

• We've seen "simple" sorting algorithms, such as:

 Bubble sort, Selection sort, and Insertion sort.

|  |  |  |  |
| --- | --- | --- | --- |
|  | | **Comparisons** | **Swaps** |
| **Bubble** | Worst | n(n-1)/2 | n(n-1)/2 |
| Best | n-1 | n-1 |
| **Selection** |  | n(n-1)/2 | n-1 |
| **Insertion** | Worst | n(n-1)/2 | n(n-1)/2 |
| Best | n-1 | n-1 |

• They all use nested loops and perform approximately ***n2*** comparisons.

• They are relatively inefficient.