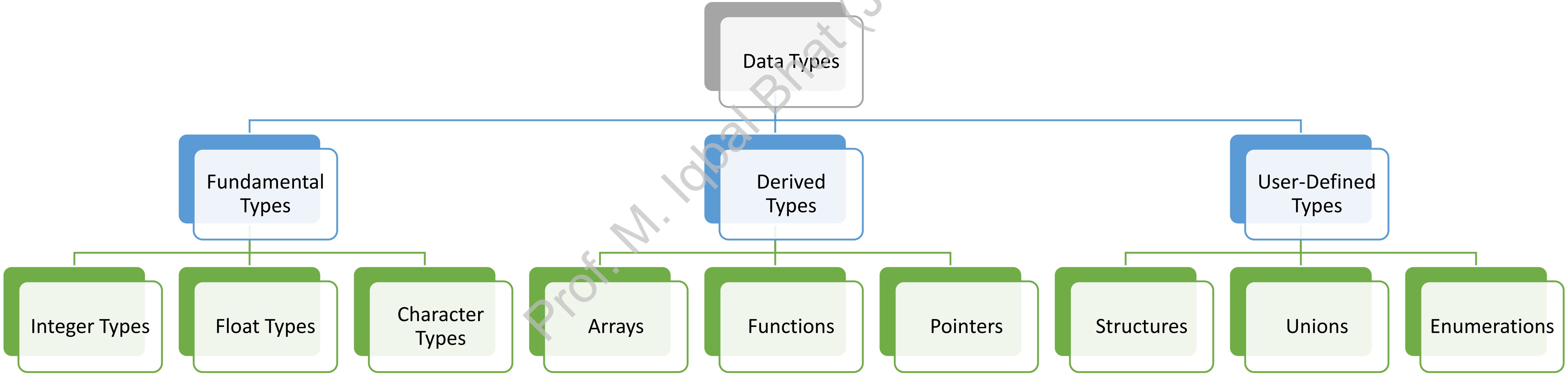
Arrays in C Language

Presented by:
Muhammad Iqbal Bhat
Assistant Professor,
Department of Higher Education J&K (UT)

Topics

- Data Types in C
- Array Basics
- Declaration and Initialization
- Accessing Array Data
- Sorting Arrays
- Multidimensional Arrays

Data Types in C



Understanding Arrays:

Motivation

If we have to write a program to get marks of 3 students, then

- int marks_student1, marks_student2, marks student3;

What About 100 students

- int marks_student1, marks_student2, …. Marks_student100;

Use arrays:

- int marks[100];

What is an Array:

- **An array is a sequenced collection of related data items that share a common name.**

Concise and Efficient Programs:

- **The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enable us to develop concise and efficient programs.**

What are arrays?

Arrays in C are a data structure that allow you to store multiple values of the same data type in a contiguous memory space.

All the elements of an array are stored in consecutive memory locations, and each element can be accessed using an index.

Arrays are useful for handling large amounts of data efficiently.

The size of an array is fixed and must be specified when the array is declared.

Arrays can only hold values of a single data type.

Arrays are commonly used in C programming to store data such as lists, matrices, and strings.

How arrays look like

value

address

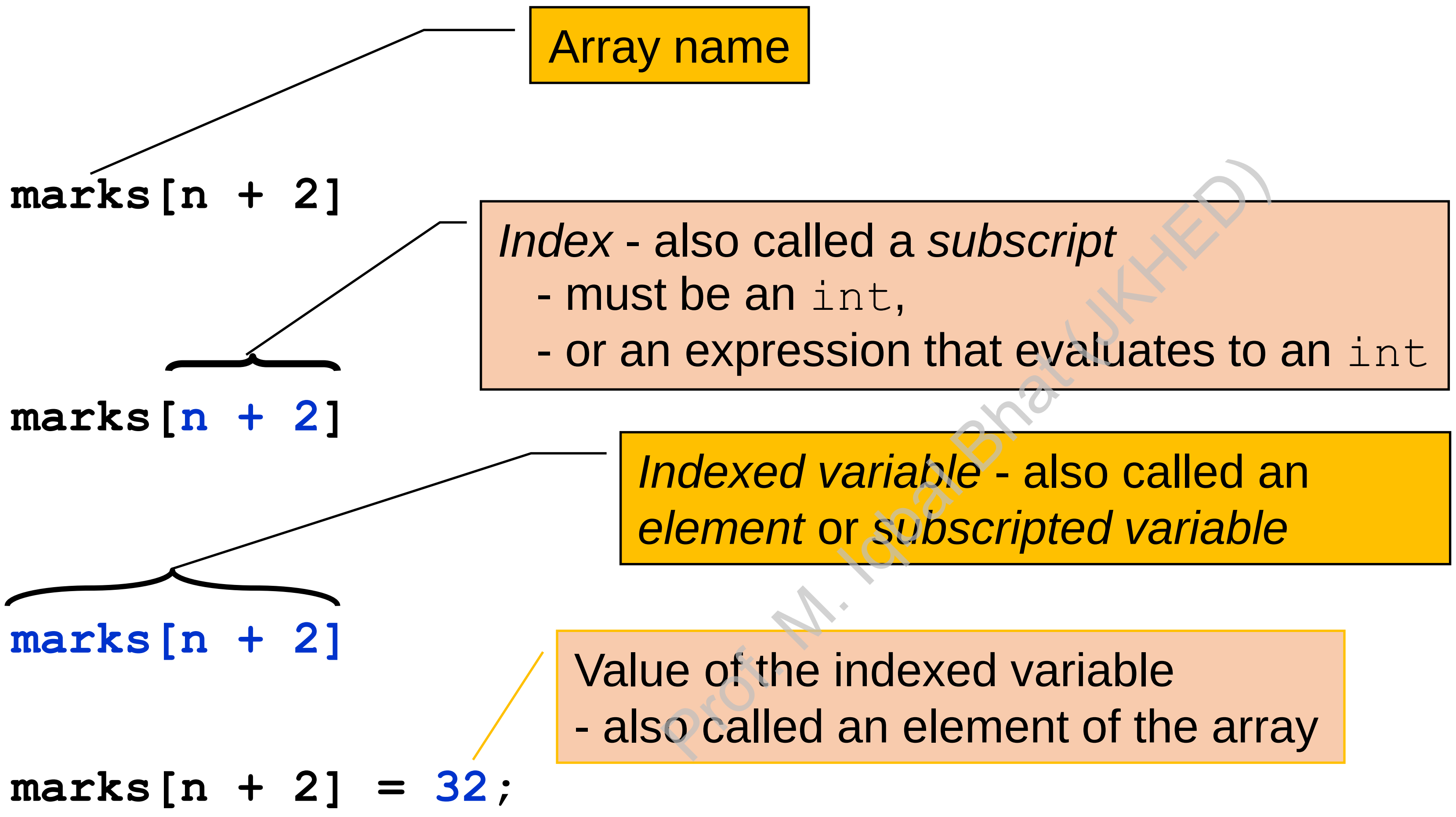
90	**80**	**70**	**50**	**20**	**98**	**20**	**60**	**80**	**90**
0	1	2	3	4	5	6	7	8	9
1000	1002	1004	1006	1008	1010	1012	1014	1016	1018
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

index

int marks[10];

access

Some Array Terminology



Subscript Range (0 to n)

- Array subscripts use zero-numbering
  - the first element has subscript 0
  - the second element has subscript 1
  - etc. - the n^{th} element has subscript n-1
  - the last element has subscript `length-1`
- For example: an int array with 4 elements

Subscript:	0	1	2	3
Value:	97	86	92	71

Programming Tip: Use Singular Array Names

- Using singular rather than plural names for arrays improves readability
- Although the array contains many elements the most common use of the name will be with a subscript, which references a *single* value.
- It is easier to read:
  - `score[3]` than
  - `scores[3]`

Initializing an Array's Values in Its Declaration

- can be initialized by putting a comma-separated list in braces
- The length of an array is automatically determined when the values are explicitly initialized in the declaration
- For example:

```
int marks[] = {90, 20, 40};
```

Initializing Array Elements in a Loop

- A `for` loop is commonly used to initialize array elements
- For example:

```
int i;           /*loop counter array index */
int a[10];
for(i = 0; i < 10; i++)
   a[i] = 0;
```

  - note that the loop counter/array index goes from `0` to `length - 1`
  - it counts through `length = 10` iterations/elements using the zero-numbering of the array index

Programming Tip:

Do not count on default initial values for array elements

- explicitly initialize elements in the declaration or in a loop

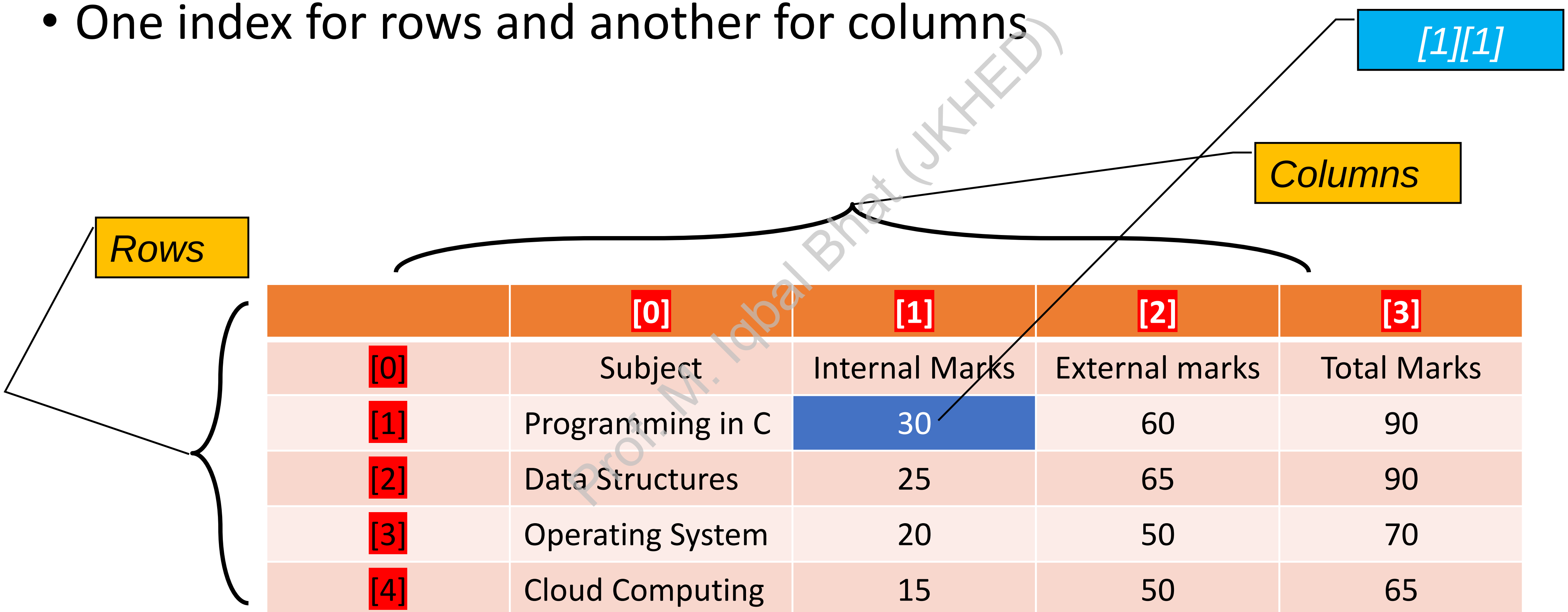
Types of Arrays

Types of Arrays

- One Dimensional (1D)
- Two Dimensional (2D)
- Multidimensional (ND)

Two Dimensional Arrays (2D arrays)

- A 2D array has two subscripts/indexes.
- One index for rows and another for columns



	[0]	[1]	[2]	[3]
[0]	Subject	Internal Marks	External marks	Total Marks
[1]	Programming in C	30	60	90
[2]	Data Structures	25	65	90
[3]	Operating System	20	50	70
[4]	Cloud Computing	15	50	65

Two Dimensional Arrays (2D arrays)

- General Syntax for Declaration:
- datatype array_name [row_size][col_size];

[1][1]

Columns

Rows

	[0]	[1]	[2]	[3]
[0]	Subject	Internal Marks	External marks	Total Marks
[1]	Programming in C	30	60	90
[2]	Data Structures	25	65	90
[3]	Operating System	20	50	70
[4]	Cloud Computing	15	50	65

Initializing 2D arrays

```
int array[2][3]=    {
                        {1,2,3},
                        {4,5,6}
                    };
```

1 [0,0]	2 [0,1]	3 [0,2]
4 [1,0]	5 [1,1]	6 [1,2]

Addition of Two Matrices

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 9 & 11 \end{bmatrix} = \begin{bmatrix} 1+5 & 2+6 \\ 3+9 & 4+11 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 12 & 15 \end{bmatrix}$$

Examples of Arrays

1. Program to find the sum of elements in an array:

```
#include <stdio.h>
int main() {
  int array[5] = {1, 2, 3, 4, 5};
  int sum = 0;

  for (int i = 0; i < 5; i++) {
    sum += array[i];
  }

  printf("The sum of the elements in the array is: %d\n", sum);

  return 0;
}
```

2. Program to find the maximum element in an array:

```
#include <stdio.h>

int main() {
  int array[5] = {10, 23, 5, 17, 8};
  int max = array[0];

  for (int i = 1; i < 5; i++) {
    if (array[i] > max) {
      max = array[i];
    }
  }

  printf("The maximum element in the array is: %d\n", max);

  return 0;
}
```

3. Program to sort an array in ascending order:

```
#include <stdio.h>
 int main() {
  int array[5] = {3, 1, 4, 2, 5};
  int temp;
   for (int i = 0; i < 5; i++) {
    for (int j = i + 1; j < 5; j++)
{
      if (array[i] > array[j]) {
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
      }
    }
  }
printf("The sorted array in
ascending order is: ");
   for (int i = 0; i < 5; i++) {
    printf("%d ", array[i]);
  }
   printf("\n");
   return 0;
}
```

4. Program to search for an element in an array::

```
#include <stdio.h>

int main() {
  int array[5] = {10, 23, 5, 17, 8};
  int element = 17;
  int found = 0;

  for (int i = 0; i < 5; i++) {
    if (array[i] == element) {
      found = 1;
      break;
    }
  }
if (found) {
    printf("The element %d is found in the array.\n", element);
  } else {
    printf("The element %d is not found in the array.\n", element);
  }

  return 0;
}
```

5. Program to reverse the elements in an array:

```
#include <stdio.h>

int main() {
  int array[5] = {1, 2, 3, 4, 5};
  int temp;

  printf("The original array is: ");

  for (int i = 0; i < 5; i++) {
    printf("%d ", array[i]);
  }

  for (int i = 0, j = 4; i < j; i++, j--) {
    temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }

printf("\nThe reversed array is: ");

  for (int i = 0; i < 5; i++) {
   printf("%d ", array[i]);
  }

  printf("\n");

  return 0;
}
```

Questions?

Sorting an Array

- Sorting a list of elements is another very common problem (along with searching a list)
  - sort numbers in ascending order
  - sort numbers in descending order
  - sort strings in alphabetic order
  - etc.

- There are many ways to sort a list, just as there are many ways to search a list

- *Selection sort*
  - one of the easiest
  - not the most efficient, but easy to understand and program

Selection Sort Algorithm for an Array of Integers

To sort an array on integers in ascending order:

1. Find the smallest number and record its index
2. swap (interchange) the smallest number with the first element of the array
   - the sorted part of the array is now the first element
   - the unsorted part of the array is the remaining elements
3. repeat Steps 2 and 3 until all elements have been placed
   - each iteration increases the length of the sorted part by one

Selection Sort Example

Key:
- smallest remaining value (yellow)
- sorted elements (blue)

Problem: sort this 10-element array of integers in ascending order:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
7	6	11	17	3	15	5	19	30	14

1st iteration: smallest value is 3, its index is 4, swap a[0] with a[4]

before:	7	6	11	17	3	15	5	19	30	14
after:	3	6	11	17	7	15	5	19	30	14

2nd iteration: smallest value in remaining list is 5, its index is 6, swap a[1] with a[6]

3	6	11	17	7	15	5	19	30	14
3	5	11	17	7	15	6	19	30	14

How many iterations are needed?

Example: Selection Sort

- Notice the precondition: every array element has a value
- may have duplicate values
- broken down into smaller tasks
  - "find the index of the smallest value"
  - "interchange two elements"
  - `private` because they are helper methods (users are not expected to call them directly)

```java
/*****************************************************
*Precondition:
*Every indexed variable of the array a has a value.
*Action: Sorts the array a so that
*a[0] <= a[1] <= ... <= a[a.length - 1].
*****************************************************/
public static void sort(int[] a)
{
   int index, indexOfNextSmallest;
   for (index = 0; index < a.length - 1; index++)
   {//Place the correct value in a[index]:
       indexOfNextSmallest = indexOfSmallest(index, a);
       interchange(index,indexOfNextSmallest, a);
       //a[0] <= a[1] <=...<= a[index] and these are
       //the smallest of the original array elements.
       //The remaining positions contain the rest of
       //the original array elements.
   }
}
```

Insertion Sort

- Basic Idea:
  - Keeping expanding the sorted portion by one
  - Insert the next element into the right position in the sorted portion
- Algorithm:
  1. Start with one element [is it sorted?] – sorted portion
  2. While the sorted portion is not the entire array
     1. Find the right position in the sorted portion for the next element
     2. Insert the element
     3. If necessary, move the other elements down
     4. Expand the sorted portion by one

Insertion Sort: An example

- First iteration
  - Before: **[5]**, 3, 4, 9, 2
  - After: [3, **5]**, 4, 9, 2
- Second iteration
  - Before: **[3, 5]**, 4, 9, 2
  - After: **[3**, 4, **5]**, 9, 2
- Third iteration
  - Before: **[3, 4, 5]**, 9, 2
  - After: **[3, 4, 5**, 9**]**, 2
- Fourth iteration
  - Before: **[3, 4, 5, 9]**, 2
  - After: [2, **3, 4, 5, 9]**

Bubble Sort

- Basic Idea:
  - Expand the sorted portion one by one
  - “Sink” the largest element to the bottom after comparing adjacent elements
  - The smaller items “bubble” up
- Algorithm:
  - While the unsorted portion has more than one element
    - Compare adjacent elements
    - Swap elements if out of order
    - Largest element at the bottom, reduce the unsorted portion by one

Bubble Sort: An example

- First Iteration:
  - [5, 3], 4, 9, 2 → [3, 5], 4, 9, 2
  - 3, [5, 4], 9, 2 → 3, [4, 5], 9, 2
  - 3, 4, [5, 9], 2 → 3, 4, [5, 9], 2
  - 3, 4, 5, [9, 2] → 3, 4, 5, [2, 9]
- Second Iteration:
  - [3, 4], 5, 2, **9** → [3, 4], 5, 2, **9**
  - 3, [4, 5], 2, **9** → 3, [4, 5], 2, **9**
  - 3, 4, [5, 2], **9** → 3, 4, [2, 5], **9**
- Third Iteration:
  - [3, 4], 2, **5, 9** → [3, 4], 2, **5, 9**
  - 3, [4, 2], **5, 9** → 3, [2, 4], **5, 9**
- Fourth Iteration:
  - [3, 2], **4, 5, 9** → [2, 3], **4, 5, 9**

How to Compare Algorithms in Efficiency (speed)

- Empirical Analysis
  - Wall-clock time
  - CPU time
  - Can you predict performance before implementing the algorithm?
- Theoretical Analysis
  - Approximation by counting important operations
  - Mathematical functions based on input size (*N*)

How Fast/Slow Can It Get?
(10G Hz, assume 10^{10} operations/sec)

N	$N\log_2 N$	N^2	2^N
10	33	100	1,024
100 (10^{-8} sec)	664	10,000	1.3 x 10^{30} (4 x10^{12} years)
1,000	9,966	1,000,000	Forever??
10,000	132,877	100,000,000	Eternity??

Theoretical Analysis (Sorting)

- Counting important operations
  - Comparisons (array elements)
    - >, <, …
  - Swaps/moves (array elements)
    - 1 swap has 3 moves
- Comparison is the more important operation—could be expensive
- Size of input (*N*) = Number of array elements
- Three cases for analysis
  - Worst case (interesting, popular analysis)
  - Best case (not so interesting)
  - Average case (discussed in another course)

Selection Sort

- Comparisons
  - $N - 1$ iterations
  - First iteration: how many comparisons?
  - Second iteration: how many comparisons?
  - $(N - 1) + (N - 2) + \ldots + 2 + 1 = N(N-1)/2 = (N^2 - N)/2$
- Moves (worst case: every element is in the wrong location)
  - $N - 1$ iterations
  - First iteration: how many swaps/moves?
  - Second iteration: how many swaps/moves?
  - $(N - 1) \times 3 = 3N - 3$

Insertion Sort

- Comparisons (worst case: correct order)
  - $N - 1$ iterations
  - First iteration: how many comparisons?
  - Second iteration: how many comparisons?
  - $1 + 2 + \ldots + (N - 2) + (N - 1) = N(N-1)/2 = (N^2 - N)/2$
- Moves (worst case: reverse order)
  - $N - 1$ iterations
  - First iteration: how many moves?
  - Second iteration: how many moves?
  - $3 + 4 + \ldots + N + (N + 1) = (N + 4)(N - 1)/2 = (N^2 + 3N - 4)/2$

Bubble Sort

- Comparisons
  - $N - 1$ iterations
  - First iteration: how many comparisons?
  - Second iteration: how many comparisons?
  - $(N - 1) + (N - 2) + \ldots + 2 + 1 = N(N-1)/2 = (N^2 - N)/2$
- Moves (worst case: reverse order)
  - $N - 1$ iterations
  - First iteration: how many swaps/moves?
  - Second iteration: how many swaps/moves?
  - $[(N - 1) + (N - 2) + \ldots + 2 + 1] \times 3 = 3N(N-1)/2 = (3N^2 - 3N)/2$

Summary of Worst-case Analysis

	Comparisons (more important)	Moves
Selection	$(N^2 – N)/2$	$3N - 3$
Insertion	$(N^2 – N)/2$	$(N^2 + 3N - 4)/2$
Bubble	$(N^2 – N)/2$	$(3N^2 – 3N)/2$

Sorting Algorithm Tradeoffs

- Easy to understand algorithms
  - not very efficient
  - less likely to have mistakes
  - require less time to code, test, and debug
  - Selection, Insertion, Bubble Sorting algorithms
  - Bubble Sort is the easiest to implement
- Complicated but more efficient
  - useful when performance is a major issue
  - programming project for Chapter 11 describes a more efficient sorting algorithm

"Getting the wrong result is always inefficient."