

Data Structures and Algorithms

Chapter 5 Sorting

Simple Sort

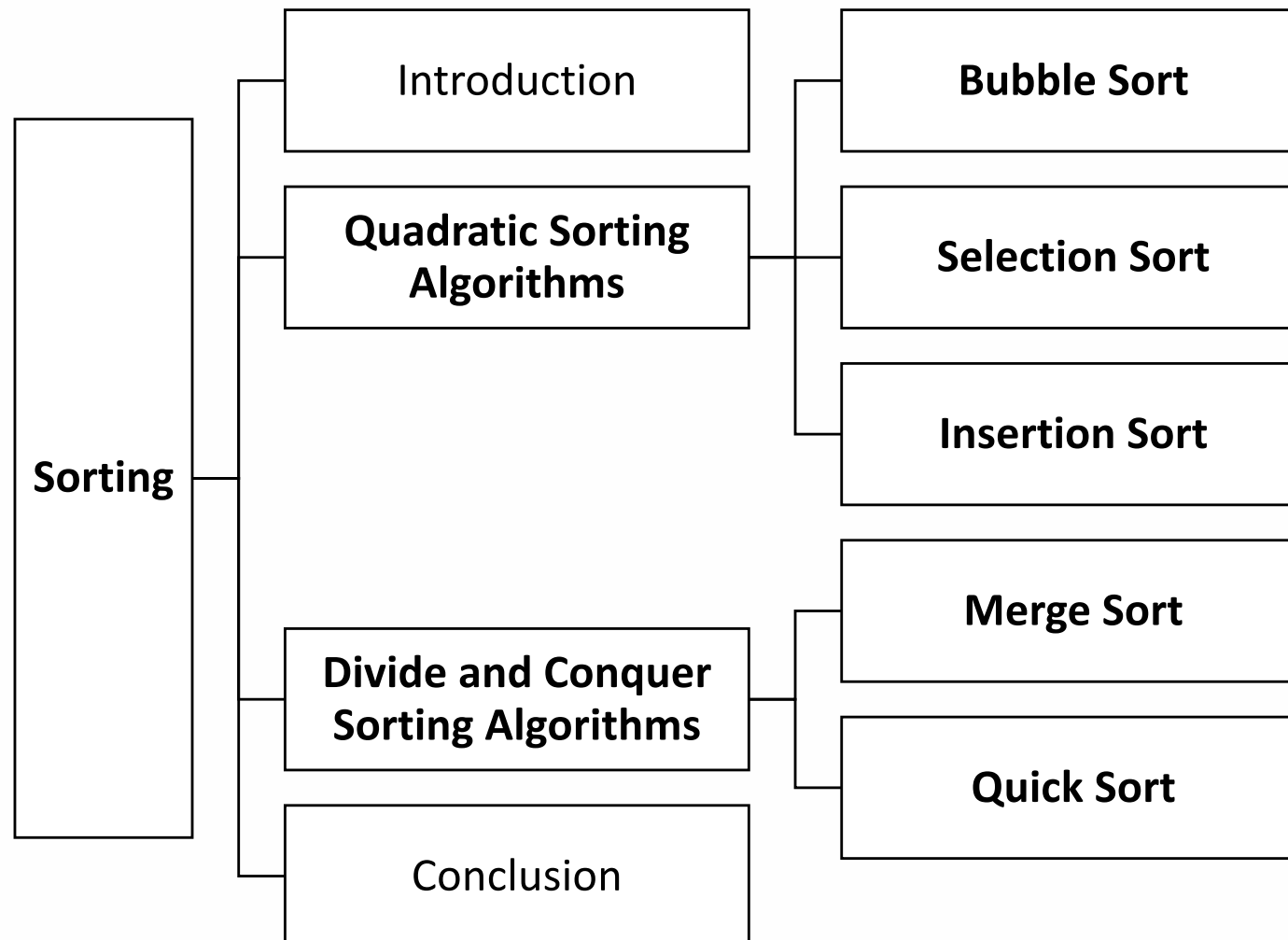
Nor Bahiah Hj Ahmad & Dayang Norhayati A.Jawawi
School of Computing

Objectives

At the end of the class, students are expected to be able to do the following:

- Understand the purpose of sorting technique as operations on data structure.
- Use simple sorting techniques in problem solving: Bubble sort, Insertion sort and Selection sort.
- Use divide and conquer sorting techniques in problem solving: Merge sort and Quick sort.
- Able to analyze the efficiency of the sorting technique.

Class Content



Introduction: **Sorting**

Definition:

- A process in which data in a list are organized in certain order; either ascending or descending order.

Examples:

- Sorted in **Ascending Order**: phone directory and dictionary
- Sorted in **Descending Order**: number of scores / points earned by every team in a competition. The winner get the highest score.

Advantages of Sorted Lists:

- Easier to understand and analyze data collection.
- Searching process will be much faster.

Introduction: **Sorting Algorithms Categories**

- **Internal Sort**

- Requires that the collection of data fit entirely in the computer's main memory.
- Suitable to sort a **small size** of list.

- **External Sort**

- The collection of data will not fit in the computer's main memory all at once, but it must reside (locate) in secondary storage.
- Suitable to sort **large size** of data.

Introduction: Types Of Lists To Be Sorted

- **List of simple data types**, such as **integers**, **char** or **strings**
 - Examples:
 - list of numbers (**int** type), or list of book titles (**string** type)

Student's Marks
65
72
100
92
98

Unsorted



Student's Marks
100
98
92
72
65

**Sorted in
descending order**

List of Books
Data Structure
Learning English
Math for Kids
Effective Communication
Learn C++

Unsorted



List of Books
Data Structure
Effective Communication
Learn C++
Learning English
Math for Kids

**Sorted in
ascending order**

Introduction: **Types of Lists to be Sorted** (continued...)

- **List of Records** – the list contain more than one element or record.
 - Examples:
 - A list that contains student's information.
 - Each record contains several field and a field called the record key.
 - Record key – field that become the identifier to the record.

List of Records		Student Name	Matric Number	CPA	} Fields
	[0]	Hisham	A5021	3.09	
	[1]	Zainal	A1051	2.55	
	[2]	Maria	A2000	3.60	
	[3]	Adam	A5501	3.00	
	[4]	Zahid	A2233	2.95	

Introduction: **Sorting List of Records**

For sorting purposes, the records will be sorted based on the sorting key, which is part of data item that we consider when sort a data collection.

	Student Name	Matric Number	CPA
[0]	Maria	A2000	3.60
[1]	Hisham	A5021	3.09
[2]	Adam	A5501	3.00
[3]	Zahid	A2233	2.95
[4]	Zainal	A1051	2.55

Sorting Key

The list can be sorted either by student's name, matric number, or CPA.

The list above is sorted into descending order based on CPA value.

From the sorted list, we can easily retrieve the highest score and the lowest score for the exam.

Sorting Process

Main activity in the sorting process:

- **Compare:** compare between two elements. If they are not in correct order, then
- **Swap:** Change the position of the elements in order to get the right order.

The **efficiency** of sorting algorithm is measured based on :

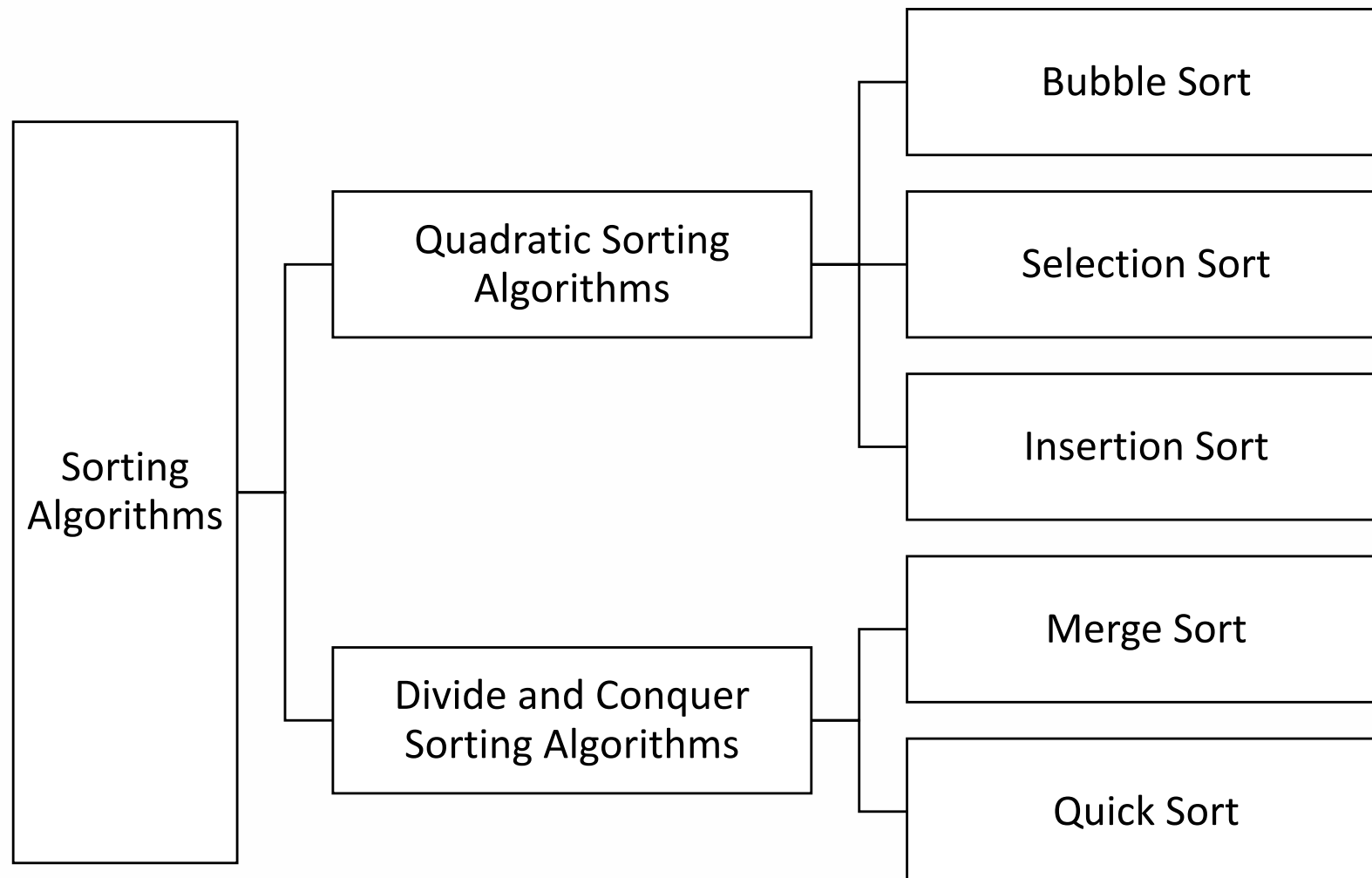
- Number of **comparisons**
- Number of **swapping** between elements

Sorting Process (continued...)

The **sorting efficiency** is measured based on the execution time of the algorithm when tested using sample cases of data as follows:

- **Worst-case analysis** considers the **maximum** amount of work an algorithm will require on a problem of a given size. (Data is totally **unsorted**)
- **Best-case analysis** considers the **minimum** amount of work an algorithm will require on a problem of a given size. (Data is **almost sorted**)
- **Average-case analysis** considers the **expected** amount of work that an algorithm will require on a problem of a given size.

Sorting Algorithms



Bubble Sort

6 5 3 1 8 7 2 4

Bubble Sort

Main Activities:

- Take multiple passes over the array
 - **Compare** adjacent elements in the list
 - **Exchange** the elements if they are out of order
 - Each pass **moves the largest (or smallest)** elements to the end of the array
 - **Repeating** this process eventually sorts the array into **ascending (or descending)** order.
- Bubble sort is a quadratic algorithm **$O(n^2)$** , which is only suitable to sort array with **small size of data**.

Bubble Sort (continued...)

Pass 1: Unsorted List

- Compare, swap (0,1)
- Compare, swap (1,2)
- Compare (2,3), no swap
- Compare (3,4), no swap
- Compare, swap (4,5)
- **99** is in right position



Bubble Sort (continued...)

Pass 2

- Compare, swap (0, 1)
- no swap (1,2)
- no swap (2,3)
- Compare, swap (3, 4)
- **21** is in the right position



Bubble Sort (continued...)

Pass 3

- no swap (0,1)
- no swap (1,2)
- Compare, swap (2, 3)
- **12** in right position

3	8	12	1	21	99
3	8	12	1	21	99
3	8	12	1	21	99
3	8	1	12	21	99
3	8	1	12	21	99

Pass 4

- no swap (0,1)
- Compare, swap (1, 2)
- **8** in right position

3	8	1	12	21	99
3	1	8	12	21	99
3	1	8	12	21	99

Pass 5

- Compare, swap (0, 1)
- Element **1** and **3** in right position - **Done**

3	1	8	12	21	99
1	3	8	12	21	99
1	3	8	12	21	99

Bubble Sort Implementation

// Sorts items in an array into ascending order.

void BubbleSort (dataType data[], int listSize)

{ int pass, tempValue;

for (pass =1; pass < listSize; pass++)

 { // moves the largest element to the

 // end of the array

for (int x = 0; x < listSize - pass; x++)

 { // compare adjacent elements

if (data[x] > data[x+1])

 { // swap elements

 tempValue = data[x];

 data[x] = data[x+1];

 data[x+1] = tempValue;

 }

 }

 }

} // end Bubble Sort

} **External for loop** is used to control the number of passes needed.

} **Internal for loop** is used to control swap if they are not in order. After the internal loop has finished execution, the largest element in the array will be moved at the top.
if statement is used to compare the adjacent elements.

Example of Bubble Sort Implementation : Sort [7 8 3 1 6] into Ascending Order

// Sorts items in an array into ascending order.

void BubbleSort (dataType data[], int listSize)

{ int pass, tempValue;

for (pass =1; pass < listSize; pass++)

 { // moves the largest element to the

 // end of the array

for (int x = 0; x < listSize - pass; x++)

 { // compare adjacent elements

if (data[x] > data[x+1])

 { // swap elements

 tempValue = data[x];

 data[x] = data[x+1];

 data[x+1] = tempValue;

 }

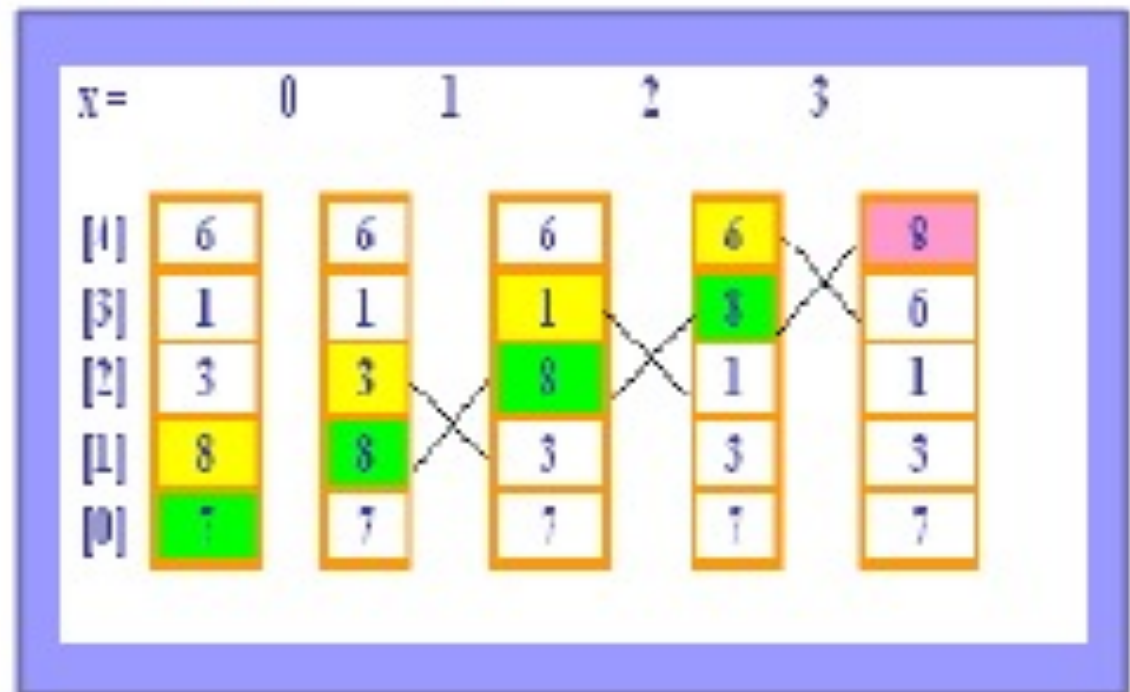
 }

 }

} // end Bubble Sort

pass = 1

List Size = 5



Example of Bubble Sort Implementation : Sort [7 8 3 1 6] into Ascending Order (continued...)

// Sorts items in an array into ascending order.

void BubbleSort (dataType data[], int listSize)

{ int pass, tempValue;

for (pass =1; pass < listSize; pass++)

 { // moves the largest element to the

 // end of the array

for (int x = 0; x < listSize - pass; x++)

 { // compare adjacent elements

if (data[x] > data[x+1])

 { // swap elements

 tempValue = data[x];

 data[x] = data[x+1];

 data[x+1] = tempValue;

 }

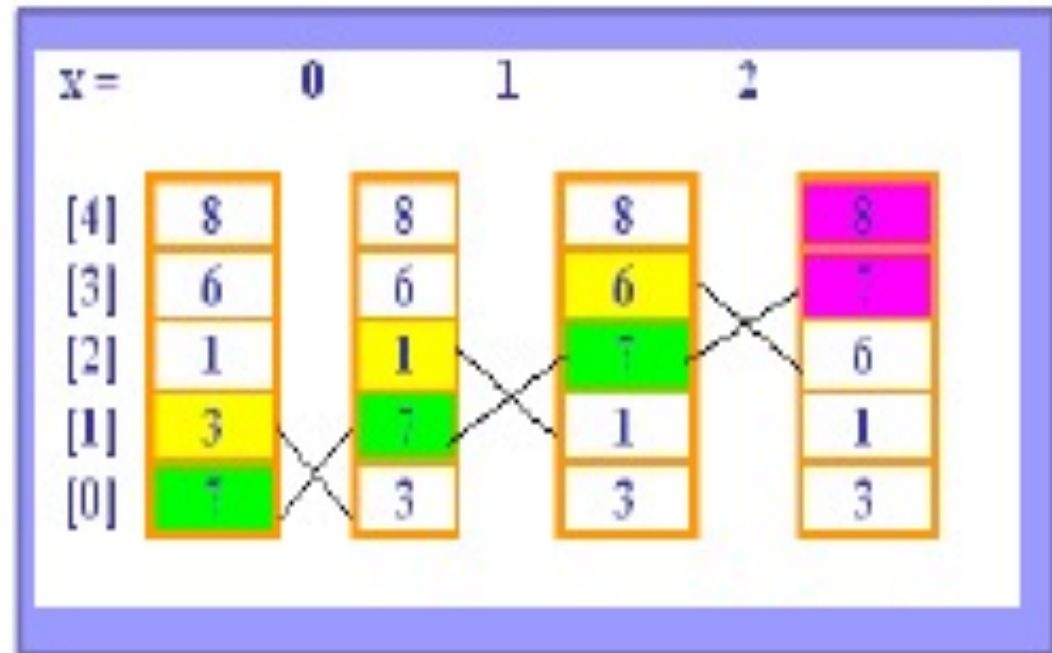
 }

 }

} // end Bubble Sort

pass = 2

listSize = 5



Example of Bubble Sort Implementation : Sort [7 8 3 1 6] into Ascending Order (continued...)

// Sorts items in an array into ascending order.

void BubbleSort (dataType data[], int listSize)

{ int pass, tempValue;

for (pass =1; pass < listSize; pass++)

 { // moves the largest element to the

 // end of the array

for (int x = 0; x < listSize - pass; x++)

 { // compare adjacent elements

if (data[x] > data[x+1])

 { // swap elements

 tempValue = data[x];

 data[x] = data[x+1];

 data[x+1] = tempValue;

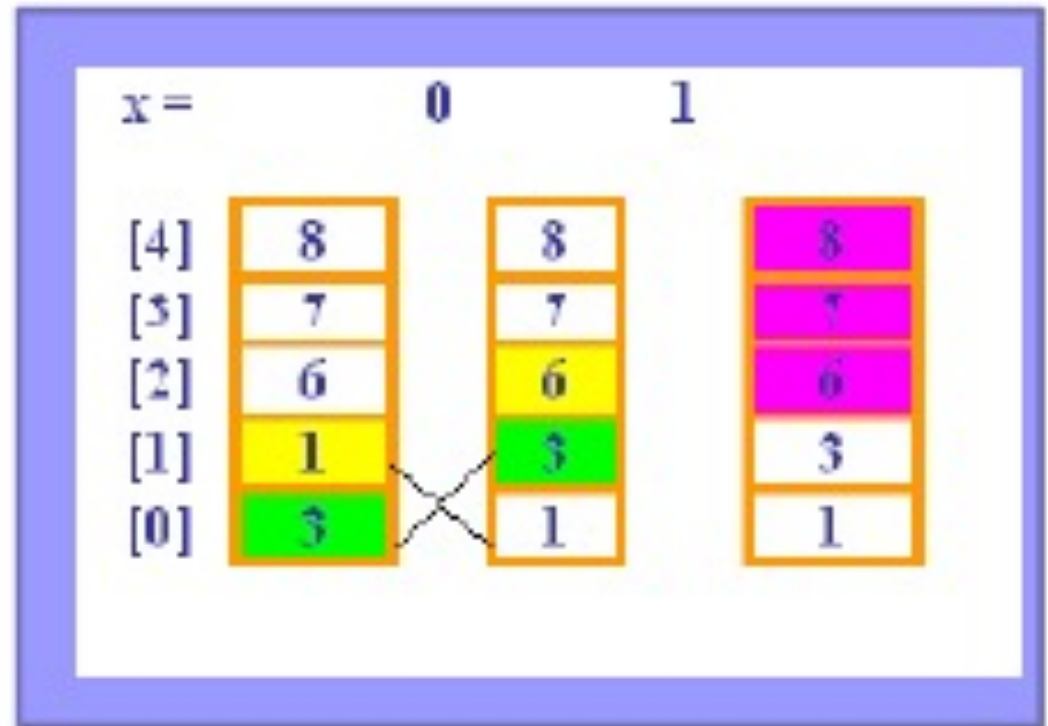
 }

 }

 }

} // end Bubble Sort

pass= 3



Example of Bubble Sort Implementation : Sort [7 8 3 1 6] into Ascending Order (continued...)

// Sorts items in an array into ascending order.

void BubbleSort (dataType data[], int listSize)

{ int pass, tempValue;

for (pass =1; pass < listSize; pass++)

 { // moves the largest element to the

 // end of the array

for (int x = 0; x < listSize - pass; x++)

 { // compare adjacent elements

if (data[x] > data[x+1])

 { // swap elements

 tempValue = data[x];

 data[x] = data[x+1];

 data[x+1] = tempValue;

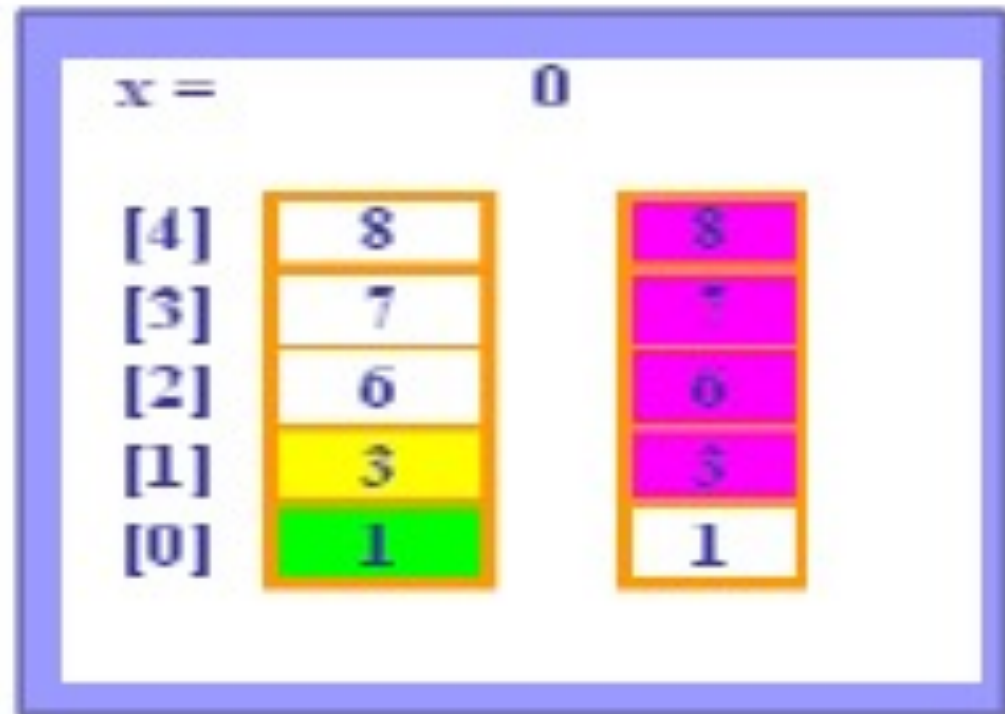
 }

 }

 }

} // end Bubble Sort

pass = 4



Bubble Sort Analysis

- The **number of comparison** between elements and the number of **exchange** between elements determine the **efficiency** of Bubble Sort algorithm.
- Generally, the **number of comparisons between elements** in Bubble Sort can be stated as follows:

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

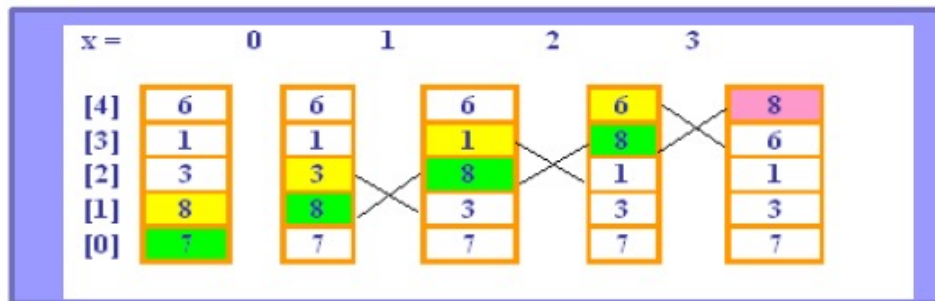
- However, in any cases, (**worse case, best case or average case**) the **number of comparisons** between elements is the **same**.

Bubble Sort Analysis [7 8 3 1 6]

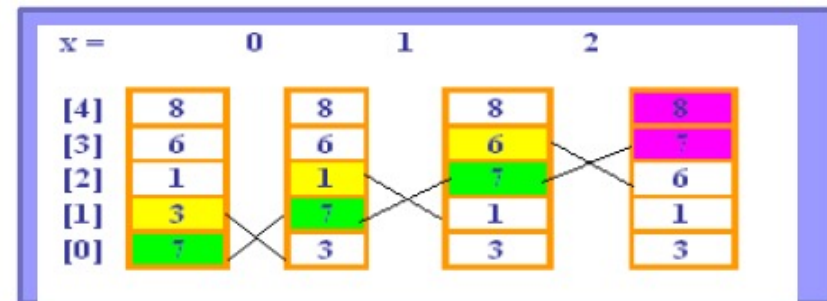
$$(n-1) + (n-2) + \dots + 2 + 1 = n(n-1) / 2 = O(n^2)$$

The number of comparisons:

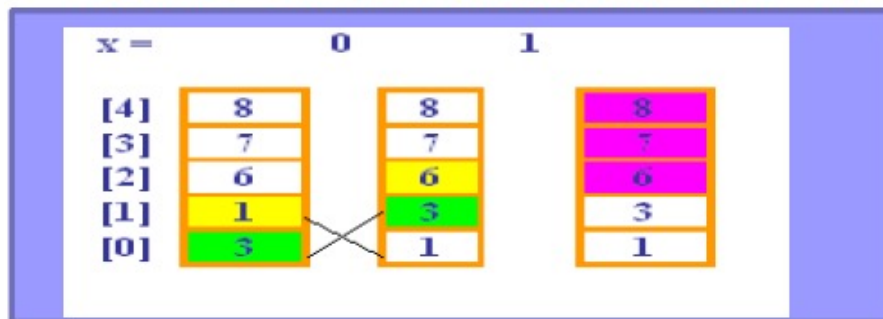
$$(5-1) + (5-2) + (5-3) + (5-4) = 4 + 3 + 2 + 1 = 10$$



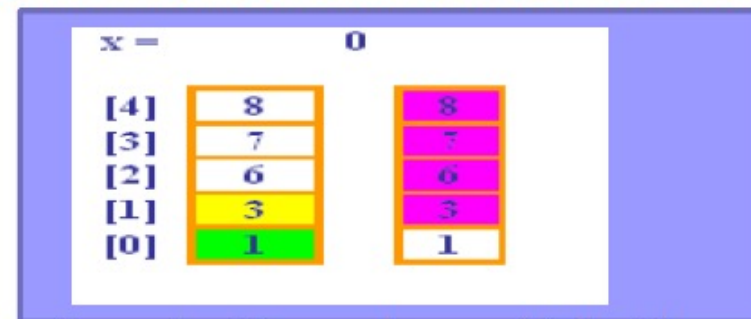
Pass 1 : Comparison (listSize-pass=4)



Pass 2 : Comparisons (listSize-pass:(5-2=3))



Pass 3 : Comparison (5-3= 2)



Pass 4 : Comparisons (5-4=1)

Bubble Sort Analysis (continued...)

In any cases, (worse case, best case or average case) to sort the list in ascending order the number of comparisons between elements is the same.

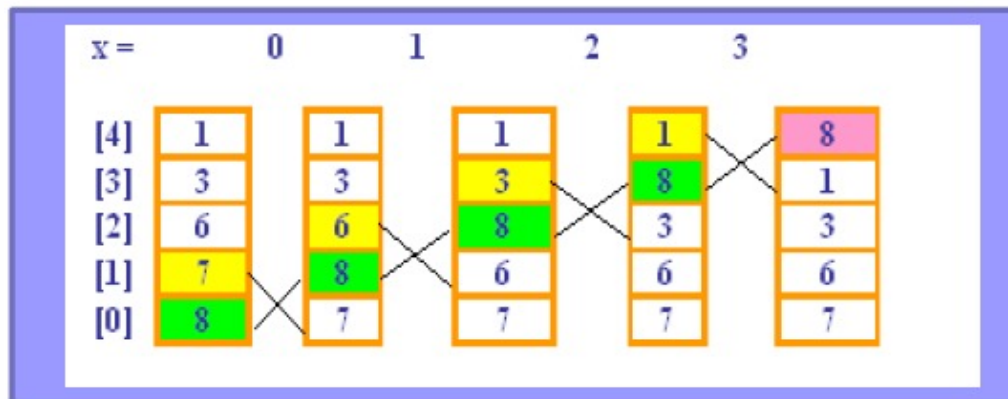
- Worse Case [8 7 6 3 1]
- Average Case [7 8 3 1 6]
- Best Case [1 3 6 7 8]

All lists with 5 elements need 10 comparisons to sort all the data.

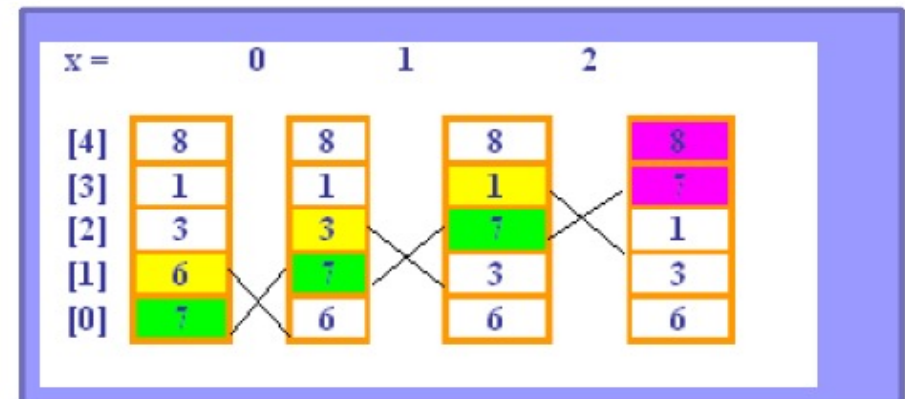
Worse Case Analysis [8 7 6 3 1]

The number of comparisons to sort data in this list:

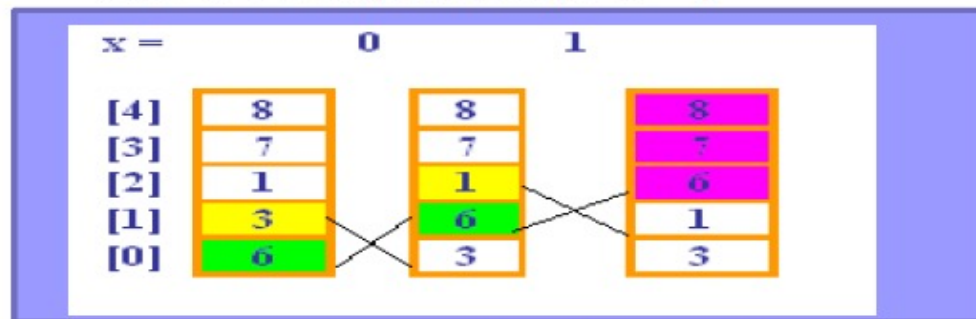
$$(5-1) + (5-2) + (5-3) + (5-4) = 4 + 3 + 2 + 1 = 10$$



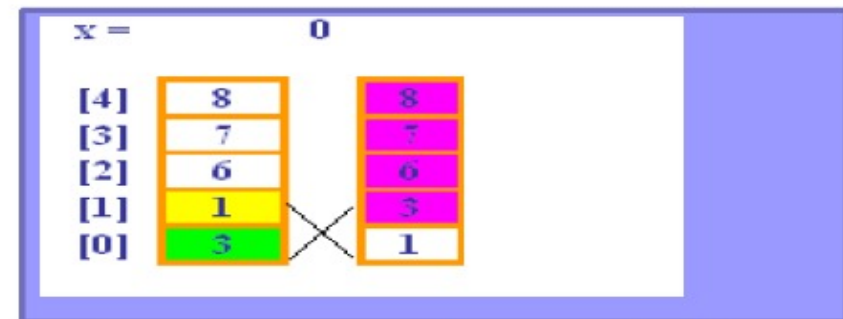
Pass 1 : Comparison (5-1=4)



Pass 2 : Comparisons (5-2=3)



Pass 3 : Comparison (5-3= 2)

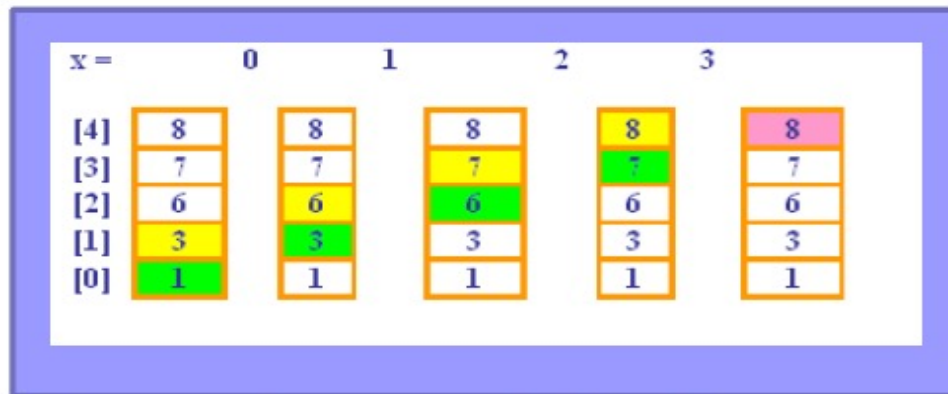


Pass 4 : Comparisons (5-4=1)

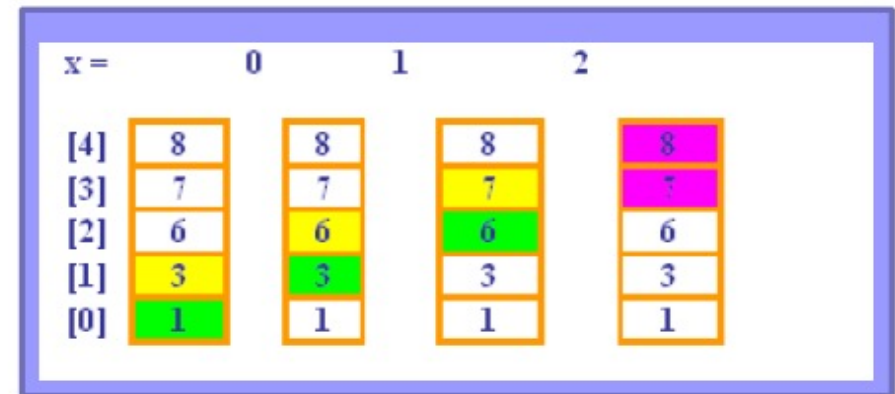
Best Case Analysis [1 3 6 7 8]

The number of comparisons to sort data in this list:

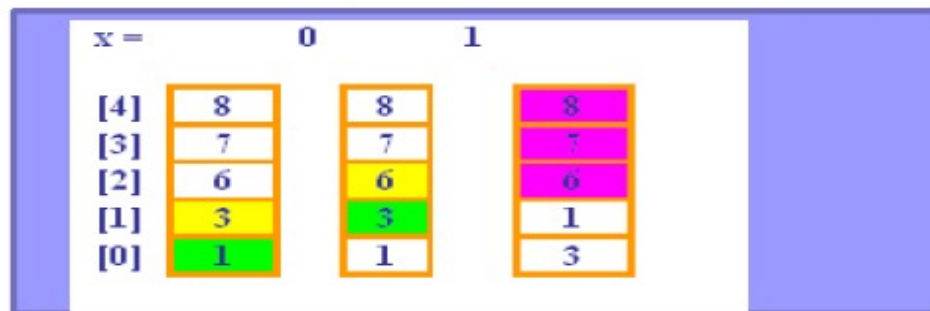
$$(5-1) + (5-2) + (5-3) + (5-4) = 4 + 3 + 2 + 1 = 10$$



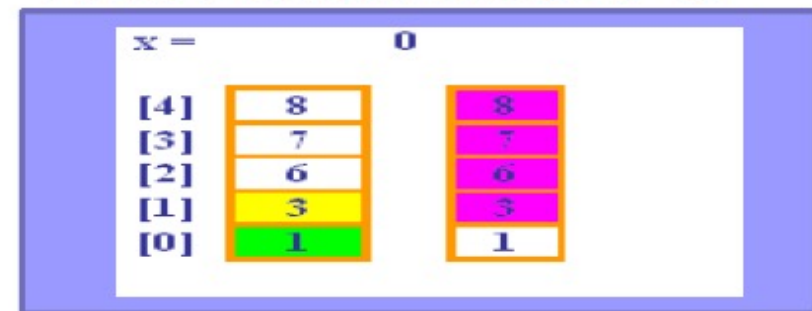
Pass 1 : Comparison (5-1=4)



Pass 2 : Comparisons (5-2=3)



Pass 3 : Comparison (5-3= 2)



Pass 4 : Comparisons (5-4=1)

Bubble Sort Analysis (continued...)

- In the example given, it shows that the number of comparison for the worse case and best case is the same - with ten comparisons.
- The **difference** can be seen in the **number of swapping elements**. The **worse case** has a **maximum number of swapping**: 10, while the **best case** has **no swapping** since all data is already in the **right position**.
- For best case, we can observe that starting with pass one, there is no exchange of data occur.
- From the example, it can be concluded that in any pass, if there is **no exchange of data** occurs, the list is already **sorted**. The next pass shouldn't be continued, and the sorting process should stop.

Improved Bubble Sort

// Sorts items in an array into ascending order.

void bubbleSort (DataType data[], int n)

```
{   int temp;

    bool sorted = false; // false when swaps occur
    for (int pass = 1; (pass < n) && !sorted; ++pass)
    {   sorted = true; // assume sorted
        for (int x = 0; x < n-pass; ++x)
        {   if (data[x] > data[x+1])
            {   // exchange items
                temp = data[x];
                data[x] = data[x+1];
                data[x+1] = temp;
                sorted = false; // signal exchange
            } // end if
        } // end for
    } // end for
}
```

To **improve the efficiency** of Bubble Sort, a condition that check whether the list is sorted should be add at the external loop.

A Boolean variable, **sorted** is added in the algorithm to signal whether there is any exchange of elements occur in certain pass.

In external loop, **sorted** is set to **true**. If there is exchange of data inside the inner loop, sorted is set to **false**.

Another pass will continue, if sorted is false and will stop if sorted is true.

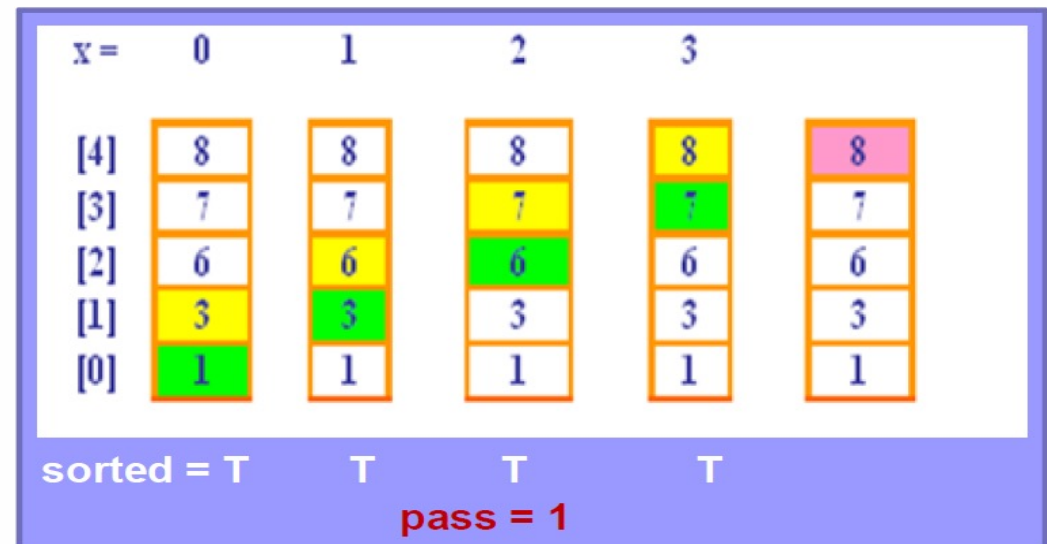
Improved Bubble Sort : Best Case [1 3 6 7 8]

// Sorts items in an array into ascending order.

```
void bubbleSort (DataType data[], int n)
```

```
{
  int temp;
  bool sorted = false; // false when swaps occur
  for (int pass = 1; (pass < n) && !sorted; ++pass)
  {
    sorted = true; // assume sorted
    for (int x = 0; x < n-pass; ++x)
    {
      if (data[x] > data[x+1])
      {
        // exchange items
        temp = data[x];
        data[x] = data[x+1];
        data[x+1] = temp;
        sorted = false; // signal exchange
      }
    }
  }
}
```

In **Pass 1**, there is **no exchange** of data occur and variable sorted is always **True**. Therefore, condition statement in **external loop** will become **false** and the loop will **stop** execution. In this example, **Pass 2** will **not be continued**.



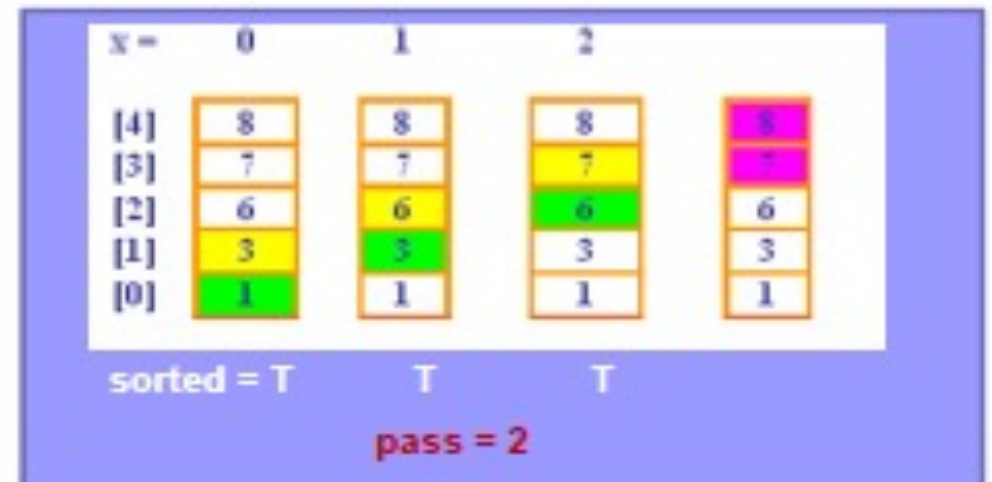
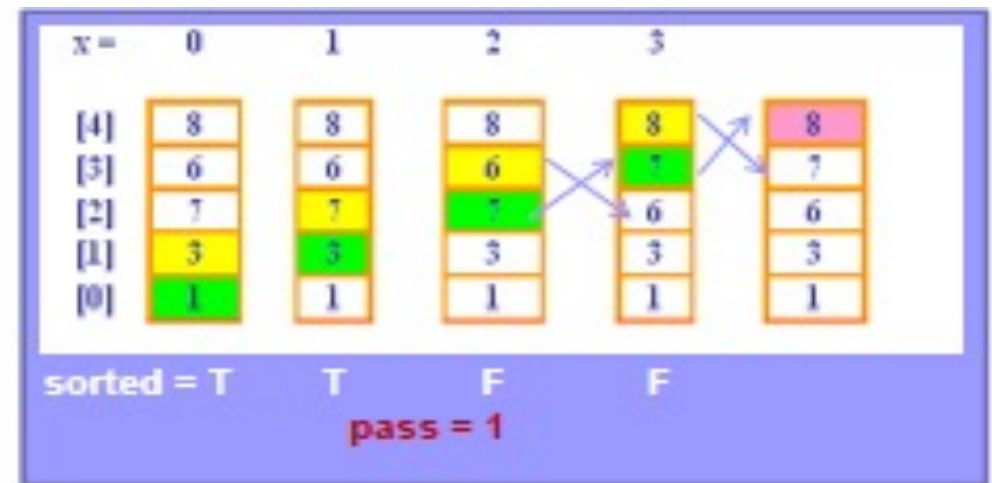
For **best case**, the **number of comparison** between elements is **4, (n-1)** which is **O(n)**.

Improved Bubble Sort : **Average Case** [1 3 7 6 8]

Average Case: [1 3 7 6 8]

We have to go through 2 passes only. The subsequent passes is not continued since the array is already sorted.

Conclusion: For improved Bubble Sort, the **sorting time** and the **number of comparisons** between data in average case and best case can be **minimized**.



Bubble Sort: **Algorithm Complexity**

- Time consuming operations
 - compares, swaps.
- **Number of Compares**
 - a *for* loop embedded inside a *while* loop
 - Worst Case $(n-1)+(n-2)+(n-3) \dots +1$, or **$O(n^2)$**
 - Best Case – $(n-1)$ or **$O(n)$**
- **Number of Swaps**
 - inside a conditional -> number of swaps data dependent !!
 - Best Case 0, or **$O(1)$**
 - Worst Case $(n-1)+(n-2)+(n-3) \dots +1$, or **$O(n^2)$**
- **Space**
 - size of the array
 - an *in-place* algorithm

Selection Sort

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Selection Sort

- Strategy
 - Choose the **largest / smallest** item in the array and place the item in its correct place
 - Choose the **next largest / next smallest** item in the array and place the item in its correct place.
 - **Repeat** the process until all items are **sorted**.
- **Does not depend on the initial arrangement** of the data
- Only appropriate for small n - $O(n^2)$ algorithm

Selection Sort

12	8	3	21	99	1
12	0	0	21	99	1
12	8	3	21	1	99
12	8	3	21	1	99
12	8	3	1	21	99
12	8	3	1	21	99
1	8	3	12	21	99
1	8	3	12	21	99
1	3	8	12	21	99
1	3	8	12	21	99
1	3	8	12	21	99

Start - Unsorted

Pass 1

Find the largest element in the array (99) and put at the last index of the array

Pass 2

Find the second largest element in the array (21) and put at the second last index of the array

Pass 3

Find the next largest element in the array (12) and put at the current last index of the array

Pass 4

Find the next largest element in the array (8) and put at the current last index of the array

Pass 5

Find the next largest element in the array (3) and put at the current last index of the array

selectionSort() Implementation

```
void selectionSort(DataType Data[], int n)
```

```
{
```

```
    for (int last = n-1; last >= 1; --last)
```

```
    {
```

```
        // select largest item in the Array
```

```
        int largestIndex = 0;
```

```
        // largest item is assumed start at index 0
```

```
        for (int p=1; p <= last; ++p)
```

```
        {
```

```
            if (Data[p] > Data[largestIndex])
```

```
                largestIndex = p;
```

```
        } // end for
```

```
        // swap largest item Data[largestIndex] with Data[last]
```

```
        swap(Data[largestIndex], Data[last]);
```

```
    } // end for
```

```
} // end selectionSort
```

last: index of the last item
in the subarray of items yet
to be sorted

largestIndex: index of the
largest item found

swap: change largest value
with item at last index of
the subarray

swap() function

```
void swap(DataType& x, DataType& y)
```

```
{
```

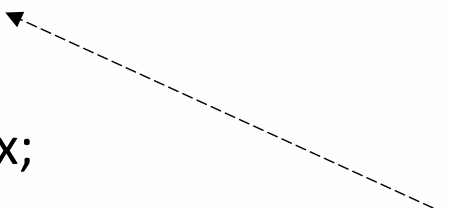
```
    DataType temp = x;
```

```
    x = y;
```

```
    y = temp;
```

```
} // end swap
```

Need to pass x and y by
reference



Selection Sort Implementation: [7 8 3 1 6]

```

void selectionSort(int Data[], int n)
{
    for (int last = n-1; last >= 1; --last)
    {
        // select largest item in theArray
        int largestIndex = 0;
        //largest item start at index 0
        for (int p=1; p <= last; ++p)
        {
            if (Data[p]>Data[largestIndex])
                largestIndex = p;
        }
        // end for
        // swap largest item Data[largest]
        // with Data[last]
        swap(Data[largestIndex],Data[last]);
    } // end for
} // end selectionSort
  
```

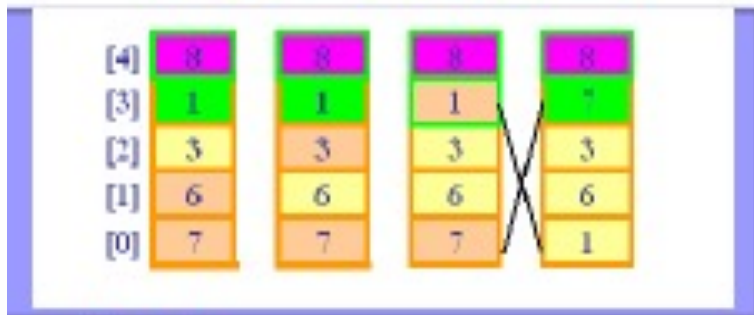


Pass 1
last = 4
largestIndex = 0, 1, 1, 1
p = 1, 2, 3, 4

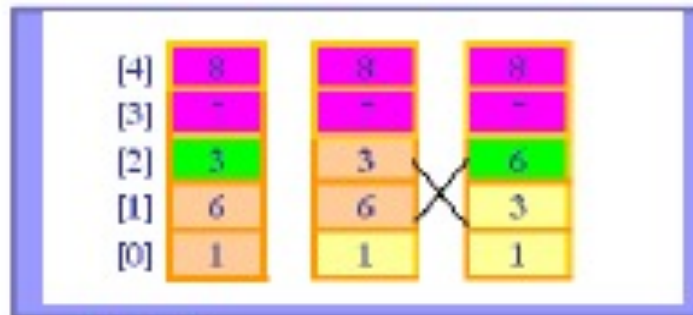
The largest value in the array will be searched from index 1 to index 4. In this pass, the largest value is 8 and was found at index 1 and will be put at last (4).

There are four comparisons in this pass.

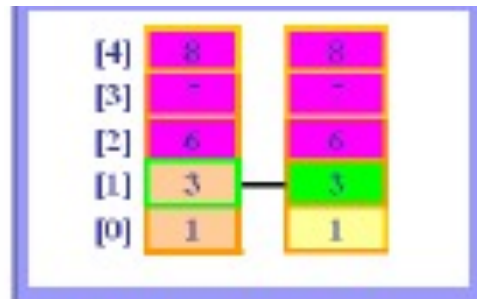
Selection Sort Implementation: [7 8 3 1 6] (continued...)



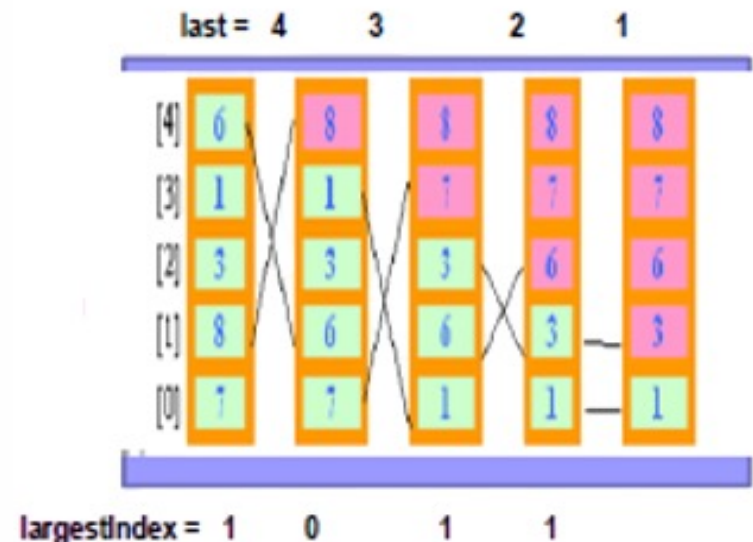
Pass 2
last = 3
largestIndex = 0, 0, 0
p = 1, 2, 3



Pass 3
last = 2
largestIndex = 0, 1
p = 1, 2

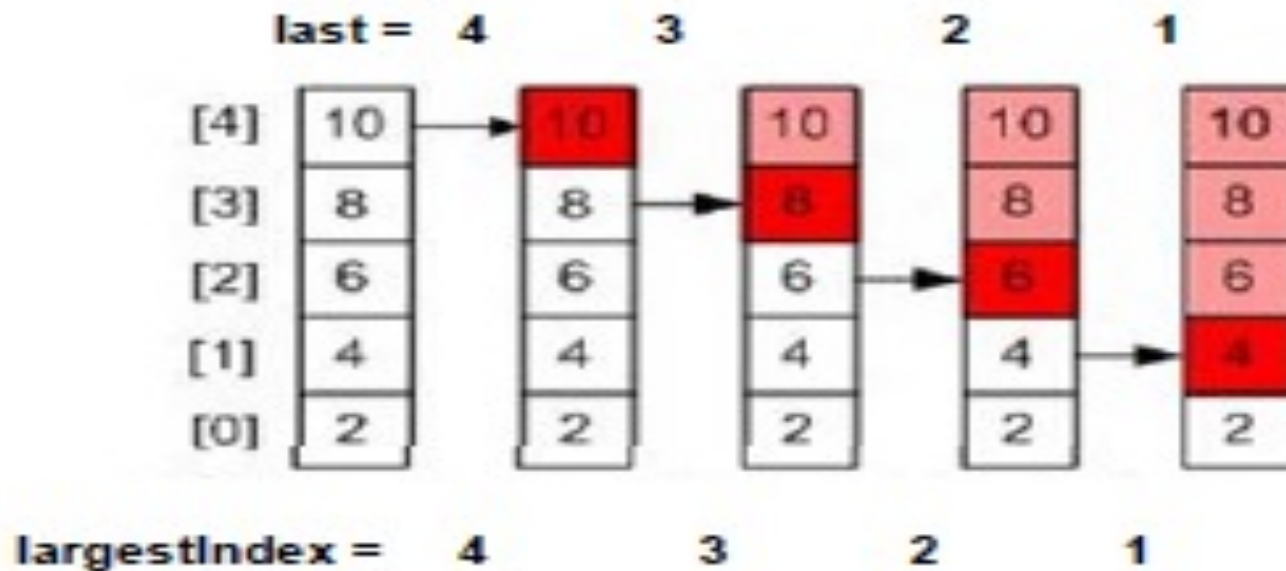


Pass 4
last = 1
largestIndex = 0, 1
p = 1



Step by step changes in the list that show the swapping process during selection sort implementation on array [7 8 3 1 6]

Selection Sort Implementation for Best Case [2 4 6 8 10]



Step by step changes in the list that show the swapping process during selection sort implementation on array [2 4 6 8 10]

Selection Sort Analysis

- For an array with size n , the **external loop** will iterate from $n-1$ to 1.

for (int last = n-1; last >= 1; --last)

- For each iteration, to find the largest number in subarray, the **number of comparison** inside the **internal loop** must equal to the value of last.

for (int p=1; p <= last; ++p)

- Therefore the **total comparison** for Selection Sort in each iteration is

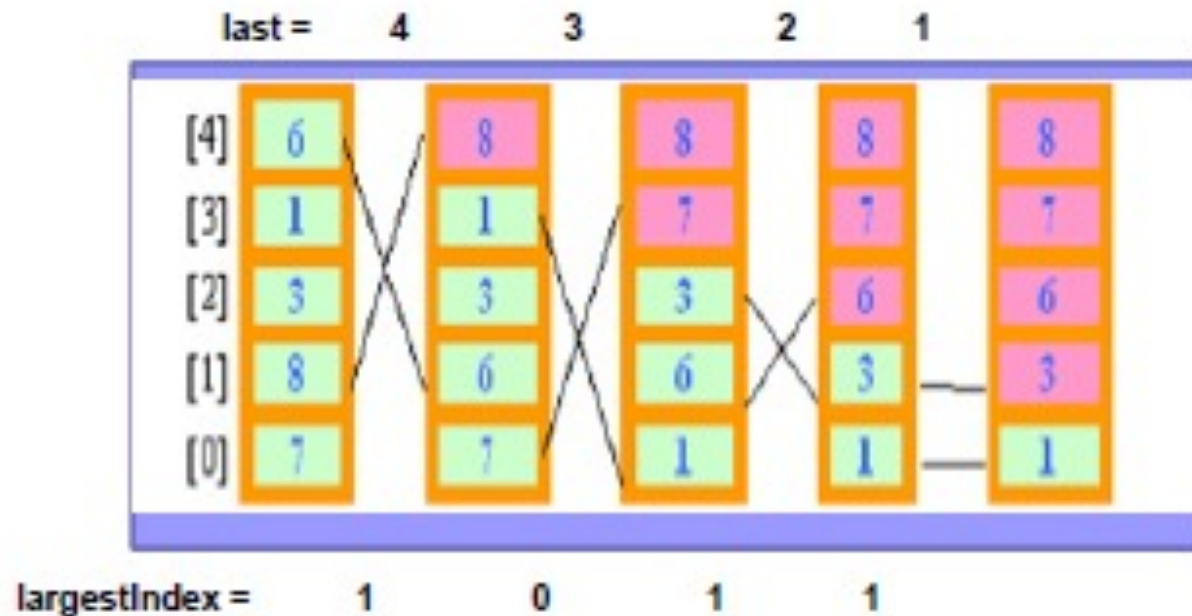
$(n-1) + (n-2) + \dots + 2 + 1$

- Generally, the **number of comparisons** between elements in Selection Sort can be stated as follows:

$$(n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$$

Selection Sort Analysis (continued...)

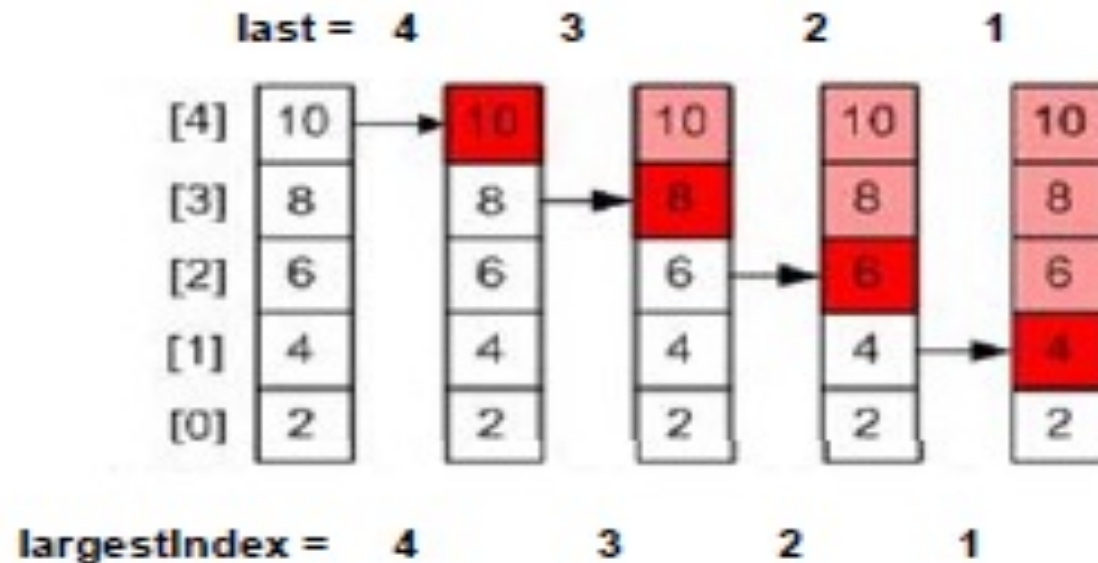
Like Bubble Sort, in any cases of Selection Sort (worse case, best case or average case) the number of comparisons between elements is the same.



Number of Comparisons: $4 + 3 + 2 + 1 = 10$

For array $n=5 \Rightarrow (n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2 = O(n^2)$

Selection Sort Analysis for Best Case [2 4 6 8 10]



Number of Comparisons for best case : $4 + 3 + 2 + 1 = 10$

For array $n = 5 \Rightarrow (n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2 = O(n^2)$

Selection Sort Issue

- It can be seen that the **swapping** process occur even though the **largest index is at last**.
- This is not efficient and can be improved by putting a condition statement as follows:

If (largestIndex != last);

swap(Data[largestIndex],Data[last]);

Selection Sort – Algorithm Complexity

- **Time Complexity** for Selection Sort is the same for all cases - worse case, best case or average case **$O(n^2)$** . The **number of comparisons** between elements is the same.
- The **efficiency** of Selection Sort **does not depend** on the **initial arrangement** of the data.

Insertion Sort

6 5 3 1 8 7 2 4

Insertion Sort

- Strategy
 - Take multiple passes over the array
 - Partition the array into two regions: **sorted** and **unsorted**
 - Take each item from the **unsorted region** and **insert** it into its **correct order** in the **sorted region**
 - Find **next unsorted** element and **insert** it in **correct place**, relative to the ones **already sorted**
- Analysis
 - Appropriate for **small arrays** due to its **simplicity**

Insertion Sort Implementation

21	8	3	12	99	1
21	8	3	12	99	1
8	21	3	12	99	1
8	21	3	12	99	1
3	8	21	12	99	1
3	8	21	12	99	1
3	8	12	21	99	1
3	8	12	21	99	1
3	8	12	21	99	1
3	8	12	21	99	1
1	3	8	12	21	99

Start - Unsorted

Insert 8 before 21

Insert 3 before 8

Insert 12 before 21

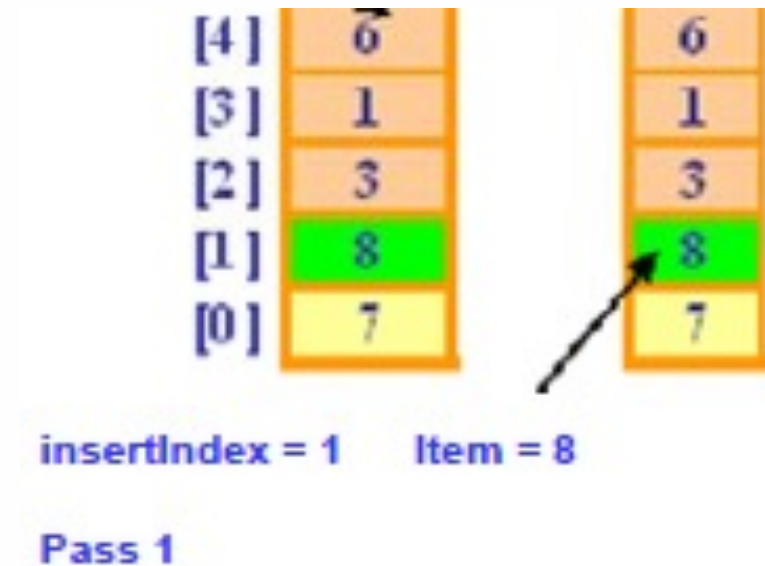
Keep 99 in place

Insert 1 before 3

Insertion Sort Implementation [7 8 3 1 6]

```

void insertionSort(dataType data[])
{
    dataType item;
    int pass, insertIndex;
    for (pass=1; pass<n; pass++)
    {
        item = data[pass];
        insertIndex = pass;
        while((insertIndex > 0) && (data[insertIndex -1] > item))
        {
            // insert the right item
            data[insertIndex] = data[insertIndex -1];
            insertIndex --;
        }
        data[insertIndex] = item;
        // insert item at the right place
    } // end for
}
  
```



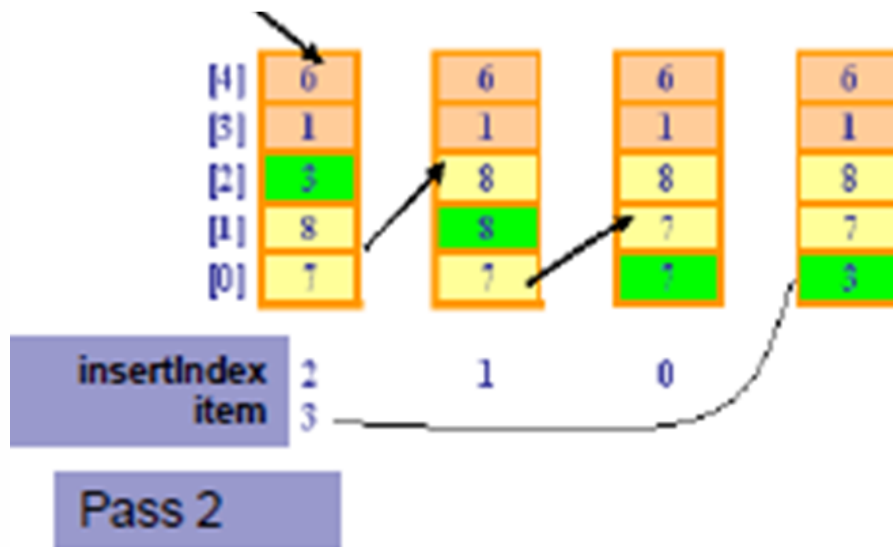
item=8 > data[0]=7.

while loop condition is false,
therefore data[1] will be assigned
with item = 8.

No of comparison = 1

Insertion Sort Implementation [7 8 3 1 6] (continued...)

Original list

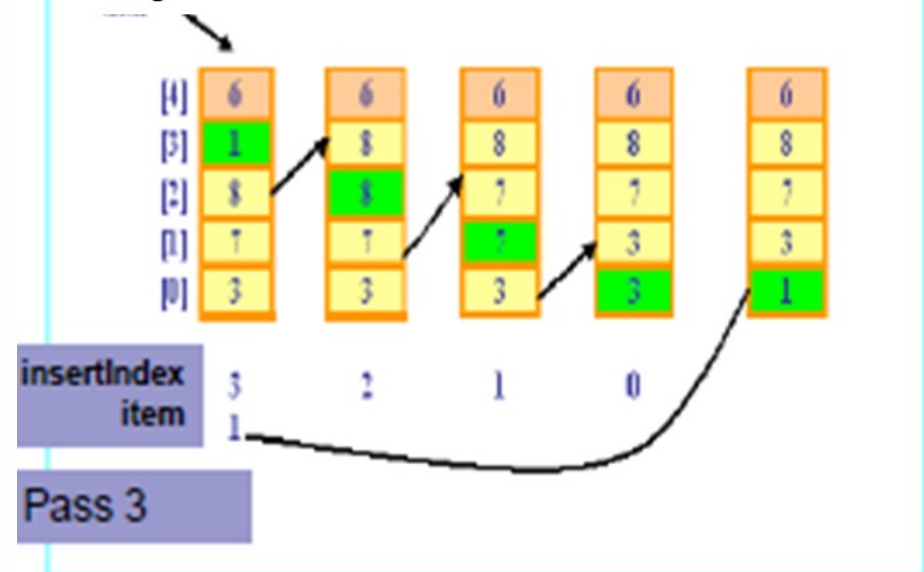


Item to be insert is 3.

Insertion point is from index 0-2, which is between 7 and 8.

No of comparison = 2

Original list



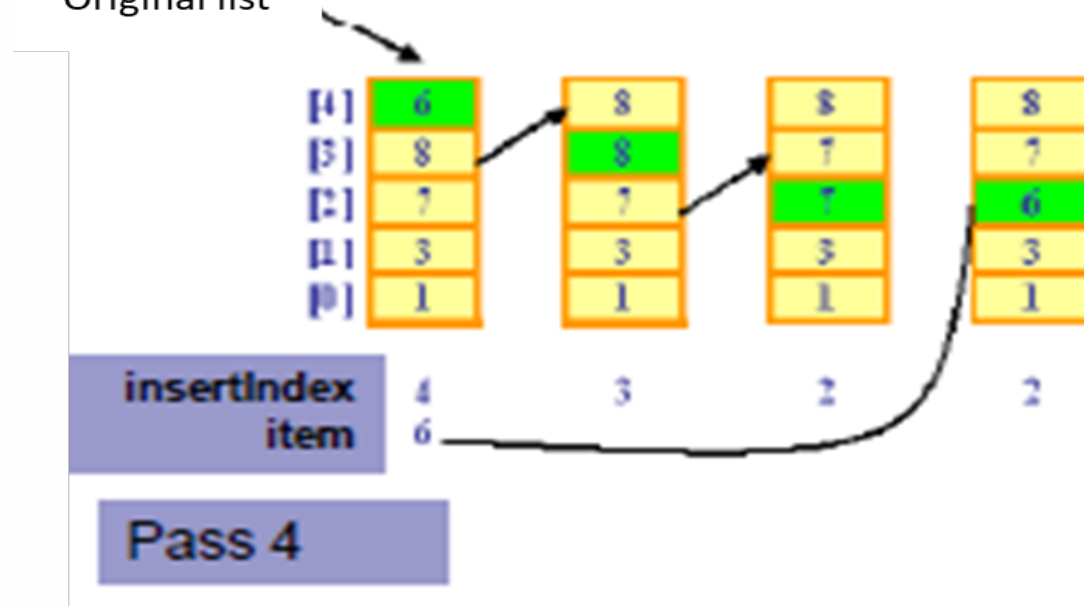
Item to be insert is 1.

Insertion point is from index 0-3, which is between 3, 7, and 8.

No of comparison = 3

Insertion Sort Implementation [7 8 3 1 6] (continued...)

Original list



Item to be insert is 6.

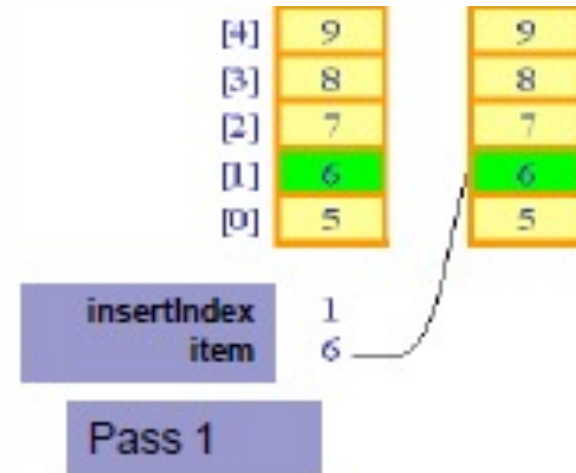
Insertion point is from index 0-4, which is between 1, 3, 7, and 8.

at index, item (6) > data[1]=3, while loop condition is false and therefore data[2] is assigned with value for item = 6.

No of comparison = 3

Insertion Sort for Best Case [5 6 7 8 9]

```
void insertionSort(dataType data[])
{
    dataType item;
    int pass, insertIndex;
    for (pass=1; pass<n; pass++)
    {
        item = data[pass];
        insertIndex = pass;
        while((insertIndex > 0) && (data[insertIndex -1] > item))
        {
            // insert the right item
            data[insertIndex] = data[insertIndex -1];
            insertIndex --;
        }
        data[insertIndex] = item;
        // insert item at the right place
    } // end for
}
```



item=6 > data[0]=5.

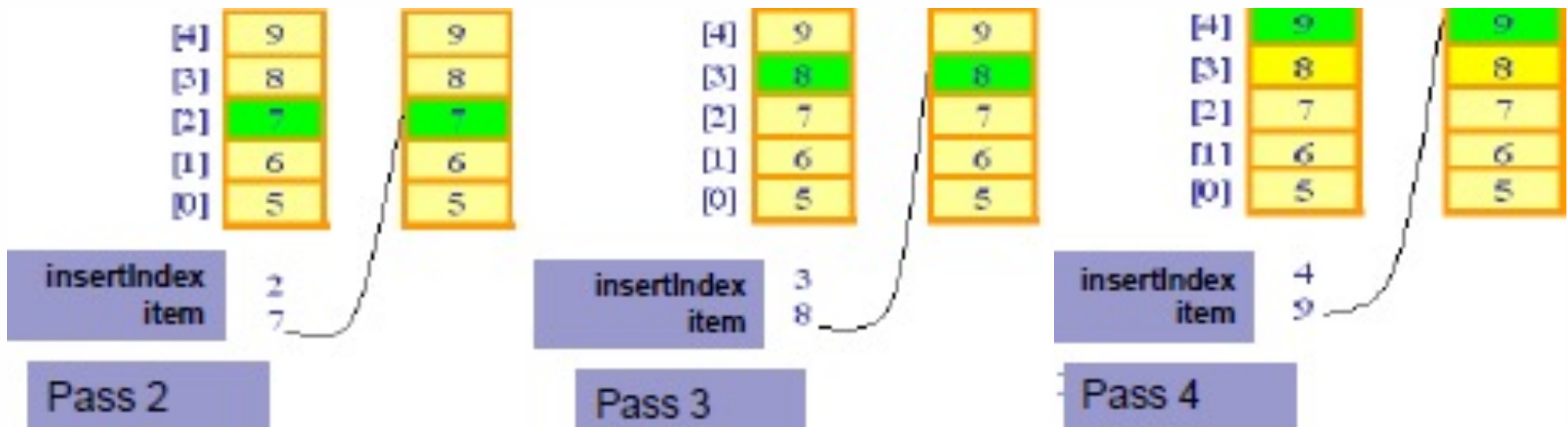
while condition is false and data[1] is assigned with item=6.

No of comparison = 1

Note:

Best Case for Insertion Sort can be achieved when data is almost sorted or totally sorted. Each pass will have 1 comparison only.

Insertion Sort for Best Case [5 6 7 8 9] (continued...)



item=7 > data[1]=6.

while condition is false and data[2] is assigned with item=7.

No of comparison = 1

item=8 > data[2]=7.

while condition is false and data[3] is assigned with item=8.

No of comparison = 1

item=9 > data[3]=8.

while condition is false and data[4] is assigned with item=9.

No of comparison = 1

Insertion Sort Analysis – Best Case

There are 4 passes to sort array with elements [5 6 7 8 9].

In each pass there is only 1 comparison.

Example,

Pass 1, 1 comparison

Pass 2, 1 comparison

Pass 3, 1 comparison

Pass 4, 1 comparison

In this example, the total comparisons for an array with size 5 is 4.

Therefore, for **best case**, the **number of comparison is $n-1$** which gives **linear time complexity - linear $O(n)$** .

Insertion Sort Analysis – Worse Case

Worse case for insertion sort is when we have **totally unsorted** data.

In each pass, the **number of iteration for while loop is maximum**.

Pass 4, 4 comparison - (n-1)

Pass 3, 3 comparison -(n-2)

Pass 2, 2 comparison -(n-3)

Pass 1, 1 comparison - (n-4)

The number of comparisons between elements in Insertion Sort is stated as follows:

$$\sum_{i=1}^{n-1} i = (n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$$

Insertion Sort for Worse Case [9 7 5 3 1]

```
void insertionSort(dataType data[])
{
    dataType item;
    int pass, insertIndex;
    for (pass=1; pass<n; pass++)
    {
        item = data[pass];
        insertIndex = pass;
        while((insertIndex > 0) && (data[insertIndex -1] > item))
        {
            // insert the right item
            data[insertIndex] = data[insertIndex -1];
            insertIndex --;
        }
        data[insertIndex] = item;
        // insert item at the right place
    } // end for
}
```

Pass 4 –

Have to compare data at data[4-1], data[4-2], data[4-3], and data[4-4].

Pass 3 –

Have to compare data at data[3-1], data[3-2], and data[3-3].

Pass 2 –

Have to compare data at data[2-1], and data[2-2].

Pass 1 –

Have to compare data at data[1-1] only.

Insertion Sort – Algorithm Complexity

- How many compares are done?
 - $1 + 2 + \dots + (n-1)$, **$O(n^2)$ worst case**
 - $(n-1) * 1$, **$O(n)$ best case**
- How many element shifts are done?
 - $1+2+\dots+(n-1)$, **$O(n^2)$ worst case**
 - 0 , **$O(1)$ best case**
- How much space?
 - In-place algorithm

Worst Case Complexity

- **Bubble Sort**
 - Number of Comparisons: $O(n^2)$
 - Number of Swaps: $O(n^2)$
- **Selection Sort**
 - Number of Comparisons: $O(n^2)$
 - Number of Swaps: $O(n)$
- **Insertion Sort**
 - Number of Comparisons: $O(n^2)$
 - Number of Swaps: $O(n^2)$

Summary

Types of Sort	Insert Sort	Bubble Sort	Selection Sort
Number of Comparisons			
Best Case	$O(n)$	$O(n^2)$	$O(n^2)$
Average Case	$O(n^2)$	$O(n^2)$	$O(n^2)$
Worst Case	$O(n^2)$	$O(n^2)$	$O(n^2)$
Number of Swaps			
Best Case	0	0	$O(n)$
Average Case	$O(n^2)$	$O(n^2)$	$O(n)$
Worst Case	$O(n^2)$	$O(n^2)$	$O(n)$

References

Nor Bahiah et al. Struktur data & Algoritma Menggunakan C++. UTM Press, 2005

Next...



Do you have
any
Questions?

