S-17

Q1.a) What is an array? Explain how one dimensional and two-dimensional array are stored in memory. Give example of each.

An **array** is a collection of data items, all of the same type, accessed using a common name. A one-dimensional **array** is like a list; A two dimensional**array** is like a table; The **C** language places no limits on the number of dimensions in an **array**, though specific implementations may

A single-dimensional array is the simplest form of an array that requires only *one* subscript to access an array element. Like an ordinary variable, an array must have been declared before it is used in the program. The syntax for declaring a single-dimensional array is

data_type array_name [size] ;

For example, an array marks of type int and size five can be declared using this statement.

int marks [5];

Initialization of Single- Dimensional Array: Once an array is declared, the next step is to initialize each array element with a valid and appropriate value. Note that unless an array is initialized, all the array elements contain garbage values. An array can be initialized at the time of its declaration. The syntax for initializing an array at the time of its declaration

data_type array_name [size] ={value 1,value 2,…… ,value n};

The values are assigned to the array elements in the order in which they are listed.

That is, value1, value 2 and value n are assigned to the first, second and nth element of the array, respectively.

For example, the array marks can be initialized while declaring using this statement.

int marks[5]={51,62,43,74,55};

If an array is declared and initialized simultaneously, then specifying its size is optional. For example, the statement int marks [] = {51, 62,43,74,55} is also valid.

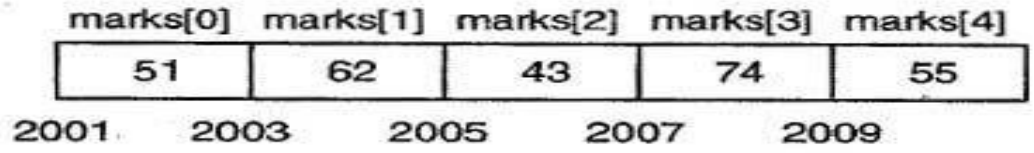
Accessing Single-Dimensional Array Elements: Once an array is declared and initialized, the values stored in the array can be accessed any time. Each individual array element can be accessed using the name of the array and the subscript value. Every element in an array is associated with a unique subscript value, starting from 0 to size-l (where, size refers to the maximum number of elements that can be stored in the array). The syntax for accessing the values stored in a single dimensional array is

array_name [subscript]

For examp1e, the elements of the array marks can be referred to as marks [0], marks [1], Marks [2], marks [3] and marks [4], respectively.

Single-dimensional arrays are always allocated contiguous blocks of memory. This implies that all the elements in an array are always stored next to each other. The memory

representation of the array marks is shown in Figure. As each element is of the type int (that is, 2 bytes long), the array marks occupies ten contiguous bytes in memory and these bytes are reserved in the memory at the compile-time.



Memory Representation of marks

Initialization of 2D Array

There are two ways to initialize a two Dimensional arrays during declaration.

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

OR

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

Although both the above declarations are valid, I recommend you to use the first method as it is more readable, because you can visualize the rows and columns of 2d array in this method.

Things that you must consider while initializing a 2D array

We already know, when we initialize a normal array (or you can say one dimensional array) during declaration, we need not to specify the size of it. However that's not the case with 2D array, you must always specify the second dimension even if you are specifying elements during the declaration. Let's understand this with the help of few examples –

```
/* 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 }
```

How to store user input data into 2D array

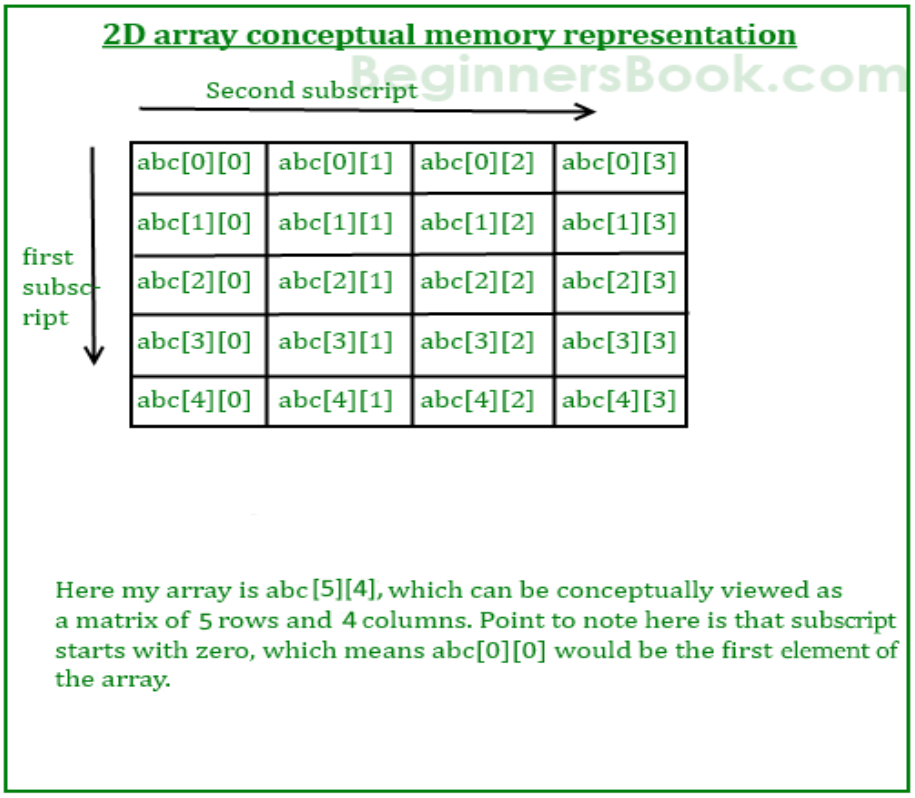
We can calculate how many elements a two dimensional array can have by using this formula:

The array arr[n1][n2] can have n1*n2 elements. The array that we have in the example below is having the dimensions 5 and 4. These dimensions are known as subscripts. So this array has first subscript value as 5 and second subscript value as 4.

So the array abc[5][4] can have 5*4 = 20 elements.
To store the elements entered by user we are using two for loops, one of them is a nested loop. The outer loop runs from 0 to the (first subscript -1) and the inner for loops runs from 0 to the (second subscript -1). This way the the order in which user enters the elements would be abc[0][0], abc[0][1], abc[0][2]…so on.

```
#include<stdio.h>
int main(){
  /* 2D array declaration*/
  int abc[5][4];
  /*Counter variables for the loop*/
  int i, j;
  for(i=0; i<5; i++) {
    for(j=0;j<4;j++) {
      printf("Enter value for abc[%d][%d]:", i, j);
      scanf("%d", &abc[i][j]);
    }
  }
  return 0;
}
```

In above example, I have a 2D array abc of integer type. Conceptually you can visualize the above array like this:



b)Write a program to reverse string without using string handing function.

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

int main() {
  char str[100], temp;
  int i, j = 0;

  printf("\nEnter the string :");
  gets(str);

  i = 0;
  j = strlen(str) - 1;

  while (i < j) {
    temp = str[i];
    str[i] = str[j];
    str[j] = temp;
    i++;
    j--;
  }

  printf("\nReverse string is :%s", str);
  return (0);
}
```

Q.2a) Write a program to add two matrices & store the result in third matrix

```
#include <stdio.h>


int main()

{

  int m, n, c, d, first[10][10], second[10][10], sum[10][10];



  printf("Enter the number of rows and columns of matrix\n");

  scanf("%d%d", &m, &n);

  printf("Enter the elements of first matrix\n");
```

```
  for (c = 0; c < m; c++)
    for (d = 0; d < n; d++)
      scanf("%d", &first[c][d]);

  printf("Enter the elements of second matrix\n");

  for (c = 0; c < m; c++)
    for (d = 0 ; d < n; d++)
      scanf("%d", &second[c][d]);

  printf("Sum of entered matrices:-\n");

  for (c = 0; c < m; c++) {
    for (d = 0 ; d < n; d++) {
      sum[c][d] = first[c][d] + second[c][d];
      printf("%d\t", sum[c][d]);
    }
    printf("\n");
  }

  return 0;
}
```

b) Differentiate between structure & union.

Structure	Union
Keyword struct defines a structure.	Keyword union defines a union.
Example structure declaration:	Example union declaration:

struct s_tag
{
  int ival;
  float fval;
  char *cptr;
}s;
	union u_tag
	{
	  int ival;
	  float fval;
	  char *cptr;
	}u;

Within a structure all members gets memory allocated and members have addresses that increase as the declarators are read left-to-right. That is, the members of a structure all begin at different offsets from the base of the structure. The offset of a particular member corresponds to the order of its declaration; the first member is at offset 0. The total size of a structure is the sum of the size of all the members or more because of appropriate alignment.	For a union compiler allocates the memory for the largest of all members and in a union all members have offset zero from the base, the container is big enough to hold the WIDEST member, and the alignment is appropriate for all of the types in the union. When the storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered.
Within a structure all members gets memory allocated; therefore any member can be retrieved at any time.	While retrieving data from a union the type that is being retrieved must be the type most recently stored. It is the programmer's responsibility to keep track of which type is currently stored in a union; the results are implementation-dependent if something is stored as one type and extracted as another.
One or more members of a structure can be initialized at once.	A union may only be initialized with a value of the type of its first member; thus union u described above (during example declaration) can only be initialized with an integer value.

Q3a) Explain following with syntax & example.
i) getc ( ) ii) fseek ( )

iii) fwrite ( ) iv) fscanf ( )

i) int getc(FILE *stream) gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.

Declaration

Following is the declaration for getc() function.

int getc(FILE *stream)

Parameters

- stream − This is the pointer to a FILE object that identifies the stream on which the operation is to be performed.

Return Value

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

Example

The following example shows the usage of getc() function

```
#include<stdio.h>

int main ()
{
  char c;

  printf("Enter character: ");
  c = getc(stdin);
  printf("Character entered: ");
  putc(c, stdout);

  return(0);
}
```

ii) **fseek(FILE *stream, long int offset, int whence)** sets the file position of the **stream** to the given **offset**.

Declaration

Following is the declaration for fseek() function.

```
int fseek(FILE *stream, long int offset, int whence)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.
- **offset** − This is the number of bytes to offset from whence.

- **whence** − This is the position from where offset is added. It is specified by one of the following constants −

Sr.No.	Constant & Description
1	**SEEK_SET** Beginning of file
2	**SEEK_CUR** Current position of the file pointer
3	**SEEK_END** End of file

Return Value

This function returns zero if successful, or else it returns a non-zero value.

Example

The following example shows the usage of fseek() function.

```
#include <stdio.h>

int main () {
   FILE *fp;

   fp = fopen("file.txt","w+");
   fputs("This is tutorialspoint.com", fp);

   fseek( fp, 7, SEEK_SET );
   fputs(" C Programming Language", fp);
   fclose(fp);

   return(0);
}
```

iii) . **fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)** writes data from the array pointed to, by **ptr** to the given **stream**.

Declaration

Following is the declaration for fwrite() function.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Parameters

- **ptr** − This is the pointer to the array of elements to be written.
- **size** − This is the size in bytes of each element to be written.
- **nmemb** − This is the number of elements, each one with a size of **size** bytes.
- **stream** − This is the pointer to a FILE object that specifies an output stream.

Return Value

This function returns the total number of elements successfully returned as a size_t object, which is an integral data type. If this number differs from the nmemb parameter, it will show an error.

Example

The following example shows the usage of fwrite() function.

```
#include<stdio.h>

int main () {
   FILE *fp;
   char str[] = "This is tutorialspoint.com";

   fp = fopen( "file.txt" , "w" );
   fwrite(str , 1 , sizeof(str) , fp );

   fclose(fp);

   return(0);
}
```

iv) **fscanf(FILE *stream, const char *format, ...)** reads formatted input from a stream.

Declaration

Following is the declaration for fscanf() function.

```
int fscanf(FILE *stream, const char *format, ...)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.

- **format** − This is the C string that contains one or more of the following items − *Whitespace character*, *Non-whitespace character* and *Format specifiers*. A format specifier will be as **[=%[*][width][modifiers]type=]**, which is explained below −

Sr.No.	Argument & Description
1	***** This is an optional starting asterisk indicates that the data is to be read from the stream but ignored, i.e. it is not stored in the corresponding argument.
2	**width** This specifies the maximum number of characters to be read in the current reading operation.
3	**modifiers** Specifies a size different from int (in the case of d, i and n), unsigned int (in the case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument: h : short int (for d, i and n), or unsigned short int (for o, u and x) l : long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g) L : long double (for e, f and g)
4	**type** A character specifying the type of data to be read and how it is expected to be read. See next table.

fscanf type specifiers

type	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
e, E, f, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid	float *

	entries are -732.103 and 7.12e4	
o	Octal Integer:	int *
s	String of characters. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank, newline and tab).	char *
u	Unsigned decimal integer.	unsigned int *
x, X	Hexadecimal Integer	int *

- **additional arguments** − Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter (if any). There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

This function returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

Example

The following example shows the usage of fscanf() function.

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


int main () {
   char str1[10], str2[10], str3[10];
   int year;
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fputs("We are in 2012", fp);

   rewind(fp);
   fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);

   printf("Read String1 |%s|\n", str1 );
   printf("Read String2 |%s|\n", str2 );
   printf("Read String3 |%s|\n", str3 );
   printf("Read Integer |%d|\n", year );
```

```
  fclose(fp);

  return(0);
}
```

b) Write a program to copy abc.txt file into xyz.txt file.

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

void main() {
  FILE *fp1, *fp2;
  char ch;
  clrscr();

  fp1 = fopen("abc.txt", "r");
  fp2 = fopen("xyz.txt", "w");

  while (1) {
    ch = fgetc(fp1);

    if (ch == EOF)
      break;
    else
      putc(ch, fp2);
  }

  printf("File copied Successfully!");
  fclose(fp1);
  fclose(fp2);
}
```

Q4a) Given a text file soript.txt, create another file deleting all the vowels.

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

#define MAX_LENGTH 64


int main()
{
    FILE* f = fopen("soript.txt", "r");
```

```
 char word[MAX_LENGTH];
 int length = strlen(word);
 int i;


 while (fgets(word, MAX_LENGTH, f)) {

      for (i = 0;  i<length; i++){
         if(word[i] == 'a' | word[i] == 'e'|
           word[i] == 'i' | word[i] == 'o'|
           word[i] == 'u' | word[i] == 'y'){
              break;
         }
         else{
            printf("%s ", word);
      }
      }
}
}
```

b) What are the different types of file? Also explain different opening modes of file.

TYPES OF FILES

There are 2 kinds of files in which data can be stored in 2 ways either in characters coded in their ASCII character set or in binary format. They are

1. Text Files.
2. Binary Files

TEXT FILES

A Text file contains only the text information like alphabets ,digits and special symbols. The ASCII code of these characters are stored in these files.It uses only 7 bits allowing 8 bit to be zero.

1.w(write):

This mode opens new file on the disk for writing.If the file exist,disk for writing.If the file exist, then it will be over written without then it will be over written without any confirmation.

SYNTAX:

1 fp=fopen("data.txt","w");

2 "data.txt" is filename

3 "w" is writemode.

2. r (read)

This mode opens an preexisting file for reading.If the file doesn’t Exist then the compiler returns

a NULL to the file pointer
SYNTAX:

```
fp=fopen("data.txt","r");
```

3. w+(read and write)
This mode searches for a file if it is found contents are destroyed If the file doesn't found a new file is created.
SYNTAX:

```
fp=fopen("data.txt","w+");
```

4.a(append)
This mode opens a preexisting file for appending the data.
SYNTAX:

```
fp=fopen("data.txt","a");
```

5.a+(append+read)
the end of the file.
SYNTAX:

```
fp=fopen("data.txt","a+");
```

6.r+ (read +write)
This mode is used for both Reading and writing

BINARY FILES

A binary file is a file that uses all 8 bits of a byte for storing the information .It is the form which can be interpreted and understood by the computer.
The only difference between the text file and binary file is the data contain in text file can be recognized by the word processor while binary file data can't be recognized by a word processor.
1.wb(write)
this opens a binary file in write mode.
SYNTAX:

```
fp=fopen("data.dat","wb");
```

2.rb(read)
this opens a binary file in read mode
SYNTAX:

```
fp=fopen("data.dat","rb");
```

3.ab(append)

this opens a binary file in a Append mode i.e. data can be added at the end of file.

SYNTAX:

fp=fopen(“data.dat”,”ab”);

4.r+b(read+write)

this mode opens preexisting File in read and write mode.

SYNTAX:

fp=fopen(“data.dat”,”r+b”);

5.w+b(write+read)

this mode creates a new file for reading and writing in Binary mode.

SYNTAX:

fp=fopen(“data.dat”,”w+b”);

6.a+b(append+write)

this mode opens a file in append mode i.e. data can be written at the end of file.

SYNTAX:

fp=fopen(“data.dat”,”a+b”);

STREAMS

A Stream refers to the characters read or written to a program. The streams are designed to allow the user to access the files efficiently .A stream is a file or physical device like keyboard, printer and monitor.

The FILE object contains all information about stream like current position, pointer to any buffer, error and EOF(end of file).

C supports a number of functions that have the ability to perform the basic file operations which includes

1. naming a file
2. opening a file
3. reading data from a file
4. writing data into a file
5. closing a file

Opening a file - for creation and edit

Opening a file is performed using the library function in the "**stdio.h**" header file: fopen().

The syntax for opening a file in standard I/O is:

```
ptr = fopen("fileopen","mode")
```

For Example:

```
fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");
```

- Let's suppose the file newprogram.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the

mode 'w'.
The writing mode allows you to create and edit (overwrite) the contents of the file.

- Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode 'rb'.
  The reading mode only allows you to read the file, you cannot write into the file.

5. a) Explain with example, the difference between pointer to structure and pointer within Structure

Q.5b)Write short note on:
i) Static memory allocation.

ii) Dynamic memory allocation.

iii) Array of pointers.

i) When we declare variables, we actually are preparing all the variables that will be used, so that the compiler knows that the variable being used is actually an important part of the program that the user wants and not just a rogue symbol floating around. So, when we declare variables, what the compiler actually does is allocate those variables to their rooms (refer to the hotel analogy earlier). Now, if you see, this is being done before the program executes, you can't allocate variables by this method while the program is executing.

```
// All the variables in below program
// are statically allocated.
void fun()
{
  int a;
}
int main()
{
  int b;
  int c[10]
}
```

ii) Declare the size of an array before you use it. Hence, the array you declared may be insufficient or more than required to hold data. To solve this issue, you can allocate memory dynamically.
Dynamic memory management refers to manual memory management. This allows you to obtain more memory when required and release it when not necessary.
Although C inherently does not have any technique to allocate memory dynamically, there are 4 library functions defined under <stdlib.h> for dynamic memory allocation.

Function	Use of Function

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

C malloc()

The name malloc stands for "memory allocation".

The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

```
ptr = (cast-type*) malloc(byte-size)
```

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr = (int*) malloc(100 * sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

C calloc()

The name calloc stands for "contiguous allocation".

The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

```
ptr = (cast-type*)calloc(n, element-size);
```

This statement will allocate contiguous space in memory for an array of `n` elements. For example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by `ptr`.

Using C malloc() and free()

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

int main()
{
   int num, i, *ptr, sum = 0;

   printf("Enter number of elements: ");
   scanf("%d", &num);

   ptr = (int*) malloc(num * sizeof(int));  //memory allocated using malloc
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
```

```
    printf("Enter elements of array: ");
    for(i = 0; i < num; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

C realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

Syntax of realloc()

```
ptr = realloc(ptr, newsize);
```

Here, ptr is reallocated with size of newsize.

iii) array of pointers :-Just like we can declare an array of int, float or char etc, we can also declare an array of pointers, here is the syntax to do the same.

Syntax: datatype *array_name[size];

Let's take an example:

int *arrop[5];

Here arrop is an array of 5 integer pointers. It means that this array can hold the address of 5 integer variables, or in other words, you can assign 5 pointer variables of type pointer to int to the elements of this array.

The following program demonstrates how to use an array of pointers.

#include<stdio.h>

#define SIZE 10

```
int main()
{
   int *arrop[3];
   int a = 10, b = 20, c = 50, i;
   arrop[0] = &a;
   arrop[1] = &b;
   arrop[2] = &c;
   for(i = 0; i < 3; i++)
   {
      printf("Address = %d\t Value = %d\n", arrop[i], *arrop[i]);
   }

   return 0;
}
```

6. a) What is pointer? state its advantages. Also give details of pointer arithmetic.

A pointer is a variable that contains an address which is a location of another variable in memory. Since a pointer is a variable, its value is also stored in the memory in another location. Suppose we assign the address of quantity to a variable p. The link between the variables p and quantity can be visualized as shown in the figure.

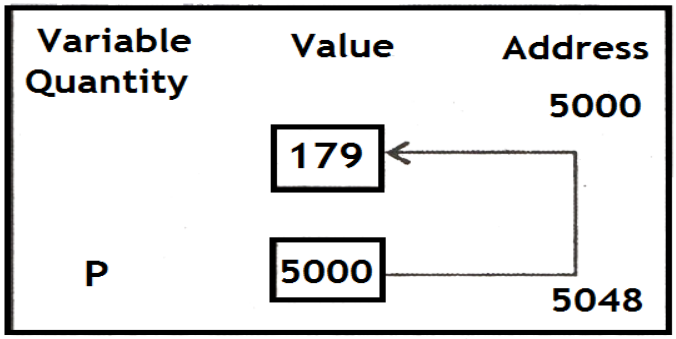


Fig : Pointer as as variable

The address of p is 5048. Since the value of variable p is the address of the variable quantity, we may access the value of a quantity by using the value of p and therefore, we say that the variable p points to the variable quantity. Thus, p gets the name 'pointer'.

Advantages :

1. Pointers reduce the length and complexity of a program.
2. They increase execution speed.
3. A pointer enables us to access a variable that is defined outside the function.
4. Pointers are more efficient in handling the data tables.
5. The use of a pointer array of character strings results in saving of data storage space in memory.

b) Write a program to print greatest numbers in an array using pointers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10],n,i,max;
int *p;
clrscr();
printf("Enter the size of array: ");
scanf("%d",&n);
printf("Enter %d elements in the array:\n",n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("Elements in the array are:\n");
for(i=0;i<n;i++)
printf("%5d",a[i]);
p=&a[0];
max=a[0];
for(i=0;i<n;i++)
{
if(max<=*p)
max=*p;
p++;
}
printf("\nMaximum element in the array is: %d",max);
getch();
}
```

7. a) Write a program in c-to draw five concentric circles and fill the inner most circle with BLUE color.

```
#include<graphics.h>
#include<conio.h>
#include<dos.h>
```

```
main()
{
  int gd = DETECT, gm;
int rad=15,midx,midy;
int I,j=4;

  initgraph(&gd, &gm, "C:\\TC\\BGI");

midx=getmaxx()/2;
midy=getmaxy()/2;
  for(i=1;i<=25;i++)
  {
    setcolor(i);
    circle(midx,midy,rad);
setfillstyle(SOLID_FILL, BLUE);
delay(100);
rad=rad+10;
j++;
}

  getch();
  closegraph();
  return 0;
}
```

b) What is the difference between graphic mode & text mode?

Text mode	Graphics mode
1)In text mode the content of screen is internally represent in term of character	1)In graphics mode the content of screen is internally represented in terms of pixels
2) The screen consist of a uniform grid of character cell.	**2)** the screen is treated as array of pixels
3) In text mode we can display only text	**3)** In graphics mode we can display anything that can be made using pixels.
4) No need to include special header file for using text mode	**4)** Graphics.h has to be included in the program using graphics mode.
5) In text mode the consumption of memory is lower	**5)** In graphics mode consumption of memory is high.
6) The screen manipulation is faster	6) The screen manipulation is slower

c)Explain Initgraph ( ) in detail, using example.

Initialization – initgraph() function

First of all we call the initgraph() function that will initialize the graphics mode on the computer. The method initigraph() has the following prototype.

initgraph(&graphdriver, &graphmode, "pathtodriver");

The method initgraph() initializes the graphics system by loading the graphics driver from disk (or validating a registered driver) then putting the system into graphics mode. The method initgraph() also resets all graphics settings (color, palette, current position, viewport, etc.) to their defaults. The initialization result is set to 0 which can be retrieved by calling graphresult().

The initgraph() method has the following parameters.

graphdriver

This is an integer value that specifies the graphics driver to be used. You can give graphdriver a value using a constant of the graphics_drivers enumeration type which is listed in graphics.h . Normally we use value as "0" (requests auto-detect). Other values are 1 to 10 and description of each enumeration type is listed here.

graphmode

This is an integer value that specifies the initial graphics mode (unless *graphdriver = DETECT). If *graphdriver = DETECT , then initgraph() method sets *graphmode to the highest resolution available for the detected graphics driver. You can give *graphmode a value using a constant of the graphics_modes enumeration type and description of each enumeration type is listed here.

pathtodriver

Specifies the directory path where initgraph() looks for graphics drivers (*.BGI) first.

If they're not there, initgraph() method looks in the current directory.

If pathtodriver is null, the driver files must be in the current directory.

Both *graphdriver and *graphmode parameters must be set to valid graphics_drivers and graphics_mode values or you'll get unpredictable results. (The exception is graphdriver = DETECT.)

Q8a) Explain video Adopter in detail.

Video display adapter

is a special printed circuit board that plugs into the one of the several expansion slots present on the mother board of a computer system.

How are the images either text or graphics, produced on the screen?

This Task is done by the display adapter because it is not possible for the microprocessor to send signal necessary to produce the image on the screen. So in this case display adapter acts as an agent between the video screen and the microprocessor.

Video display adapter done this work with the help of following components:

- VDU memory on which microprocessor writes the information to be produced on the screen.
- Display Adapter circuitry which transfers the information from video memory to screen.

In this way image is produced on the screen. There are various types of Display adapters which are supported by 8086 microprocessor family:

- Monochrome display adapter (MA)
- Hercules display adapter
- Color Graphics adapter (CGA)
- Enhanced Graphics Adapter (EGA)

- Multicolor Graphics Adapter (MCGA)
- Video Graphics Adapter (VGA)
- Super Video Graphics Adapter (SVGA)
- Extended Graphics Adapter (XGA)

- **Monochrome display adaptor (MA)**
  was the first display adapter. This is a simple adapter which can only operates on text mode. It has no capability to operates on the Graphics mode.

- **Hercules Display Adapter**
  is an advance version of the MA. It has all the features of the MA but in additionally it can also operate in Graphics mode.

- **Color Graphics Adapter (CGA)**
  was in demand for several years but for today's perspectives it has very limited qualities. This adapter can also operate on both text and graphics mode like Hercules Display Adapter. In text mode it operates in 25 rows by 80 columns mode with 16 colors. In Graphics mode two resolutions are available medium resolutions graphics mode (320*200) with forr colors available from a palette of 16 and two colors mode (640*200).

- **Multicolor Graphics Adapter (MCGA)**
  This display adapter is an advance version of the EGA. It also includes all the functionality and display modes of MA, CGA and EGA. It additionally includes two more graphics modes one is 640*480 pixel mode in 2 colors and second is 320*200 pixel mode in 256 color.

- **Video Graphics Array (VGA)**
  The VGA supports all the display modes of MA, CGA, EGA and MCGA. In addition it also supports the graphics mode of resolution 640*480 in 16 colors. SVGA and XGA are also includes all the functions and display modes of all already discussed display adapters. SVGA includes two more display modes of resolution 800*600 and 1024*768 and XGA also includes two new modes: 640*480 pixel mode with 65536 colors and 1024*768 pixel mode with 256 color. Any graphical image is also influenced by the Display screen or monitor. Many monitors cannot produce color or graphics, some produce poor quality of images and some are also there to produce good quality of images. Each display adapter supports certain type of monitors. There are various monitor used with the 8086 microprocessor based computer system which are mentioned below:

- Monochrome monitors
- Composite monochrome monitors
- Composite color monitors
- TV sets
- RGB monitors

- VGA monitors
- VGA color monitors

- **Monochrome monitors**
  use to display high resolution text, but these monitors have not any ability to display graphics. These types of monitors can only work with the Monochrome Adapter (MA).

- **Composite monochrome monitors**
  work with the Color Graphics Adapter (CGA). These monitors provide a fairly good one color image. These types of monitors can display text or graphics but not able to generate colors.

- **Composite color monitors**
  produce not only text and graphics but also colors. The demerit of these types of monitors is that these have some serious limitations like a 80-column display is often unreadable and these have very short number of color combinations and images produce through these type of system are not good in quality and resolutions.

- **TV SETS**
  are almost same as the composite color monitor technically. These types of monitor produce very low quality of images. “Text displays must be in 40 column or 20 colors mode to ensure that text is readable”.

- **RGB Monitors**
  are the best monitors among the all types of monitors. These monitors produce high quality of text as well as images of high quality and high resolution with a big number of color combinations. These monitors operate on three input signal to encode the primary colors Red, Green, Blue.

- **VGA Mono Monitors**
  are used to produces output on VGA or MCGA system. VGA Color Monitors are typically used in computers equipped with VGA card.

b) Explain following functions using example.

i) moverel ( )

The header file graphics.h contains **moverel()** function which moves a point from the current position(x_pos1, y_pos1) to a point that is at a relative distance (x, y) from the Current Position and then advances the Current Position by (x, y).

Note : getx and **gety** can be used to find the current position.

Syntax :

```
moverel(int x, int y);


The end position is calculated as :
```

```
x_pos2 = x_pos1 + x;

y_pos2 = y_pos1 + y;
```

ii) moveto ( )

The header file graphics.h contains **moveto()** function which changes the current position to (x, y)

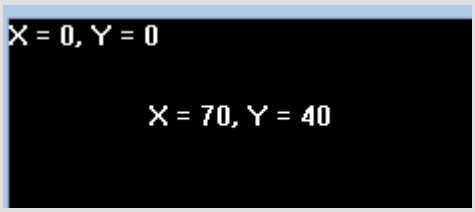
Syntax :

```
void moveto(int x, int y);
```

Examples :

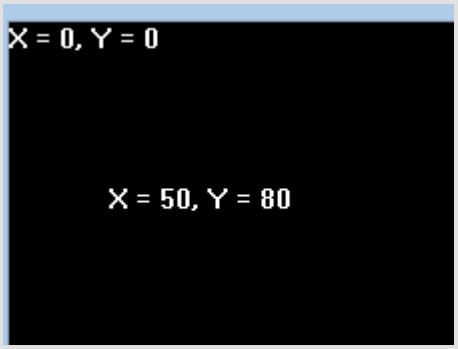
Input : x = 70, y = 40

Output :



Input : x = 50, y = 80

Output :



Q.9a)Compute time and space complexity for bubble sort method.

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

If we have total n elements, then we need to repeat this process for n-1 times.

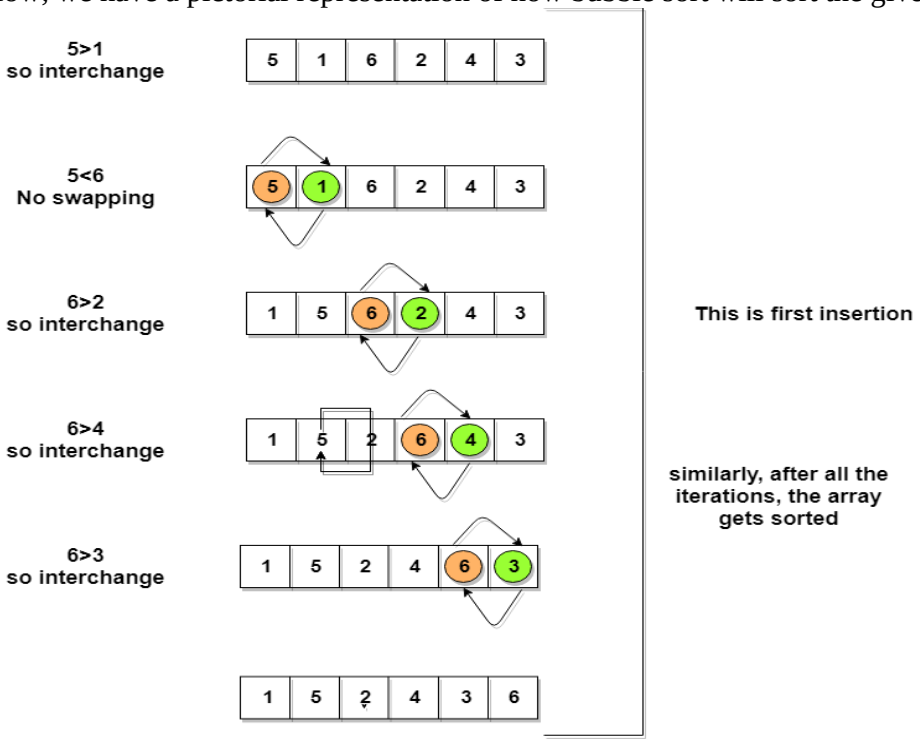
It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Let's consider an array with values {5, 1, 6, 2, 4, 3}
Below, we have a pictorial representation of how bubble sort will sort the given array.



So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.

Similarly after the second iteration, 5 will be at the second last index, and so on.

Complexity Analysis of Bubble Sort

In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,

(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1

Sum = n(n-1)/2

i.e $O(n^2)$

Hence the **time complexity** of Bubble Sort is **$O(n^2)$**.

The main advantage of Bubble Sort is the simplicity of the algorithm.

The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for temp variable.

Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.

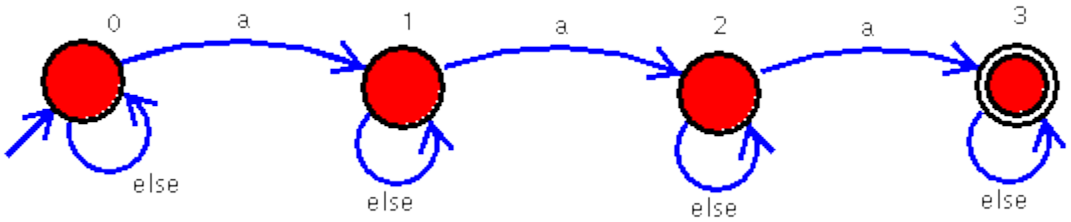
Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: **$O(n^2)$**
- Best Case Time Complexity [Big-omega]: **O(n)**
- Average Time Complexity [Big-theta]: **$O(n^2)$**
- Space Complexity: **O(1)**

b)

Q10a) Explain basic models of computation

1)The finite state machine



A finite state machine is a set of **states** (represented by circles) and **transitions** (represented by arrows connecting states). One state is designated as the **initial state**, via an arrow pointing to a state. And states can be designated as **final states**, via a doubled circle.

Does this accept anteater?

a n t e a t e r

0 --> 1 --> 1 --> 1 --> 1 --> 2 --> 2 --> 2 --> 2

Since 2 is *not* a final state, anteater is rejected.

Does this accept aardvark?

a a r d v a r k

0 --> 1 --> 2 --> 2 --> 2 --> 2 --> 3 --> 3 --> 3

Since 3 *is* a final state, aardvark is accepted.

What types of strings does this machine accept?

Try drawing an automaton for each of the following languages.

1. each string containing the letter sequence abb anywhere within it.
   Examples: abb, abaabb, bbbabbb.
2. each string with an even number of a's and an even number of b's.
   Examples: aaaaaa, abababab, abba.
3. each string of the form aa...abb...b, with the same number of as and bs.
   Examples: ab, aabb, aaabbb.

2) A **Turing machine** is a mathematical model of computation that defines an abstract machine, which manipulates symbols on a strip of tape according to a table of rules.Despite the model's simplicity, given any computer algorithm, a Turing machine capable of simulating that algorithm's logic can be constructed.

The machine operates on an infinite memory tape divided into discrete *cells*.The machine positions its *head* over a cell and "reads" (scans) the symbol there. Then, as per the symbol and its present place in a *finite table* of user-specified instructions, the machine (i) writes a symbol (e.g., a digit or a letter from a finite alphabet) in the cell (some models allowing symbol erasure or no writing),then (ii) either moves the tape one cell left or right (some models allow no motion, some models move the head), then (iii) (as determined by the observed symbol and the machine's place in the table) either proceeds to a subsequent instruction or halts the computation.

3) *Random Access Machine* :-Machine-independent algorithm design depends upon a hypothetical computer called the *Random Access Machine* or RAM. Under this model of computation, we are confronted with a computer where:

- Each ``simple" operation (+, *, -, =, if, call) takes exactly 1 time step.
- Loops and subroutines are *not* considered simple operations. Instead, they are the composition of many single-step operations. It makes no sense for ``sort" to be a single-step operation, since sorting 1,000,000 items will take much longer than sorting 10 items. The time it takes to run through a loop or execute a subprogram depends upon the number of loop iterations or the specific nature of the subprogram.
- Each memory access takes exactly one time step, and we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk, which simplifies the analysis.

Under the RAM model, we measure the run time of an algorithm by counting up the number of steps it takes on a given problem instance. By assuming that our RAM executes a given number of steps per second, the operation count converts easily to the actual run time.

The RAM is a simple model of how computers perform. A common complaint is that it is too simple, that these assumptions make the conclusions and analysis too coarse to believe in practice. For example, multiplying two numbers takes more time than adding two numbers on most processors, which violates the first assumption of the model. Memory access times differ greatly depending on whether data sits in cache or on the disk, thus violating the third assumption. Despite these complaints, the RAM is an excellent model for understanding how an algorithm will perform on a real computer. It strikes a fine balance by capturing the essential behavior of computers while being simple to work with. We use the RAM model because it is useful in practice.

4) Logic circuits:- In computational complexity theory and circuit complexity, a **Boolean circuit** is a mathematical model for digital logic circuits. A formal language can be decided by a family of Boolean circuits, one circuit for each possible input length. Boolean circuits are also used as a formal model for combinational logic in digital electronics.
Boolean circuits are defined in terms of the logic gates they contain. For example, a circuit might contain binary AND and OR gates and unary NOT gates, or be entirely described by binary NAND gates. Each gate corresponds to some Boolean function that takes a fixed number of bits as input and outputs a single bit.

b) Explain the following.

i) Iterative Vs Recursive process.

BASIS FOR COMPARISON	RECURSION	ITERATION
Basic	The statement in a body of function calls the function itself.	Allows the set of instructions to be repeatedly executed.
Format	In recursive function, only termination condition (base case) is specified.	Iteration includes initialization, condition, execution of statement within loop and update (increments and decrements) the control variable.
Termination	A conditional statement is included in the body of the function to force the function to return without recursion call being executed.	The iteration statement is repeatedly executed until a certain condition is reached.
Condition	If the function does not converge to some condition called (base case), it leads to infinite recursion.	If the control condition in the iteration statement never become false, it leads to infinite iteration.
Infinite Repetition	Infinite recursion can crash the system.	Infinite loop uses CPU cycles repeatedly.
Applied	Recursion is always applied to functions.	Iteration is applied to iteration statements or "loops".
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not uses stack.
Overhead	Recursion possesses the overhead of repeated function calls.	No overhead of repeated function call.
Speed	Slow in execution.	Fast in execution.

BASIS FOR COMPARISON	RECURSION	ITERATION
Size of Code	Recursion reduces the size of the code.	Iteration makes the code longer.

ii) Functional programming

Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style. It is a declarative type of programming style. Its main focus is on "what to solve" in contrast to an imperative style where the main focus is "how to solve". It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables.

Concepts of functional programming:

- **Pure functions**
- **Recursion**
- **Referential transparency**
- **Functions are First-Class and can be Higher-Order**
- **Variables are Immutable**

Pure functions: These functions have two main properties. First, they always produce the same output for same arguments irrespective of anything else.

Secondly, they have no side-effects i.e. they do modify any argument or global variables or output something.

Later property is called immutability. The pure functions only result is the value it returns. They are deterministic.

Programs done using functional programming are easy to debug because pure functions have no side effect or hidden I/O. Pure functions also make it easier to write parallel/concurrent applications. When the code is written in this style, a smart compiler can do many things – it can parallelize the instructions, wait to evaluate results when need them, and memorize the results since the results never change as long as the input doesn't change.

example of the pure function:

```
sum(x, y)          // sum is function taking x and y as arguments
   return x + y    // sum is returning sum of x and y without changing them
```

Recursion: There are no "for" or "while" loop in functional languages. Iteration in functional languages is implemented through recursion. Recursive functions repeatedly call themselves, until it reaches the base case.

example of the recursive function:

```
fib(n)
   if (n <= 1)
      return 1;
   else
      return fib(n - 1) + fib(n - 2);
```

Referential transparency: In functional programs variables once defined do not change their value throughout the program. Functional programs do not have assignment statements. If we have to store some value, we define new variables instead. This eliminates any chances of side effects because any variable can be replaced with its actual value at any point of execution. State of any variable is constant at any instant.

example:

```
x = x + 1 // this changes the value assigned to the variable x.
      // So the expression is not referentially transparent.
```

Functions are First-Class and can be Higher-Order: First-class functions are treated as first-class variable. The first class variables can be passed to functions as parameter, can be returned from functions or stored in data structures. Higher order functions are the functions that take other functions as arguments and they can also return functions.

example:

```
show_output(f)          // function show_output is declared taking argument f
                    // which are another function
   f();                 // calling passed function

print_gfg()           // declaring another function
   print("hello gfg";

show_output(print_gfg)  // passing function in another function
```

Variables are Immutable: In functional programming, we can't modify a variable after it's been initialized. We can create new variables – but we can't modify existing variables, and this really helps to maintain state throughout the runtime of a program. Once we create a variable and set its value, we can have full confidence knowing that the value of that variable will never change.

Advantages

1. Pure functions are easier to understand because they don't change any states and depend only on the input given to them. Whatever output they produce is the return value they give. Their function signature gives all the information about them i.e. their return type and their arguments.
2. The ability of functional programming languages to treat functions as values and pass them to functions as parameters make the code more readable and easily understandable.
3. Testing and debugging is easier. Since pure functions take only arguments and produce output, they don't produce any changes don't take input or produce some hidden output. They use immutable values, so it becomes easier to check some problems in programs written uses pure functions.
4. It is used to implement concurrency/parallelism because pure functions don't change variables or any other data outside of it.
5. It adopts lazy evaluation which avoids repeated evaluation because the value is evaluated and stored only when it is needed.

11. a) Create a class rectangle with private data members. Length & breadth and public member function, get data ( ) to get length & breadth, Also a member function area ( ) to print area of rectangle, write main ( ) function that create an object of rectangle and print area of rectangle.

```
#include <iostream.h>
#include<conio.h>
class Rectangle
{
            int length,breadth;
  public:
            void getdata(int,int);
          int area (void)
      {
          return (length*breadth);
      }
};
        void Rectangle::getdata(int a, int b)
      {
            length=a;
            breadth= b;
      }
          void main ()
{
  clrscr();
  Rectangle rect;
  rect.getdata(3,4);
  cout << "Area is : " << rect.area();
  getch();
}
```

b) Write short notes on.
i) Assertion & loop invariant.

Assertions are formal (or semi-formal) facts or statements about the state of your program. The **state** of your program or your computation is like a snapshot of the contents of all variables and other information at some instant of execution.

More precisely an assertion claims something about the state of your computation at the location of the assertion. For example:

```
int trialCount[50];

for (int i = 0; i < 50 ; i++)
  trialCount[i] = 0;
//ASSERT: All trialCount[0..49] == 0
```

The assertion claims that trialCount[i] for all i between 0 and 49 is 0 at the point the assertion is made -- right after the loop. Placing the assertion before the loop or inside the loop body make the assertion false. You want to make and place assertions where they are **TRUE**

An assertion is a claim about the state of the program each time execution reaches a particular point in the program text (or step in the algorithm). We attempt to prove that claim based on other properties that we have proved. After we have proved that claim, it may be used to prove other claims.

Among the claims we make and attempt to verify when reasoning about an algorithm are entry and exit assertions of functions (methods, procedures, etc). These are part of the "contract" of a function. An entry assertion is what we want to be able to assume at the beginning of function execution, or equivalently, what the caller of the function must ensure as its part of the contract. The exit assertion is what the function must ensure as its part, provided the entry assertion is satisfied.

Loop Invariant

In computer science, you could prove it formally with a *loop invariant*, where you state that a desired property is maintained in your loop. Such a proof is broken down into the following parts:

- *Initialization*: It is true (in a limited sense) before the loop runs.
- *Maintenance*: If it's true before an iteration of a loop, it remains true before the next iteration.
- *Termination*: It will terminate in a useful way once it is finished.

Insertion Sort's Invariant

Say, you have some InsertionSort code, where the outer loop goes through the whole array :

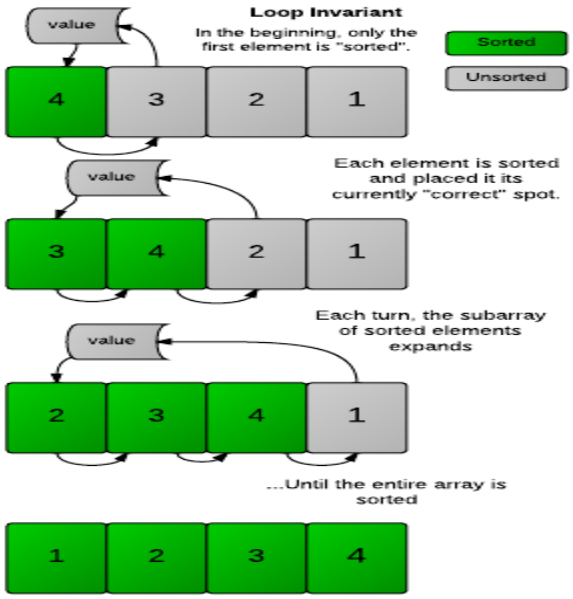
```
for(int i = 1; i < A.length; i++){
//insertion sort code
```

You could then state the following loop invariant:

At the start of every iteration of the outer loop (indexed with), the subarray until consists of the original elements that were there, but in sorted order.

To prove Insertion Sort is correct, you will then demonstrate it for the three stages:

- *Initialization* - The subarray starts with the first element of the array, and it is (obviously) sorted to begin with.
- *Maintenance* - Each iteration of the loop expands the subarray, but keeps the sorted property. An element gets inserted into the array only when it is greater than the element to its left. Since the elements to its left have already been sorted, it means is greater than all the elements to its left, so the array remains sorted. (In *Insertion Sort 2* we saw this by printing the array each time an element was properly inserted.)
- *Termination* - The code will terminate after has reached the last element in the array, which means the sorted subarray has expanded to encompass the entire array. The array is now fully sorted.



You can often use a similar process to demonstrate the correctness of many algorithms. You can see these notes for more information.

Challenge

In the InsertionSort code below, there is an error. Can you fix it? Print the array only once, when it is fully sorted.

Input Format

There will be two lines of input:

- - the size of the array
- - the list of numbers that makes up the array

Constraints

Output Format

Output the numbers in order, space-separated on one line.

Sample Input

```
6

7 4 3 5 6 2
```

Sample Output

```
2 3 4 5 6 7
```

ii) Top down & bottom up design

A top-down design is the decomposition of a system into smaller parts in order to comprehend its compositional sub-systems.

In top-down design, a system's overview is designed, specifying, yet not detailing any first-level subsystems. Then, every subsystem is refined in greater detail, for example, sometimes dividing into many different levels of subsystem, so that the whole specification is decomposed to basic elements.
As soon as these base elements are identified, it is easier to build these elements as computer modules. Once the modules are built, it is effortless to put them together, building the whole system from these individual elements.

top-down design is also known as a stepwise design.

A top-down design is generally a plan made in plain, simple English for the program. It is very important to note that a top-down design must be independent of any programming language. The top-down design must never incorporate references to library functions or syntactic elements specific to a particular language.
That is the reason why top-down designs are written in plain English. The concept driving a top-down design is to break up the task that a program executes into a very few extensive subtasks.

ii) Bottom-Up Design: Any design method in which the most primitive operations are specified first and the combined later into progressively larger units until the whole problem can be solved: the converse of TOP-DOWN DESIGN. For example, a communications program might be built by first writing a routine to fetch a single byte from the communications port and working up from that.

While top-down design is almost mandatory for large collaborative projects, bottom up design can be highly effective for producing 'quick-and-dirty' solutions and rapid prototypes, most often by a single programmer using an interactive, interpreted language such as VISUAL BASIC, LISP

12. a) Explain object-oriented programming features in details.

- Objects
- Classes
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

***Objects*:**

Objects are the basic run-time entities in an object-oriented programming. They may represents a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represents user-defined data such as vector, time and lists. When the program is executed, the object interact by sending message to one another.

***Classes*:**

Objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variable of the type class. Once a class has been defined, we can create a number of objects belonging to that class. A class is a collection of objects of similar type.

Ex. Fruit mango;

***Data Abstraction*:**

Abstractions refer to the act of representing essential features without including background details or explanation. They are commonly known as Abstraction Data Type(ADT).

***Encapsulation*:**

The wrapping up of data and functions into single unit is known as *encapsulation.* Data encapsulation is a striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program.

***Inheritance*:**

Inheritance is the process by which objects of one class acquire the properties of object of another class. The class whose members are inherited is called the Base class and the class that inherits those members is called Derived class. It supports class of hierarchical classification.

The concept of inheritance provides the ideas of reusability. This means we can add essential features to an exciting class without modifying it.

***Polymorphism*:**

Polymorphism is another OOP concept. Polymorphism means the ability to take more than one form. An operation may exhibit different behaviors at different instances.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

1. Creating classes that define object and their behavior,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

b) Differentiate between structured programing and object oriented programming.

	structured programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.

Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.