Two Dimensional Arrays

Mirza Mohammad Lutfe Elahi

Outline

- Searching: Linear and Binary Search
- Sorting: Selection Sort
- 2D Arrays

Searching

- Traversing an array to locate a particular item
- It requires the user to specify the target item
- If the target item is found, its index is returned
- If the target item is NOT found, -1 is returned
- Two searching algorithms
  - Linear Search (works with any array)
  - Binary Search (works if the searched array is sorted)

Algorithm – Linear Search

```
1. Initialize a flag (zero) to indicate whether target is found
2. Start at the first array element (at index 0)
3. Repeat while the target is not found and there are more array
   elements
   If the current element matches the target then
       Set the flag to indicate that the target is found
   Else, Advance to the next array element
4. If the target is found then
       Return the target index as the search result
   Else, Return -1 as the search result
```

Linear Search Implementatio

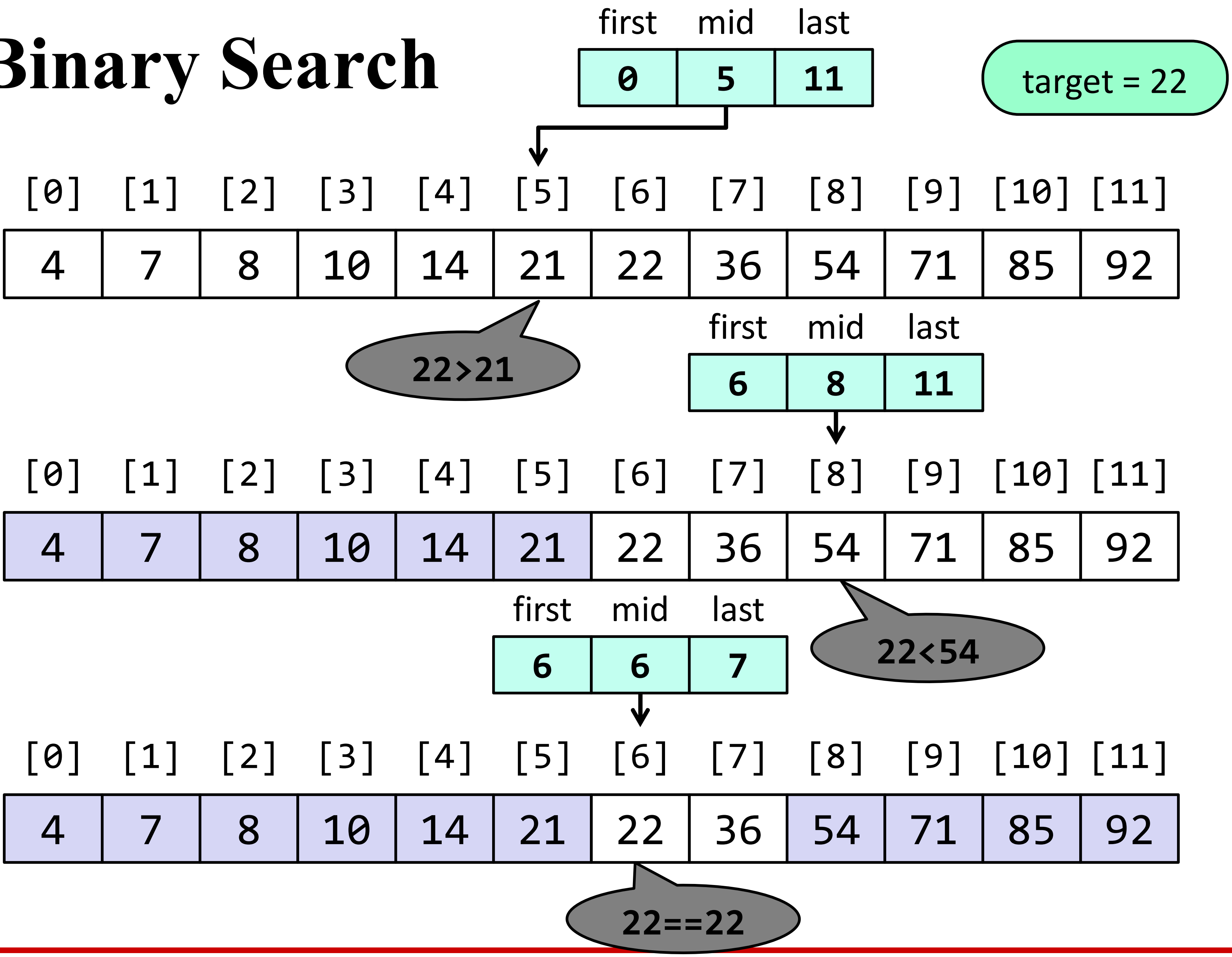
```
/* Search array a[] for target using linear search
 * Returns index of target or -1 if not found */

int linear_search(const int a[], int target, int n) {
  int i = 0, found = 0;

  while(!found && i<n) {
    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.

Binary Search Algorithm

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

Binary Search



Binary Search Implementation

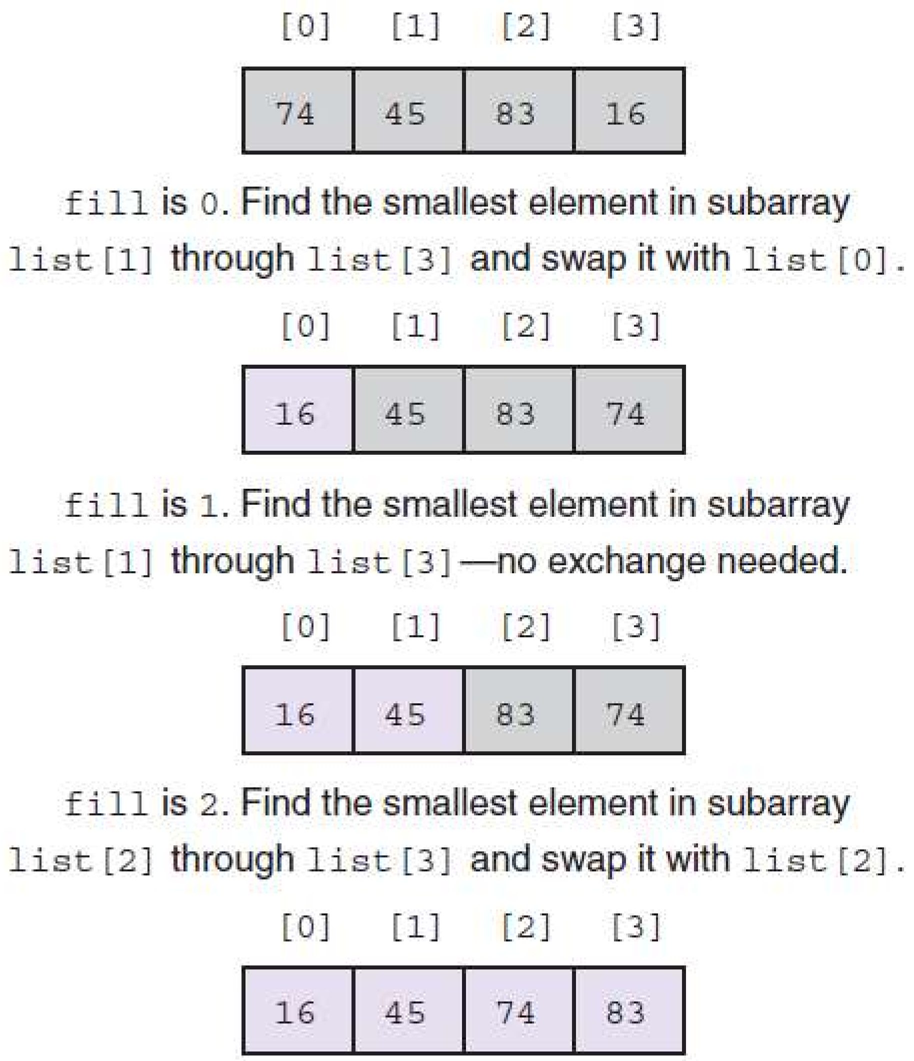
```
int binary_search (int a[], int target, int n) {
  int first = 0, last = n - 1, mid;

  while(first <= last) {   /* more to search */
    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 */
}
```

Sorting

- Sorting is the ordering of a collection of data
- 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 the Selection Sort method.

Algorithm - Selection Sort

- Find the index of the smallest element in the array.
- Swap the smallest element with the first element.
- Repeat for the 2nd, 3rd, …next smallest element.



[0] [1] [2] [3]
74 45 83 16
fill is 0. Find the smallest element in subarray list[1] 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

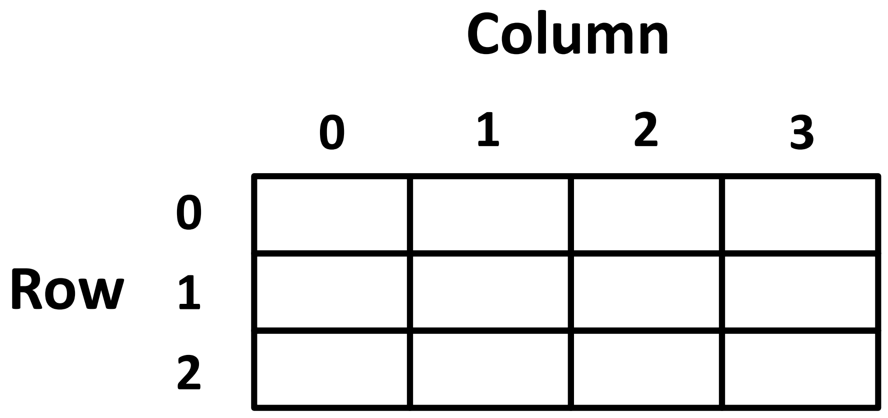
```c
/*
 * Finds the position of the smallest element in the subarray
 * list[first] through list[last].
 * Pre: first < last and elements 0 through last of array list are defined.
 * Post: Returns the subscript k of the smallest element in the subarray;
 *       i.e., list[k] <= list[i] for all i in the subarray
 */
int get_min_range(int list[], int first, int last);


/*
 * Sorts the data in array list
 * Pre: first n elements of list are defined and n >= 0
 */
void
select_sort(int list[],  /* input/output - array being sorted            */
            int n)       /* input - number of elements to sort           */
{
      int fill,          /* index of first element in unsorted subarray  */
          temp,          /* temporary storage                            */
          index_of_min;  /* subscript of next smallest element           */

      for (fill = 0; fill < n-1; ++fill) {
           /* Find position of smallest element in unsorted subarray */
           index_of_min = get_min_range(list, fill, n-1);

           /* Exchange elements at fill and index_of_min */
           if (fill != index_of_min) {
                 temp = list[index_of_min];
                 list[index_of_min] = list[fill];
                 list[fill] = temp;
           }
      }
}
```

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 consisting of multiple rows and multiple columns.

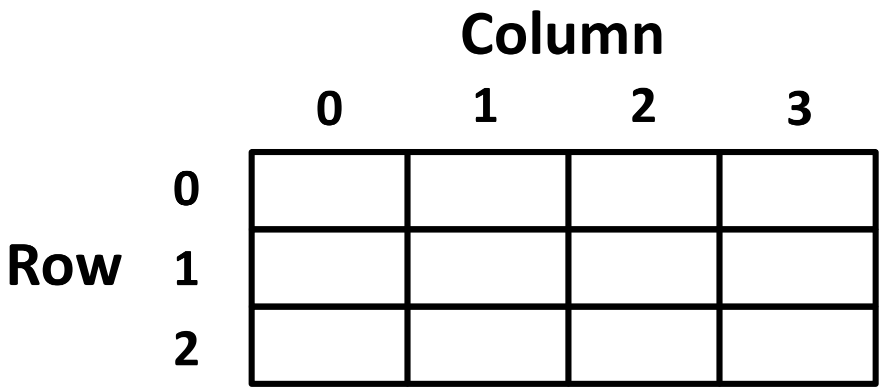


- To store the grades of 30 students in five courses, it is better to use a 2D array than five 1D arrays.

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.

2D Array Declaration

- A 2D array element is indexed by specifying its row and column indices.
- The following statement stores 64 in the cell with row index 1, and column index 3:

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

	0	1	2	3
0				
1				51
2				

- Here is another example:

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

	0	1	2	3
0				
1				51
2				57

Initialization 2D Arrays

- As with 1-D arrays, you can declare and initialize a 2D array at the same time.
- A nested list is used, where each inner list represents a row. For example:

```
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 (NOT a good practice!)

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 write:

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

- To process the elements column-wise, we write:

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

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

```
#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];
}
```

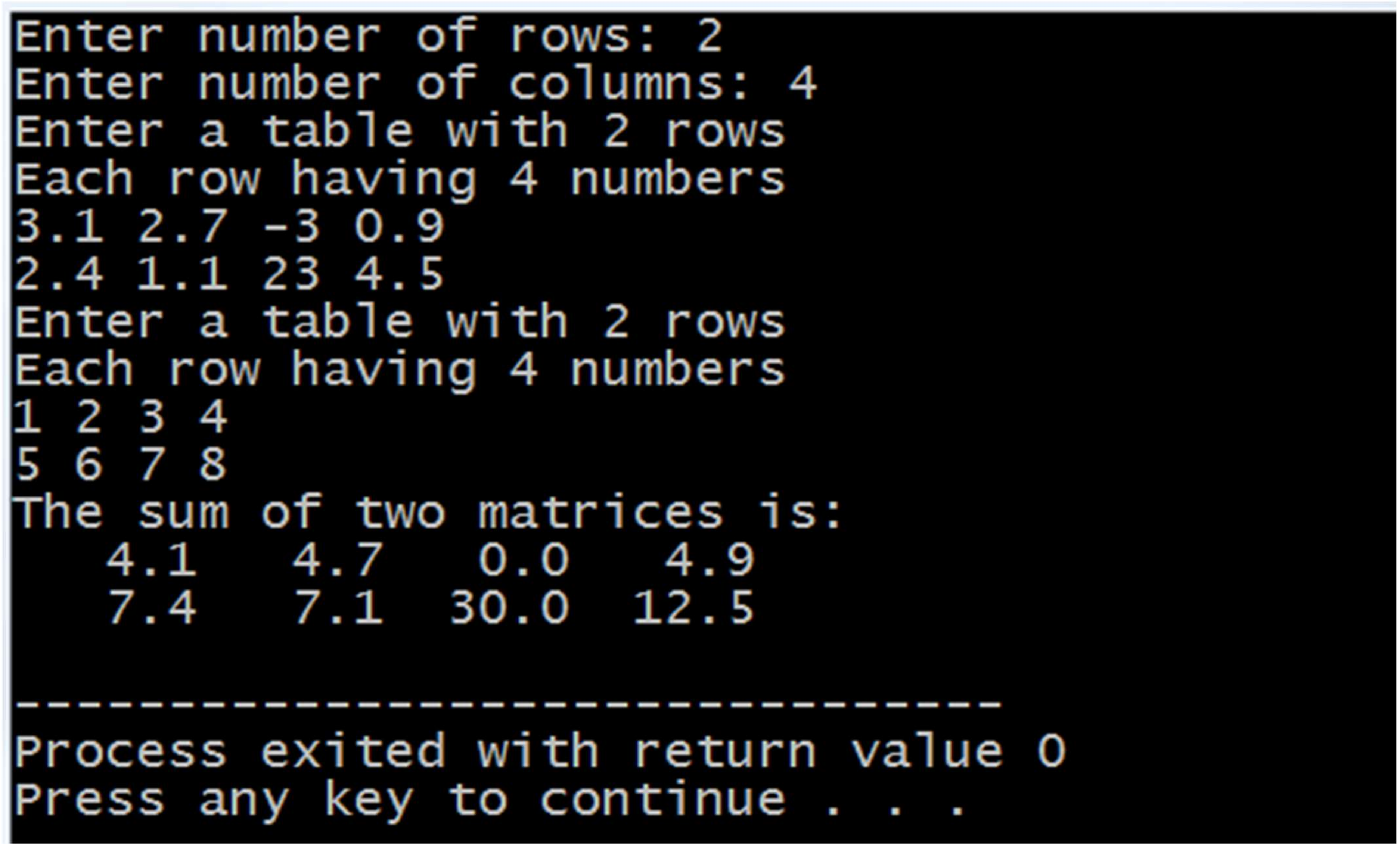
Function to Print a 2D Array

```
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 . . .
```