



**ICS-103**  
**Computer Programming in C**

**Lectures 22-24**

**Chapter 7.6**

**Searching, Sorting, and 2D Arrays**

**Dr. Tarek Ahmed Helmy El-Basuny**

## Outline of Ch. 07 Topics

- What is Searching?
- Why do we use a searching algorithm?
- **Linear**/Sequential Search Algorithm,
- Implementing **Linear** search algorithm
- Program Example of Searching in Arrays using Linear Search.
- The main idea of **Binary** Search Algorithm,
- Implementation of Binary Search Algorithm
- Animating Examples of Searching Using **Binary** Search.
- Program Example of Searching in Arrays using **Binary** Search.
- **What is Sorting?**
- **Why do we use Sorting?**
- **Selection Sort Algorithm,**
- **Program Example of Sorting Arrays by using Selection Sort Algorithm.**
- **Declaring and initializing 2D Arrays.**
- **Indexing and Processing 2D Arrays Elements**
- **Program Examples of Using 2D Arrays**

FYI , not  
for exams

# What is Searching?

- Searching is an every day activity we may do: i.e.
  - Searching for a particular item among a collection of many items.
  - Search for a person's Telephone number in a telephone book. **You need to search for the Tel. as an "int". How difficult is this?**
  - Suppose you have the same telephone book and you want to find a person's name that **has a certain telephone number. You need to search for the name as a "string". How would you find it? Why is this more difficult?**
  - Suppose you have the same telephone book where its content is **alphabetically sorted. You know the person's name and you want to search for a person's telephone number. How difficult is this?**
- Searching algorithm is a method of locating a specific item of information in a larger collection of data.



# Searching Problem

- Searching for data is one of the fundamental processes of computing.
- Often, **the difference between a fast program and a slow one** is the use of a good searching algorithm **for the searching in the data set.**
- There are many algorithms to perform searching. **We will briefly explore the two commonly used:**
  - **The Linear/Sequential Search,**
    - Start at the beginning of the collection/**list** and walk to the end, testing for a match at each item in the collection/**list.**
  - **The Binary Search (used with sorted collections)**
    - Start to test of the match at the middle of the **list/collection (because the list is sorted)**, based on the result we could determine in which half the item is in, and **recursively search that half.**
- A question you should always ask when selecting a searching algorithm is **“How fast does the search have to be?”**
- In general, **the faster the algorithm is, the more complex it is.**

## Linear/**Sequential** Search Algorithm

- ❑ The linear or (**Sequential**) search algorithm is used with unsorted collection/list. It can be used with sorted list but it is not recommended.
  - It sequentially scans the array, comparing each item with the target value.
  - If a match is found; then return the index of the matched element;
    - otherwise return  $-1$ .
- ❑ The **pseudo-code** of the linear search algorithm
  - While the target is not found and there are more array elements,
  - Start with first array element and compare it with the target,
    - If the current element matches the target,
      - Set a flag to indicate that the target was found, and return the target index as the search result.
      - Else, advance to the next array element, or return  $-1$  as the search result.

## Linear Search Implementation

```
/* Search array a[] for target using linear search  
 * Returns the index of target or -1 if not found */
```

```
int linear_search(const int a[], int target, int n) {  
    int i = 0, found = 0;  
  
    while (!found && i < n) {//we did not find the target and there are elements  
        if (a[i] == target)  
            found = 1;  
        else  
            ++i;  
    }  
    if (found) return i;  
    else return -1;  
}
```

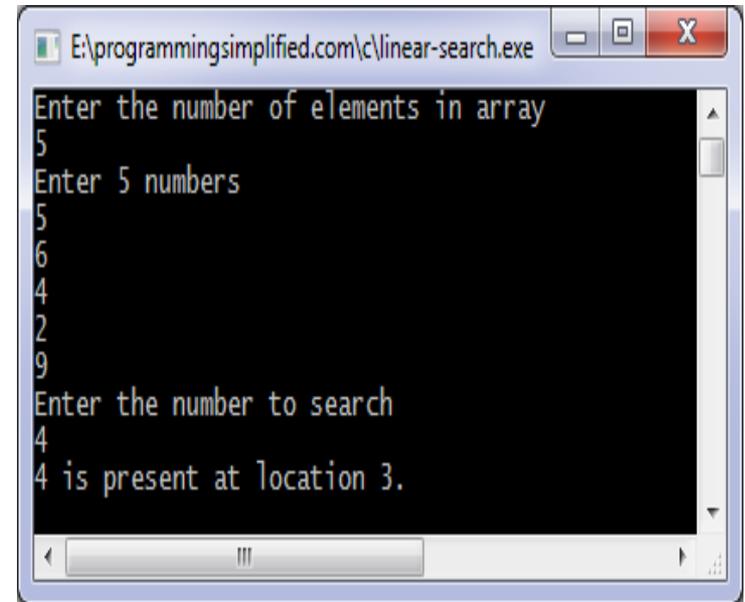
- For an array of **n** elements, linear search uses an average of **n/2** comparisons to find an item.
- The worst case being **n** comparisons.

## Example: Searching in Arrays using Linear Search

*/\* a program that reads number of elements, fill them into an array then asks the user to enter a number to check for it in the array \*/*

```
#include <stdio.h>

int main() {
    int array[100], search, c, n;
    printf("Enter the number of elements in array\n");
    scanf("%d",&n);
    printf("Enter %d integer(s)\n", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
    printf("Enter the number to search\n");
    scanf("%d", &search);
    for (c = 0; c < n; c++) {
        if (array[c] == search) /* if required element found */
            { printf("%d is present at location %d.\n", search, c+1);
              break; } }
    if (c == n) printf("%d is not present in array.\n", search);
    return 0; }
```



```
E:\programmingsimplified.com\c\linear-search.exe
Enter the number of elements in array
5
Enter 5 numbers
5
6
4
2
9
Enter the number to search
4
4 is present at location 3.
```

# Binary Search Algorithm

- If an array is sorted, it is a waste of time to look for an item using **Linear search**. It would be like looking for a word in a dictionary sequentially.
- When the data collection to search in is **sorted**, we can use **Binary** search.
- With **Binary** search, we can ignore one-half of the collection and focus on the other half.



- ❑ **Binary search** works by comparing the target with the item at the **middle** of the array/list:
  1. If the target matches the middle item, then we are done.
  2. If the target is less than the middle item, then we will search the first half of the array.
  3. If the target is bigger than the middle item, then we will search the second half of the array.
- ❑ Repeat until target is found or nothing to search.

# Binary Search

- ❑ **Linear** search has linear time complexity:
  - ⇒ Time is  $n$  if the item is not found,
  - ⇒ Time is  $n/2$ , on average, if the item is found.
- ❑ If the array is sorted, we can write a faster search where the average time could be  $n/4$ . That is called Binary search.

# Binary Search Algorithm

- To find which element (**if any**) of  $a[\text{left}..\text{right}]$  is equal to  $\text{key}/\text{target}$  (where  $a$  is sorted in ascending order).

Set  $l = \text{left}$  (0), and  $r = \text{right}$  (size-1)

While  $l \leq r$ , repeat: // means there is some elements in the list

- Let  $m$  be an index of an element about the middle between  $l$  and  $r$
- $m = (l + r) / 2$
- If  $\text{key}/\text{target}$  is equal to  $a[m]$ , terminate with answer  $m$ .
- If  $\text{key}/\text{target}$  is less than  $a[m]$ , set  $r = m - 1$ , Binary search lower half.
- If  $\text{key}/\text{target}$  is greater than  $a[m]$ , set  $l = m + 1$ , Binary search upper half.
- Otherwise, terminate with answer none.



# Binary Search: Target Exist

l(left) m (middle) r(right)

0	5	11
---	---	----

Key/Target = 22

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
4	7	8	10	14	21	22	36	54	71	85	92

22 > 21

l	m	r
6	8	11

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
4	7	8	10	14	21	22	36	54	71	85	92

l	m	r
6	6	7

22 < 54

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
4	7	8	10	14	21	22	36	54	71	85	92

22 == 22

## Example of Binary Search: Target is not Exist

Search the sorted array **a** for **36**:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	5	7	10	13	13	15	19	19	23	28	28	32	32	37	41	46

1.  $(0+15)/2=7$ ;  $a[7]=19$ ;  
19 is less than 36; search 8..15

2.  $(8+15)/2=11$ ;  $a[11]=32$ ;  
32 is less than 36; search 12..15

3.  $(12+15)/2=13$ ;  $a[13]=37$ ;  
37 is more than 36; search 12..12

4.  $(12+12)/2=12$ ;  $a[12]=32$ ;  
32 is less than 36; search 13..12  
...but  $13 > 12$ , so quit: **36 not found**

## Binary Search Implementation

```
int binary_search (int a[], int target, int n) {
    int first=0, last=n-1, mid;

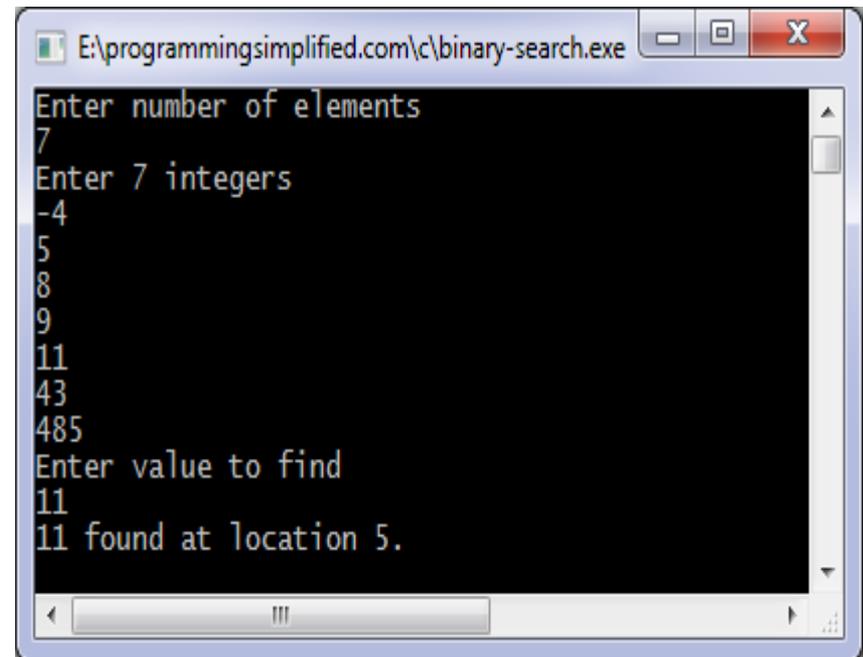
    while (first <= last) { /*there are more elements in the array*/
        mid = (first + last)/2;
        if (target == a[mid])
            return mid; /* target found */
        else if (target < a[mid])
            last = mid - 1;
        else
            first = mid + 1;
    }
    return -1; /* target not found */
}
```

## Example: Searching in Arrays using Binary Search

```
#include <stdio.h>

int main() {
    int c, first, last, middle, n, search, array[100];
    printf("Enter number of elements\n");
    scanf("%d", &n);
    printf("Enter %d integers\n", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
    printf("Enter value to find\n");
    scanf("%d", &search);
    first = 0; last = n - 1; middle = (first+last)/2;
    while (first <= last) {
        if (array[middle] < search)
            first = middle + 1;
        else if (array[middle] == search) {
            printf("%d found at location %d.\n", search, middle+1);
            break; }
        else last = middle - 1;
        middle = (first + last)/2; }
    if (first > last)
        printf("Not found! %d is not present in the list.\n", search);
    return 0; }
```

*/\* a program that reads number of elements, fills them into an array then asks the user to enter a number to check for it in the array using Binary search\*/*



```
E:\programmingsimplified.com\c\binary-search.exe
Enter number of elements
7
Enter 7 integers
-4
5
8
9
11
43
485
Enter value to find
11
11 found at location 5.
```

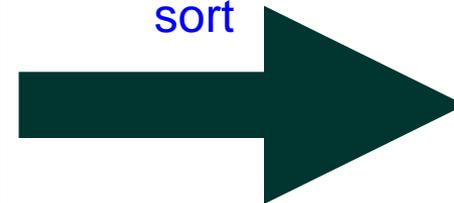
# What is **Sorting** and why do we use it?

- ❑ **Sorting** is the process of arranging the data in a collection or a storage media such that it is in **increasing** or **decreasing** order of **some key**.
- ❑ **Why do we need to sort data?**
  - To arrange names in alphabetical order.
  - Arrange students by their IDs, Grades, Names, ... etc.
  - As a **preliminary** step to search using **Binary search algorithm**.
- ❑ **Sorting speeds up searching in:**
  - Dictionaries
  - Files in a directory
  - Calendar
  - Phone list
- ❑ **Basic steps of sorting involve:**
  - Compare two items
  - Swap the two items or copy one item

Unsorted Array

11
27
10
15
31

sort



Sorted Array

10
11
15
27
31

## Sorting Problem

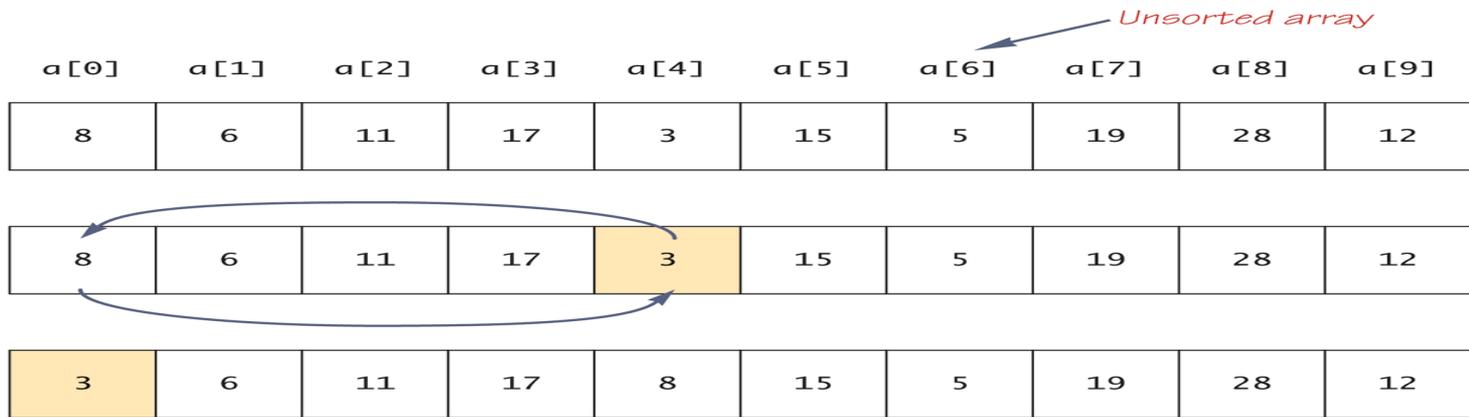
- ❑ Sorting is the ordering of a collection of data based on a certain key.
- ❑ It is a common activity in data management.
- ❑ Many programs execute faster if the data they process are sorted before processing begins.
- ❑ Other programs produce more understandable output if the data displayed is sorted.
- ❑ Many sorting algorithms have been designed.
- ❑ We shall consider only in this course the “**Selection Sort**” method.

# Selection Sort Algorithm

- Scanning the list from the beginning to find (or select) the smallest/**largest** element and swap it with the first element.
  - The rest of the list is then searched for the next smallest/**largest** to swap it with the second element.
  - This process is repeated until the rest of the list reduces to one element, by which time the list is sorted.
- i.e. given an array of length  $n$ , to sort it in **ascending** order:-
- ⇒ Search elements  $0$  through  $n-1$  and select the **smallest/largest**
    - Swap it with the element in location  $0$
  - ⇒ Search elements  $1$  through  $n-1$  and select the **smallest/largest**
    - Swap it with the element in location  $1$
  - ⇒ Search elements  $2$  through  $n-1$  and select the **smallest/largest**
    - Swap it with the element in location  $2$
  - ⇒ Search elements  $3$  through  $n-1$  and select the **smallest/largest**
    - Swap it with the element in location  $3$
  - ⇒ Continue in this fashion until there's nothing left to search.

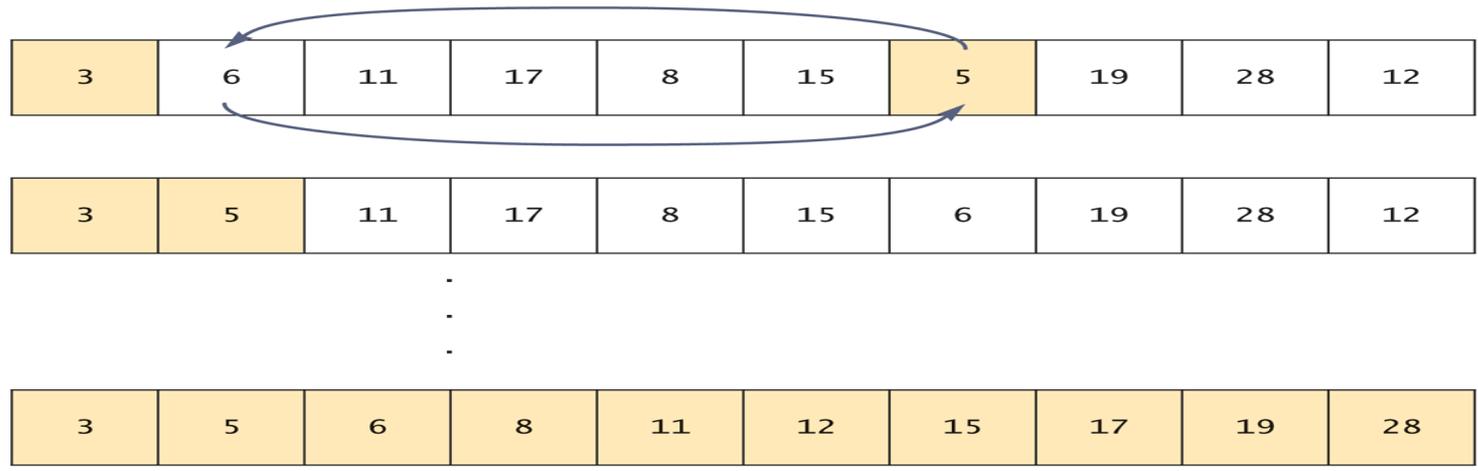
# Steps of Selection Sort

**Display 6.10 Selection Sort**



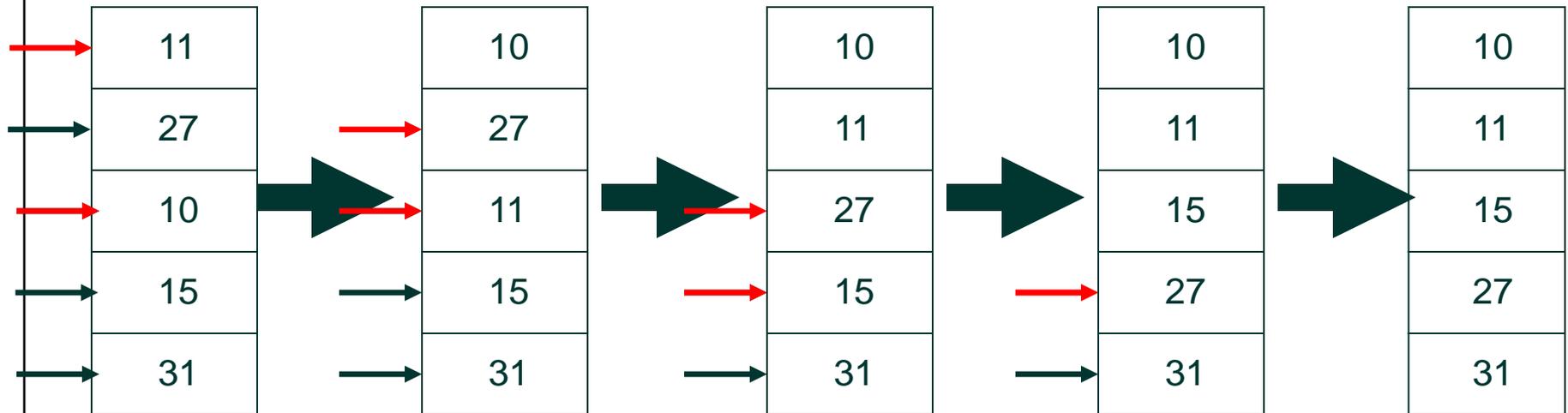
(continued)

**Display 6.10 Selection Sort**



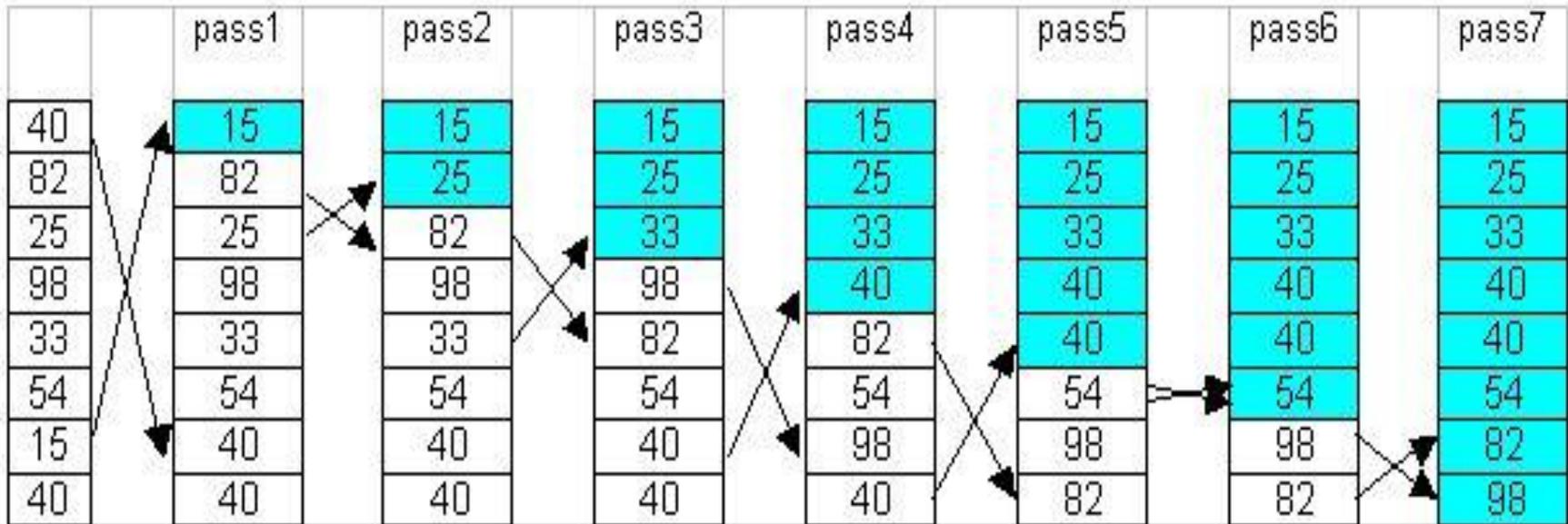
# Selection Sort

- How many time do we need to (compare) thought the array to select the next smallest/**largest** item and how many times do we go through to place that item where it belongs in array?
- Given an array of size 5, let us see how many times are needed.



## Selection Sort

- The selection sort is a combination of **searching and sorting**.
- Selecting the lowest element requires scanning all  $n$  elements (this takes  $n-1$  comparisons) and then swapping it into the first position.
- Finding the next lowest element requires scanning the remaining  $n - 1$  elements and so on.
- In general: to sort an array with  $k$  elements, Selection sort algorithm requires  $k - 1$  passes.



## Outline of Ch. 07 Topics

### □ In the last class, we discussed:

- What is Searching?
- Why do we use a searching algorithm?
- **Linear**/Sequential Search Algorithm,
- Implementing **Linear** search algorithm
- Program Example of Searching in Arrays using Linear Search.
- The main idea of **Binary** Search Algorithm,
- Implementation of Binary Search Algorithm
- Animating Examples of Searching Using **Binary** Search.
- Program Example of Searching in Arrays using **Binary** Search
- What is Sorting?
- Why do we use Sorting?
- Selection Sort Algorithm,

### □ In today's class, we are going to discuss:

- Implementing of Selection Sort Algorithm.
- Declaring and initializing 2D Arrays.
- Indexing and Processing 2D Arrays Elements
- Program Examples of Using 2D Arrays

# Selection Sort Algorithm

- ❑ Find the **index** of the smallest element in the array.
- ❑ Swap the smallest element with the first element.
- ❑ Repeat for the 2<sup>nd</sup>, 3<sup>rd</sup>, ...next smallest element.

[0]	[1]	[2]	[3]
74	45	83	16

`fill` is 0. Find the smallest element in subarray `list[0]` through `list[3]` and swap it with `list[0]`.

[0]	[1]	[2]	[3]
16	45	83	74

`fill` is 1. Find the smallest element in subarray `list[1]` through `list[3]`—no exchange needed.

[0]	[1]	[2]	[3]
16	45	83	74

`fill` is 2. Find the smallest element in subarray `list[2]` through `list[3]` and swap it with `list[2]`.

[0]	[1]	[2]	[3]
16	45	74	83

## Selection Sort Function

```
11. /*
12.  * Sorts the data in array list
13.  * Pre: first n elements of list are defined and n >= 0
14.  */
15. void
16. select_sort(int list[], /* input/output - array being sorted
17.                int n) /* input - number of elements to sort */
18. {
19.     int fill, /* first element in unsorted subarray
20.             temp, /* temporary storage
21.             index_of_min; /* subscript of next smallest element
22.
23.     for (fill = 0; fill < n-1; ++fill) {
24.         /* Find position of smallest element in unsorted subarray */
25.         index_of_min = get_min_range(list, fill, n-1);
26.
27.         /* Exchange elements at fill and index_of_min */
28.         if (fill != index_of_min) {
29.             temp = list[index_of_min];
30.             list[index_of_min] = list[fill];
31.             list[fill] = temp;
32.         }
33.     }
34. }
```

## Smallest Element in Sub-Array

```
/* Find the smallest element in the sub-array list[first]  
* through list[last], where first < last.  
* Return the index of the smallest element in sub-array  
*/
```

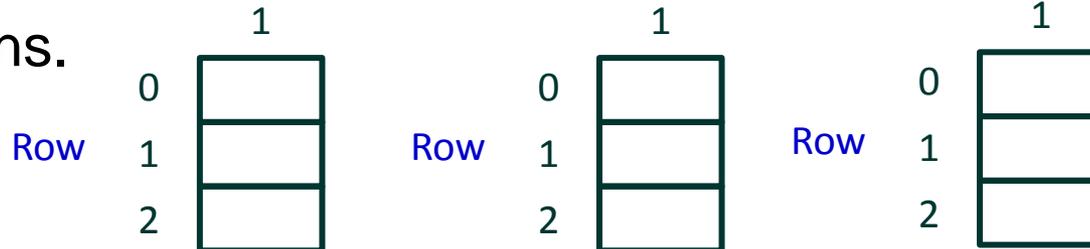
```
int get_min_range(int list[], int first, int last) {  
    int i;  
    int index_min = first;  
    for (i=first+1; i<=last; i++) {  
        if (list[i] < list[index_min]) index_min = i;  
    }  
    return index_min;  
}
```

# Two-Dimensional Arrays

- A **2D** array is a contiguous collection of elements of the same type, that may be viewed as a **table** or a **matrix** consisting of multiple **rows** and multiple **columns**.

		Column			
		0	1	2	3
Row	0				
	1				
	2				

- To store the grades of 30 students in 3 courses,
- By using 1-D array, we need **3** 1-D arrays each of 30 rows and one columns.



- We can use one 2-D array of **30 rows and 5 columns**.

## 2D Array Declaration

- ❑ A 2D array is declared by specifying **the type of element**, **the name of the variable**, followed by **the number of rows** and the number of columns.
- ❑ As with 1D arrays, it is a good practice to declare the row and column sizes as **named constants**:

```
#define ROWS 3
```

```
#define COLS 4
```

```
. . .
```

```
int table[ROWS][COLS];
```

- ❑ Both rows and columns are indexed from zero to rows-1 and columns-1.

	Column			
	0	1	2	3
Row 0				
1				
2				

## Indexing 2D Arrays

- ❑ A 2D array element is indexed by specifying its **row** and **column** indices.
- ❑ C arrays are **row major**, which means that we always refer to the row first.
- ❑ The following statement stores **51** in the cell with row index **1**, and column index **3**:

```
table[1][3] = 51;
```

- ❑ Here is another example:

```
table[2][3] = table[1][3] + 6;
```

- ❑ That means stores **57** in the cell with row index **2**, and column index **3**:

	0	1	2	3
0				
1				51
2				

	0	1	2	3
0				
1				51
2				57

## Initializing 2D Arrays

□ You can declare and initialize a 2D array at the same time.

□ Example:

```
int grades[5][3] = {  
    { 78, 83, 82 },  
    { 90, 88, 94 },  
    { 71, 73, 78 },  
    { 97, 96, 95 },  
    { 89, 93, 90 }  
};
```

□ `int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};`

□ A **two-D** Array can be seen as an array of arrays.

□ Each row is itself a **one-D** array.

## Initializing 2D Arrays

```
int table[][4] = {{1,2,2,5},{3,4,6},{5,6,7,9}};
```

It is ok to omit the number of rows but not the number of columns

	0	1	2	3
0	1	2	2	5
1	3	4	6	0
2	5	6	7	9

If you provide less values than the declared size, **the remaining cells are set to zero.**

↪ */\* Valid declaration\*/*

↪ `int abc[2][2] = {1, 2, 3, 4 }`

↪ */\* Valid declaration\*/*

↪ `int abc[][2] = {1, 2, 3, 4 }`

↪ */\* Invalid declaration – you must specify second dimension\*/*

↪ `int abc[][] = {1, 2, 3, 4 }`

↪ */\* Invalid because of the same reason mentioned above\*/*

↪ `int abc[2][] = {1, 2, 3, 4 }`

## Processing 2D Arrays

- ❑ To process a 2D array, we use a nested loop, and traverse the 2D array either **row-wise** or **column-wise**
- ❑ To process the elements **row-wise**, we use the outer loop for rows and the inner loop for the column.

```
for (i = 0; i < ROWS; i++)
    for(j = 0; j < COLS; j++) {
        /* process table[i][j] */
    }
```

- ❑ To process the elements **column-wise**, we use the outer loop for column and the inner loop for the rows.

```
for (j = 0; j < COLS; j++)
    for(i = 0; i < ROWS; i++) {
        /* process table[i][j] */
    }
```

## Total Sum & Average of a 2D Array

- ❑ To find the sum of all elements in a 2D Array.
- ❑ Use for loop to tell you how many rows in the 2D array.
- ❑ Use for loop to tell you how many columns in the 2D array.
- ❑ Calculate the total sum of all elements in the array.
- ❑ Divide it by the **rows\*columns**.

### Overall Average

```
int sum = 0;
for(int r = 0; r < rows; r++)
    for(int c = 0; c < cols; c++)
        sum = sum + array[r][c];
int avg = sum / (rows*cols);
```

## 2D Array As a Parameter

- ❑ As with 1D arrays, it is possible to declare a function that takes a 2D array as a parameter.
- ❑ The size of the first dimension (**number of rows**) need not be specified in the 2D array parameter.
- ❑ However, the size of the second dimension (**columns**) must be specified as a constant.
- ❑ One solution is to use a named constant defining the maximum number of columns and use additional parameters for the **actual size of the 2D** array:

```
void read_2d(double a[][COLS], int r, int c);
```

## Program to Add 2D Arrays

```
/* A program that reads two 2D arrays, sum the corresponding  
elements of them into a sum 2D array then display it on the  
monitor */
```

```
#include <stdio.h>
```

```
#define ROWS 10      /* maximum number of rows */
```

```
#define COLS 10     /* maximum number of cols */
```

```
void read_2d (double a[][COLS], int r, int c);
```

```
void add_2d (double a[][COLS],  
            double b[][COLS],  
            double s[][COLS], int r, int c);
```

```
void print_2d(double a[][COLS], int r, int c);
```

## Function to **Read** a 2D Array

```
void read_2d (double a[][COLS], int r, int c) {  
    int i, j;  
  
    printf("Enter a table with %d rows\n", r);  
    printf("Each row having %d numbers\n", c);  
  
    for (i=0; i<r; i++)  
        for (j=0; j<c; j++)  
            scanf("%lf", &a[i][j]);  
}
```

## Function to **Add** Two 2D Arrays

```
void add_2d (double a[][COLS],  
             double b[][COLS],  
             double s[][COLS], int r, int c) {  
  
    int i, j;  
    for (i=0; i<r; i++)  
        for (j=0; j<c; j++)  
            s[i][j] = a[i][j] + b[i][j];  
}
```

## Function to **Print** a 2D Array

```
void print_2d(double a[][COLS], int r, int c) {  
    int i, j;  
  
    for (i=0; i<r; i++) {  
        for (j=0; j<c; j++)  
            printf(" %6.1f", a[i][j]);  
        printf("\n");  
    }  
}
```

## Main Function

```
int main(void) {
    double x[ROWS][COLS], y[ROWS][COLS], z[ROWS][COLS];
    int rows, cols;

    printf("Enter number of rows: ");
    scanf("%d", &rows);
    printf("Enter number of columns: ");
    scanf("%d", &cols);

    read_2d(x, rows, cols);      /* read matrix x */
    read_2d(y, rows, cols);      /* read matrix y */
    add_2d(x, y, z, rows, cols); /* Add x + y */

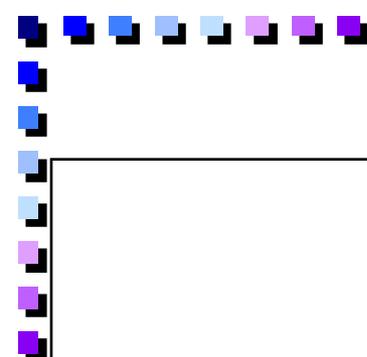
    printf("The sum of two matrices is:\n");
    print_2d(z, rows, cols);     /* Print matrix z */

    return 0;
}
```

## Sample Run . . .

```
Enter number of rows: 2
Enter number of columns: 4
Enter a table with 2 rows
Each row having 4 numbers
3.1 2.7 -3 0.9
2.4 1.1 23 4.5
Enter a table with 2 rows
Each row having 4 numbers
1 2 3 4
5 6 7 8
The sum of two matrices is:
  4.1    4.7    0.0    4.9
  7.4    7.1   30.0   12.5

-----
Process exited with return value 0
Press any key to continue . . .
```



**The End!!**

**Thank you**

**Any Questions?**

