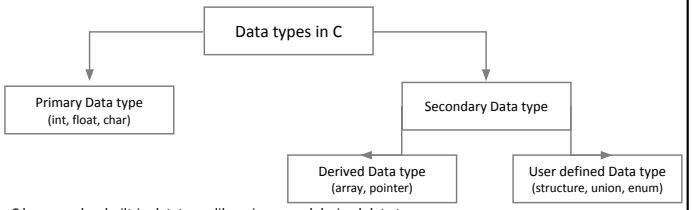
21CSC201J
DATA STRUCTURES AND ALGORITHMS

UNIT-1
Topic : Structures
(Structures - Self-referential structures- Pointers and structures)

Data Types

- Data types are defined as the data storage format that a variable can store a data.



- C language has built-in datatypes like primary and derived data types.
- But, still not all real world problems can be solved using those data types.
- We need custom datatype for different situation.

User Defined Datatype

- We need combination of various datatypes to understand different entity/object.
- **Example-1:**
  - Book Title: Let Us C Datatype: char / string

Author: Yashavant Kanetkar Datatype: char / string
e: 320 Datatype: int
ce: 255.00 Datatype: float

- **Example-2:**

Student Name: ABC Datatype: char / string
Roll_No: 180540107001 Datatype: int
PI: 7.46 Datatype: float
klog: 01 Datatype: int

What is Structure?

- Structure is a collection of logically related data items of different datatypes grouped together under single name.
- Structure is a user defined datatype.
- Structure helps to build a complex datatype which is more meaningful than an array.
- But, an array holds similar datatype record, when structure holds different datatypes records.
- Two fundamental aspects of Structure:
  - Declaration of Structure Variable
  - Accessing of Structure Member

Syntax to Define Structure

- To define a structure, we need to use **struct** keyword.
- This keyword is reserved word in C language. We can only use it for structure and its object declaration.

Syntax

```
struct structure_name
{
    member1_declaration;
    member2_declaration;
    . . .
    memberN_declaration;
};
```

me of custom type

mer is individual

- Members can be normal variables, pointers, arrays or other structures.
- Member names within the particular structure must be distinct from one another.

Example to Define Structure

Example

```
struct student
{
    char name[30]; // Student Name
    int roll_no; // Student Roll No
    float CPI; // Student CPI
    int backlog; // Student Backlog
};
```

- You must terminate structure definition with semicolon ;.
- You cannot assign value to members inside the structure definition, it will cause compilation error.

Example

```
struct student
{
    char name[30] = "ABC"; // Student Name
    . . .
};
```

Create Structure variable

- A data type defines various properties about data stored in memory.
- To use any type we must declare its variable.
- Hence, let us learn how to create our custom structure type objects also known as structure variable.
- In C programming, there are two ways to declare a structure variable:
  1. Along with structure definition
  2. After structure definition

Create Structure Variable – Cont.

1. Declaration along with the structure definition

Syntax

```
struct structure_name
{
    member1_declaration;
    member2_declaration;
    . . .
    memberN_declaration;
} structure_variable;
```

Example

```
struct student
{
    char name[30]; // Student Name
    int roll_no; // Student Roll No
    float CPI; // Student CPI
    int backlog; // Student Backlog
} student1;
```

Create Structure Variable – Cont.

2. Declaration after Structure definition

Syntax

```
struct structure_name structure_variable;
```

Example

```
struct student
{
    char name[30]; // Student Name
    int roll_no; // Student Roll No
    float CPI; // Student CPI
    int backlog; // Student Backlog
};
struct student student1; // Declare structure variable
```

Access Structure member (data)

- Structure is a complex data type, we cannot assign any value directly to it using assignment operator.
- We must assign data to individual structure members separately.
- C supports two operators to access structure members, using a structure variable.
  1. Dot/period operator (.)
  2. Arrow operator (->)

Access Structure member (data) – Cont.

1. Dot/period operator (.)
   - It is known as member access operator. We use dot operator to access members of simple structure variable.

Syntax

```
structure_variable.member_name;
```

Example

```
// Assign CPI of student1
student1.CPI = 7.46;
```

1. Arrow operator (->)
   - In C language it is illegal to access a structure member from a pointer to structure variable using dot operator.
   - We use arrow operator to access structure member from pointer to structure.

Syntax

```
pointer_to_structure->member_name;
```

Example

```
// Student1 is a pointer to student type
student1 -> CPI = 7.46;
```

Write a program to read and display student information using structure.

Program

```
#include <stdio.h>
struct student
{
    char name[40]; // Student name
    int roll; // Student enrollment
    float CPI; // Student mobile number
    int backlog;
};
int main()
{
    struct student student1; // Simple structure variable
    // Input data in structure members using dot operator
    printf("Enter Student Name:");
    scanf("%s", student1.name);
    printf("Enter Student Roll Number:");
    scanf("%d", &student1.roll);
    printf("Enter Student CPI:");
    scanf("%f", &student1.CPI);
    printf("Enter Student Backlog:");
    scanf("%d", &student1.backlog);
    // Display data in structure members using dot operator
    printf("\nStudent using simple structure variable.\n");
    printf("Student name: %s\n", student1.name);
    printf("Student Enrollment: %d\n", student1.roll);
    printf("Student CPI: %f\n", student1.CPI);
    printf("Student Backlog: %i\n", student1.backlog);
}
```

Output

```
Enter Student Name:aaa
Enter Student Roll Number:111
Enter Student CPI:7.89
Enter Student Backlog:0

Student using simple structure variable.
Student name: aaa
Student Enrollment: 111
Student CPI: 7.890000
Student Backlog: 0
```

Write a program to declare time structure and read two different time period and display sum of it.

Program

```
#include<stdio.h>
struct time {
    int hours;
    int minutes;
    int seconds;
};
int main() {
    struct time t1,t2;
    int h, m, s;
    //1st time
    printf ("Enter 1st time.");
    printf ("\nEnter Hours: ");
    scanf ("%d",&t1.hours);
    printf ("Enter Minutes: ");
    scanf ("%d",&t1.minutes);
    printf ("Enter Seconds: ");
    scanf ("%d",&t1.seconds);
    printf ("The Time is
    %d:%d:%d",t1.hours,t1.minutes,t1.seconds);
    //2nd time
    printf ("\n\nEnter the 2nd time.");
    printf ("\nEnter Hours: ");
    scanf ("%d",&t2.hours);
    printf ("Enter Minutes: ");
    scanf ("%d",&t2.minutes);
    printf ("Enter Seconds: ");
    scanf ("%d",&t2.seconds);
    printf ("The Time is
    %d:%d:%d",t2.hours,t2.minutes,t2.seconds);
    h = t1.hours + t2.hours;
    m = t1.minutes + t2.minutes;
    s = t1.seconds + t2.seconds;
    printf ("\nSum of the two time's is
    %d:%d:%d",h,m,s);
    return 0;
}
```

Output

```
Enter 1st time.
Enter Hours: 1
Enter Minutes: 20
Enter Seconds: 20
The Time is 1:20:20

Enter the 2nd time.
Enter Hours: 2
Enter Minutes: 10
Enter Seconds: 10
The Time is 2:10:10
Sum of the two time's is 3:30:30
```

Array of Structure

- It can be defined as the collection of multiple structure variables where each variable contains information about different entities.
- The array of structures in C are used to store information about multiple entities of different data types.

Syntax

```
struct structure_name
{
    member1_declaration;
    member2_declaration;
    ...
    memberN_declaration;
} structure_variable[size];
```

Write a program to read and display N student information using array of structure.

Program

```
#include<stdio.h>
struct student {
    char name[20];
    int rollno;
    float cpi;
};
int main( ) {
    int i,n;
    printf("Enter how many records u want to store : ");
    scanf("%d",&n);
    struct student sarr[n];
    for(i=0; i<n; i++)
    {
        printf("\nEnter %d record : \n",i+1);
        printf("Enter Name : ");
        scanf("%s",sarr[i].name);
        printf("Enter RollNo. : ");
        scanf("%d",&sarr[i].rollno);
        printf("Enter CPI : ");
        scanf("%f",&sarr[i].cpi);
    }
    printf("\n\tName\tRollNo\tMarks\t\n");
    for(i=0; i<n; i++) {
        printf("\t%s\t\t%d\t\t%.2f\t\n", sarr[i].name,
        sarr[i].rollno, sarr[i].cpi);
    }
    return 0;
}
```

Output

```
Enter how many records u want to store : 3

Enter 1 record :
Enter Name : aaa
Enter RollNo. : 111
Enter CPI : 7.89

Enter 2 record :
Enter Name : bbb
Enter RollNo. : 222
Enter CPI : 7.85

Enter 3 record :
Enter Name : ccc
Enter RollNo. : 333
Enter CPI : 8.56

    Name    RollNo  Marks
    aaa     111     7.89
    bbb     222     7.85
    ccc     333     8.56
```

Write a program to declare time structure and read two different time period and display sum of it using function.

Program

```
#include<stdio.h>
struct Time {
    int hours;
    int minutes;
    int seconds;
};
struct Time input(); // function declaration
int main()
{
    struct Time t;
    t=input();
    printf("Hours : Minutes : Seconds\n %d : %d :
    %d",t.hours,t.minutes,t.seconds);
    return 0;
}
struct Time input() // function definition
{
    struct Time tt;
    printf ("Enter Hours: ");
    scanf ("%d",&tt.hours);
    printf ("Enter Minutes: ");
    scanf ("%d",&tt.minutes);
    printf ("Enter Seconds: ");
    scanf ("%d",&tt.seconds);
    return tt; // return structure variable
}
```

Output

```
Enter Hours: 1
Enter Minutes: 20
Enter Seconds: 20
Hours : Minutes : Seconds
1 : 20 : 20
```

Nested Structure

- When a structure contains another structure, it is called nested structure.
- For example, we have two structures named Address and Student. To make Address nested to Student, we have to define Address structure before and outside Student structure and create an object of Address structure inside Student structure.

Syntax

```
struct structure_name1
{
    member1_declaration;
    member2_declaration;
    ...
    memberN_declaration;
};
struct structure_name2
{
    member1_declaration;
    member2_declaration;
    ...
    struct structure1 obj;
};
```

Write a program to read and display student information using nested of structure.

Program

```
#include<stdio.h>
struct Address
{
    char HouseNo[25];
    char City[25];
    char PinCode[25];
};
struct Student
{
    char name[25];
    int roll;
    float cpi;
    struct Address Add;
};
int main()
{
    int i;
    struct Student s;
    printf("\n\tEnter Student Name : ");
    scanf("%s",s.name);
    printf("\n\tEnter Student Roll Number : ");
    scanf("%d",&s.roll);
    printf("\n\tEnter Student CPI : ");
    scanf("%f",&s.cpi);
    printf("\n\tEnter Student House No : ");
    scanf("%s",s.Add.HouseNo);
    printf("\n\tEnter Student City : ");
    scanf("%s",s.Add.City);
    printf("\n\tEnter Student Pincode : ");
    scanf("%s",s.Add.PinCode);
    printf("\nDetails of Students");
    printf("\n\tStudent Name : %s",s.name);
    printf("\n\tStudent Roll Number :
    %d",s.roll);
    printf("\n\tStudent CPI : %f",s.cpi);
    printf("\n\tStudent House No :
    %s",s.Add.HouseNo);
    printf("\n\tStudent City :
    %s",s.Add.City);
    printf("\n\tStudent Pincode :
    %s",s.Add.PinCode);
    return 0;
}
```

Output

```
Details of Students
        Student Name : aaa
        Student Roll Number : 111
        Student CPI : 7.890000
        Student House No : 39
        Student City : rajkot
        Student Pincode : 360001
```

Self referential structures

- Self referential Structures are those structures that contain a reference to data of its same type. i.e in addition to other data a self referential structure contains a pointer to a data that it of the same type as that of the structure. For example: consider the structure node given as follows:

```
struct node
  {
int val;
struct node *next;
};
```

- Here the structure node will contain two types of data an integer val and next which is a pointer a node. Self referential structure is the foundation of other data structures.
- Such kinds of structures are used in different data structures such as to define the nodes of linked lists, trees, etc.

Write a program to read and display student information using nested of structure.

Program

```c
#include <stdio.h>
typedef struct str {
    int mem1;
    int mem2;
    struct str* next;
}str;

int main()
{
    str var1 = { 1, 2, NULL };
    str var2 = { 10, 20, NULL };
    var1.next = &var2;
    str *ptr1 = &var1;
    printf("var2.mem1: %d\nvar2.mem2: %d", ptr1->next->
mem1, ptr1->next->mem2);
   return 0;
}
```

Output

```
var2.mem1: 10
var2.mem2: 20
```

Structure Pointer

- We can define a pointer that points to the structure like any other variable.
- Such pointers are generally called Structure Pointers.
- We can access the members of the structure pointed by the structure pointer using the **(->) arrow operator.**
- Reference/address of structure object is passed as function argument to the definition of function.

Structure using Pointer

Program

```c
#include <stdio.h>
struct student {
        char name[20];
        int rollno;
        float cpi;
};
int main()
{
        struct student *studPtr, stud1;
        studPtr = &stud1;
        printf("Enter Name: ");
        scanf("%s", studPtr->name);
        printf("Enter RollNo: ");
        scanf("%d", &studPtr->rollno);
        printf("Enter CPI: ");
        scanf("%f", &studPtr->cpi);
        printf("\nStudent Details:\n");
        printf("Name: %s\n", studPtr->name);
        printf("RollNo: %d", studPtr->rollno);
        printf("\nCPI: %f", studPtr->cpi);
        return 0;
}
```

Output

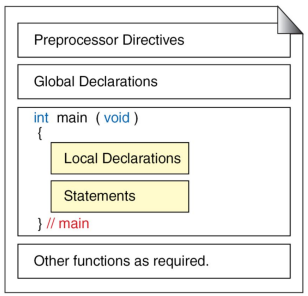
```
Enter Name: ABC
Enter RollNo: 121
Enter CPI: 7.46

Student Details:
Name: ABC
RollNo: 121
CPI: 7.460000
```

21CSC201J
DATA STRUCTURES AND ALGORITHMS
UNIT-1
Topic : Programming in C and Primitive Data Types

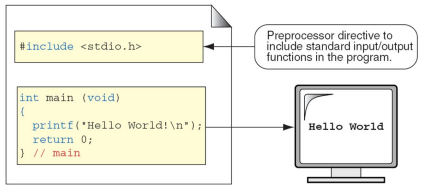
Programming in C - Background

- *C was originally developed in the 1970s, by Dennis Ritchie at Bell Telephone Laboratories, Inc*
- *C is a structured programming language.*
- *It is considered a high-level language because it allows the programmer to concentrate on the problem at hand and not worry about the machine that the program will be using.*
- *That is another reason why it is used by software developers whose applications have to run on many different hardware platforms.*
- *C contains certain additional features that allows it to be used at a lower level , acting as bridge between machine language and the high level languages.*
- *This allows C to be used for system programming as well as for applications programming*

Programming in C – Structure of C Program



Programming in C – Example of C Program



The Greeting Program

Programming in C – Example of C Program

The Greeting Program

```c
/* The greeting program. This program demonstrates
   some of the components of a simple C program.
      Written by:  your name here
      Date:        date program written
*/
#include <stdio.h>

int main (void)
{
// Local Declarations

// Statements

   printf("Hello World!\n");

   return 0;
} // main
```

Programming in C – Comments

Example of Block Comments

```
/* This is a block comment that
   covers two lines.              */

/*
** It is a very common style to put the opening token
** on a line by itself, followed by the documentation
** and then the closing token on a separate line. Some
** programmers also like to put asterisks at the beginning
** of each line to clearly mark the comment.
*/
```

Programming in C – Comments

Example of Line Comments

```
// This is a whole line comment

a = 5;          // This is a partial line comment
```

Programming in C – Identifiers

- *One feature present in all computer languages is the identifier.*
- *Identifiers allow us to name data and other objects in the program. Each identified object in the computer is stored at a unique address.*
- *Rules for Identifiers:*

1. First character must be alphabetic character or underscore.
2. Must consist only of alphabetic characters, digits, or underscores.
3. First 63 characters of an identifier are significant.
4. Cannot duplicate a keyword.

Note

C is a case-sensitive language.

Programming in C – Identifiers

- *Examples of Valid and Invalid identifiers:*

Valid Names		Invalid Name	
a	// Valid but poor style	$sum	// $ is illegal
student_name		2names	// First char digit
_aSystemName		sum-salary	// Contains hyphen
_Bool	// Boolean System id	stdnt Nmbr	// Contains spaces
INT_MIN	// System Defined Value	int	// Keyword

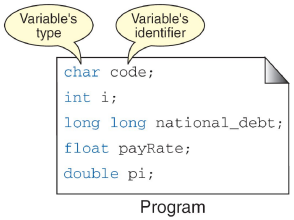
Programming in C – Identifiers (Keywords)

- *Keywords are nothing but system defined identifiers.*
- *Keywords are reserved words of the language.*
- *They have specific meaning in the language and cannot be used by the programmer as variable or constant names*
- *32 Keywords in C Programming*

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Programming in C – Variables

- *Variables are named memory locations that have a type, such as integer or character, which is inherited from their type.*
- *The type determines the values that a variable may contain and the operations that may be used with its values.*

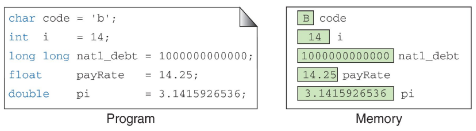


Programming in C – Variable Declarations

Examples of Variable Declarations

```
bool   fact;
short  maxItems;            // Word separator: Capital
long   long national_debt;  // Word separator: underscore
float  payRate;             // Word separator: Capital
double tax;
```

Examples of Variable Initialization



Programming in C – Constants

- *Constants are data values that cannot be changed during the execution of a program.*
- *eg. const double PI = 3.14*
- *Here, PI is a constant. Basically what it means is that, PI and 3.14 is same for this program.*
- *Like variables, constants have a type.*

Integer constants

- *A integer constant is a numeric constant (associated with number) without any fractional or exponential part. There are three types of integer constants in C programming:*

- *decimal constant(base 10)*
- *octal constant(base 8)*
- *hexadecimal constant(base 16)*

Programming in C – Constants

Floating-point constants

- *A floating point constant is a numeric constant that has either a fractional form or an exponent form.*
- *For example: 2.0,0.0000234,-0.22E-5*

Character constants

- *A character constant is a constant which uses single quotation around characters. For example: 'a', 'l', 'm', 'F'*

String constants

- *String constants are the constants which are enclosed in a pair of double-quote marks.*
- *For example: "good" ,"x","Earth is round\n"*

Programming in C – Escape Sequences

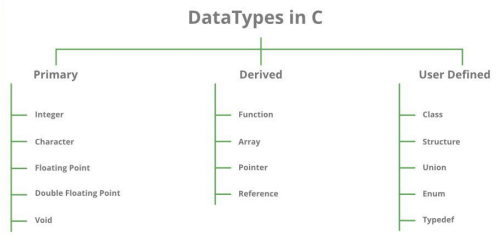
- *Sometimes, it is necessary to use characters which cannot be typed or has special meaning in C programming.*
- *For example: newline(enter), tab, question mark etc. In order to use these characters, escape sequence is used.*

Sequences	Character
\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\0	Null character

Programming in C – Data Types

- *Each variable in C has an associated data type.*
- *It specifies the type of data that the variable can store like integer, character, floating, double, etc.*
- *Each data type requires different amounts of memory and has some specific operations which can be performed over it.*
- *The data type is a collection of data with values having fixed values, meaning as well as its characteristics.*

Programming in C – Data Types

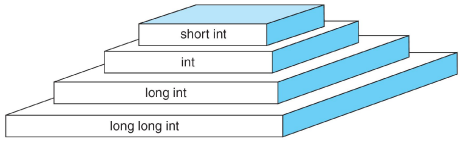


Programming in C – Primitive Data Types

- **Integer Data Type**
- *The integer datatype in C is used to store the whole numbers without decimal values. Octal values, hexadecimal values, and decimal values can be stored in int data type in C.*
- **Range:** -2,147,483,648 to 2,147,483,647
- **Size:** 4 bytes
- **Format Specifier:** %d

Programming in C – Primitive Data Types

- **Integer Types**



Note

sizeof (short) ≤ sizeof (int) ≤ sizeof (long) ≤ sizeof (long long)

Programming in C – Primitive Data Types

Typical Integer Sizes and Values for Signed Integers

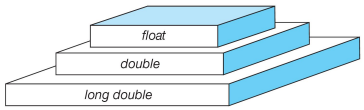
Type	Byte Size	Minimum Value	Maximum Value
short int	2	–32,768	32,767
int	4	–2,147,483,648	2,147,483,647
long int	4	–2,147,483,648	2,147,483,647
long long int	8	–9,223,372,036,854,775,807	9,223,372,036,854,775,806

Programming in C – Primitive Data Types

- **Float Data Type**
- *Float in C is used to store decimal and exponential values. It is used to store decimal numbers (numbers with floating point values) with single precision.*
- **Range:** 1.2E-38 to 3.4E+38
- **Size:** 4 bytes
- **Format Specifier:** %f

Programming in C – Primitive Data Types

- **Floating Point Types**



Note

sizeof (float) ≤ sizeof (double) ≤ sizeof (long double)

Programming in C – Primitive Data Types

- **Void Data Types**
- *The void data type in C is used to specify that no value is present. It does not provide a result value to its caller.*
- *It has no values and no operations. It is used to represent nothing.*
- *Void is used in multiple ways as function return type, function arguments as void etc.*

Programming in C – Sizeof Data Types

- **Size of Data Types in C**
- *The size of the data types in C is dependent on the size of the architecture, so we cannot define the universal size of the data types.*
- *For that, the C language provides the sizeof() operator to check the size of the data types.*

Programming in C – Sizeof Example

```c
// C Program to print size of different data type in C
#include <stdio.h>

int main()
{
    int size_of_int = sizeof(int);
    int size_of_char = sizeof(char);
    int size_of_float = sizeof(float);
    int size_of_double = sizeof(double);

    printf("The size of int data type : %d\n",
       size_of_int);
    printf("The size of char data type : %d\n",
           size_of_char);
    printf("The size of float data type : %d\n",
           size_of_float);
    printf("The size of double data type : %d",
           size_of_double);

    return 0;
}
```

Output

```
The size of int data type : 4
The size of char data type : 1
The size of float data type : 4
The size of double data type : 8
```

21CSC201J
DATA STRUCTURES AND ALGORITHMS

UNIT-1
Topic : MATRIX MULTIPLICATION

Matrix Multiplication

- A matrix is a grid that is used to store data in a structured format. It is often used with a table, where the data is represented in horizontal rows and vertical columns. Matrices are often used in programming languages and are used to represent the data in a graphical structure.
- Once the order of the matrix is declared for the first and second matrix, then the elements (input) for the matrices are needed to be entered by the user. If the order of the matrix is not proportionate to each other, then the error message will be displayed which is implanted by a programmer in the condition statement.
- In C programming matrix multiplications are done by using arrays, functions, pointers. Therefore we are going to discuss an algorithm for Matrix multiplication along with the flowchart, which can be used to write programming code for 3×3 matrix multiplication in a high-level language. This detailed explanation will help you to analyze the working mechanism of matrix multiplication and will help to understand how to write code.

Definition

Definition of Matrix Multiplication

If $A = [a_{ij}]$ is an $m \times n$ matrix and $B = [b_{ij}]$ is an $n \times p$ matrix, then the product AB is an $m \times p$ matrix given by

$$AB = [c_{ij}]$$

where $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + \cdots + a_{in}b_{nj}$.

Definition

The definition of matrix multiplication indicates a *row-by-column* multiplication, where the entry in the *i*th row and *j*th column of the product *AB* is obtained by multiplying the entries in the *i*th row *A* of by the corresponding entries in the *j*th column of *B* and then adding the results.

The general pattern for matrix multiplication is as follows.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{i1} & a_{i2} & a_{i3} & \cdots & a_{in} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1j} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2j} & \cdots & b_{2p} \\ b_{31} & b_{32} & \cdots & b_{3j} & \cdots & b_{3p} \\ \vdots & \vdots & & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nj} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1j} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2j} & \cdots & c_{2p} \\ \vdots & \vdots & & \vdots & & \vdots \\ c_{i1} & c_{i2} & \cdots & c_{ij} & \cdots & c_{ip} \\ \vdots & \vdots & & \vdots & & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mj} & \cdots & c_{mp} \end{bmatrix}$$

$$a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + \cdots + a_{in}b_{nj} = c_{ij}$$

Algorithm

Step 1: Start the Program.
Step 2: Enter the row and column of the first (a) matrix.
Step 3: Enter the row and column of the second (b) matrix
Step 4: Enter the elements of the first (a) matrix.
Step 5: Enter the elements of the second (b) matrix.
Step 6: Print the elements of the first (a) matrix in matrix form.
Step 7: Print the elements of the second (b) matrix in matrix form.
Step 8: Set a loop up to row.

Algorithm

Step 9: Set an inner loop up to the column.
Step 10: Set another inner loop up to the column.
Step 11: Multiply the first (a) and second (b) matrix and store the element in the third matrix (c)
Step 12: Print the final matrix.
Step 13: Stop the Program.

Program for Matrix multiplication

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;
system("cls");
printf("enter the number of row=");
scanf("%d",&r);
printf("enter the number of column=");
scanf("%d",&c);
printf("enter the first matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&a[i][j]);
}
}
```

Program for Matrix multiplication (contd…)

```
printf("enter the second matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&b[i][j]);
}
}
printf("multiply of the matrix=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
mul[i][j]=0;
```

Program for Matrix multiplication (contd…)

```
for(k=0;k<c;k++)
{
mul[i][j]+=a[i][k]*b[k][j];
}
}
}
//for printing result
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
printf("%d\t",mul[i][j]);
}
printf("\n");
}
return 0;
`
```

Output

Output:

```
enter the number of row=3
enter the number of column=3
enter the first matrix element=
1 1 1
2 2 2
3 3 3
enter the second matrix element=
1 1 1
2 2 2
3 3 3
multiply of the matrix=
6 6 6
12 12 12
18 18 18
```

Matrix multiplication using Dynamic Memory Allocation

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

void main()
{
int **a,**b,**c;
//int c[3][3];
int a_r,a_c,b_r,b_c;
int i,j,k;
again:
printf("\nenter rows and columns for matrix one:");
```

```
scanf("%d%d",&a_r,&a_c);printf("\nenter rows and columns for matrix two:");
scanf("%d%d",&b_r,&b_c);
    if(a_c!=b_r )
    {
    printf("\ncan not multiply");
    goto again;
        }/* allocate memory for matrix one */
    a=(int **)malloc(sizeof(int *)*a_r);
            for( i=0;i<a_r;i++)
        {
            a[i]=(int *) malloc(sizeof(int*)*a_c);
                    } /* allocate memory for matrix two */
```

```
b=(int **) malloc(sizeof(int)*b_r);
                for( i=0;i<b_r;i++)
                {
                    b[i]=(int *) malloc(sizeof(int*)*b_c);
                } /* allocate memory for sum matrix*/
    c=(int **) malloc(sizeof(int *)*a_r);
    for( i=0;i<a_r;i++)
    {
    c[i]=(int *) malloc(sizeof(int)*b_c);
    }
printf("\n enter matrix one %d by %d\n",a_r,a_c);
  for(i=0;i<a_r;i++)
  {
    for(j=0;j<a_c;j++)
    {
```

```
scanf("%d",&a[i][j]);
    }
  }
printf("\n enter matrix two %d by %d\n",b_r,b_c);
    for(i=0;i<b_r;i++)
        {
        for(j=0;j<b_c;j++)
            {
            scanf("%d",&b[i][j]);

            }
        } /*initialize product matrix */
```

```
for(i=0;i<a_r;i++)
    {
        for(j=0;j<a_c;j++)
        {
        c[i][j]=0;
        }
    } /* multiply matrix one and matrix two */
        for(i=0;i<a_r;i++)
        {
        for(j=0;j<a_c;j++)
            {
            for(k=0;k<a_r;k++)
                {
                c[i][j]=c[i][j]+a[i][k]*b[k][j];
                }
            }
        } /* display result */
```

```
printf("\n Product of matrix one and two is\n");
    for(i=0;i<a_r;i++)
        {
            for(j=0;j<a_c;j++)
            {
            printf("%d\t",c[i][j]);
            }
        printf("\n");
        }
    }
```

Output

```
enter rows and columns for matrix one: 3
3
enter rows and columns for matrix two:3
3
Enter matrix one 3 by 3
1 2 3
4 5 6
7 8 9
```


Output

```
Enter matrix two 3 by 3
1 2 3
4 5 6
7 8 9
Product of matrix one and two:
30  36  42
66  81  96
102 126 150
```


Introduction to Data Structure and Algorithms

Manoj Kumar Rana
Research Asst. Professor
Dept. of Computing Technologies
SRM Institute of Science and Technology

Data structure and Algorithms

- Algorithm
  - A step-by-step timely **instruction** or **outline** of a computational procedure
- Program
  - **Implementation** of an algorithm by using a programming language
- Data Structure
  - **Organization** of data needed to solve a problem

Data structure and Algorithms (2)

- Object
  - An instance of a **set** of heterogeneous data, defining behavior or character of a real-life entity (e.g. A Car object which contains data like color, size price, etc.)
- Object oriented program
  - A program where objects can **interact** with each other to ease the implementation of the algorithm
- Relation between data structure and object
  - Object is also one data type.
  - Multiple objects can be organized by following a specific data structure

Algorithmic problem

- Infinite set of input instances satisfying the **specification**. For example, a sorted non-increasing sequence of numbers of finite length:
  - 100000, 20000, 1000, 900, 20, 1.

Algorithmic solution

- Algorithm describes the action on input instance
- Infinitely many correct algorithms for a particular problem

Criteria of a good algorithm

- Efficient
  - **Running time**
  - Space taken
- Efficiency is measured as a function of input size
  - Number of bits used to represent input data element
  - Number of data elements

Measuring the running time

- How could we measure the running time of an algorithm?
  - Experimental study
    - Write the **program** that implements the algorithm
    - Run the program with various data sizes and types
    - Use the function **System.CurrentTimeInMillis ()** to get the accurate running time of the program

Limitation of experimental study

- It is required to implement and test the algorithm in order to determine its running time.
- Experiment is always done on limited set of inputs, not done on other inputs.
- This study depends on the **hardware** and **software** used in the experiment. To compare running times of two algorithms, same hardware and software environments must be used.

Beyond experimental study

- We will develop a **general methodology** for analyzing running time of algorithms
  - Uses a high-level description of the algorithm, instead of using one of its implementation
  - Takes into account all possible inputs
  - Independent of hardware and software environment

Pseudo code

- A mixture of natural languages and high-level programming concepts that describes the **main idea** behind a generic implementation of an algorithm or data structure
  - Eg:**ArrayMax (A, n)**

```
Input: Array A and number of elements in A, n
Output: Maximum element in A
CurrMax ☐ A[1]
for i ☐ 2 to n do
  if CurrMax < A[i] then CurrMax ☐ A[i]
return CurrMax
```


Analysis of Algorithm

- Primitive Operation: **Low-level operations** in pseudo code, independent of programming languages
  - data movement (assign)
  - control (branch, sub-routine call, return, etc.)
  - arithmetic and logical operations (addition, comparison, etc.)
- By inspecting pseudo code, we can count number of primitive operations executed by an algorithm

Example: Sorting

Input
Sequence of numbers

$a_1, a_2, a_3, \ldots\ldots\ldots, a_n$

Sort

Output
A permutation of the sequence of numbers

$b_1, b_2, b_3, \ldots\ldots\ldots, b_n$

Correctness (Requirements for the output)
For any given input the algorithm halts with the following output:

- $b_1 < b_2 < b_3 < \ldots\ldots\ldots\ldots < b_n$
- $b_1, b_2, b_3, \ldots\ldots\ldots, b_n$ is the permutation of $a_1, a_2, a_3, \ldots\ldots, a_n$

Running time depends on

- number of elements
- how partially sorted they are
- the algorithm

Insertion Sort

4	7	2	10	7	3	8

1 ☐ i ☐ j ☐ n

Strategy:

- Insert an element in the right position of a set of already sorted elements
- Continue until all the elements are sorted

Input: A[1...n], an array of integers
Output: a permutation of A, such that A[1]≤A[2] ≤A[3] ≤..... ≤ A[n]

```
for j ☐ 2 to n do
key ☐ A[j]
Insert key into the sorted sequence A[1, j-1]
i ☐ j-1
while i > 0 and A[i] > key
      do A[i+1] ☐ A[i]
         i--
A[i+1] ☐ key
```


Analysis of Insertion Sort

primitive operations	cost	times
for j ☐ 2 to n do	c_1	n
key ☐ A[j]	c_2	n-1
Insert key into the sorted sequence A[1, j-1]	0	n-1
i ☐ j-1	c_3	n-1
while i > 0 and A[i] > key	c_4	$\Sigma_{j=2}^{n} t_j$
do A[i+1] ☐ A[i]	c_5	$\Sigma_{j=2}^{n} (t_j - 1)$
i--	c_6	$\Sigma_{j=2}^{n} (t_j - 1)$
A[i+1] ☐ key	c_7	n-1

Total time = $n(c_1 + c_2 + c_3 - c_5 - c_6 + c_7) + \Sigma_{j=2}^{n} t_j (c_4 + c_5 + c_6) - (c_2 + c_3 + c_7)$

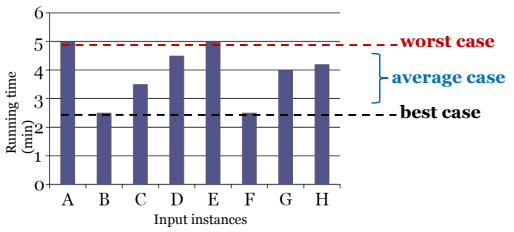
Best, worst and Average cases

Total time = $n(c_1 + c_2 + c_3 - c_5 - c_6 + c_7) + \Sigma_{j=2}^{n} t_j (c_4 + c_5 + c_6) - (c_2 + c_3 + c_7)$

- **Best case**: elements already sorted, $t_j = 1$, running time = f(n) i.e. linear time.
- **Worst case**: elements are in decreasing order, $t_j = j$, running time = $f(n^2)$ i.e. quadratic time
- **Average case**: $t_j = j/2$, running time = $f(n^2)$, i.e. quadratic time

Best/worst/average case

For a specific input size n, investigate running times of different instances

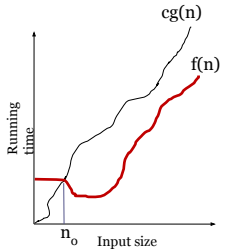


Asymptotic Analysis

- Goal: simplifying analysis of running time by getting rid of "details" which may be affected by specific implementation and hardware
  - like, rounding 10,000.01 ≈ 10,000
  - $6n^2 \approx n^2$
- How the running time of an algorithm increases with the size of the input *in a limit*
  - asymptotically more efficient algorithms are best for all but small inputs

Asymptotic notation

- The "big-oh" *O*-notation
  - asymptotic upper bound
  - f(n) = *O*(g(n)), if there exists constants c and n_0, s.t. f(n) ≤ cg(n) for n ≥ n_0
  - f(n) and g(n) are functions over non-negative integers
- Used for worst-case analysis



Asymptotic notation

- n^2 is not *O*(n) because there is no c and n_0 s.t.:
  $n^2 \le cn$ for $n \ge n_0$
  - no matter how large a c is chosen there exist an n big enough s.t. $n^2 > cn$
- Simple rule: drop lower order terms and constant factors
  - 40 n log n is *O*(n log n)
  - 3n-7 is *O*(n)
  - $8n^2 \log n + n^2 + 7n + 6$ is $O(n^2 \log n)$
- Note: although (40 n^2) is *O*(n^5), it is expected that such an approximation be as small an order as possible

Asymptotic analysis of running time

- The *O*-notation expresses number of primitive operations executed as function of input size
- Comparing asymptotic running times
  - an algorithm running in *O*(n) time is better than an algorithm running in *O*(n^2) time
  - Similarly *O*(log n) is better than *O*(n)
  - hierarchy: $\log n < n < n^2 < n^3 < 2^n$
- Caution! Beware of very large constant factor. An algorithm running in 10000000 n time is still *O*(n), but might be less efficient than an algorithm running in time $2n^2$ time, which is *O*(n^2)

Example of Asymptotic Analysis

```
Algorithm prefixAverages1 (X)
Input: An n element array X of integer numbers
Output: An n element array A of integer numbers
  where A[i] is the average of elements X[1], X[2],
  ....., X[i]
for i ☐ 1 to n do                    } n iterations
  a ☐ 0
  for j ☐ 1 to i do                  } i iterations
    a ☐ a + X[j]                       with i = 1, 2,
  A[i] ☐ a/i                           3, ....., n
return A
```

Analysis: running time is *O*(n^2)

A better Algorithm

```
Algorithm prefixAverages2 (X)
Input: An n element array X of integer numbers
Output: An n element array A of integer numbers
  where A[i] is the average of elements X[1], X[2], ....., X[i]
m ☐ 0
for i ☐ 1 to n do
  m ☐ m + X[i]
  A[i] ☐ s/i
return A
```

Analysis: running time is O(n)

Asymptotic Notation (terminology)

- Special classes of algorithms:
  - Logarithmic: *O*(log n)
  - Linear: *O*(n)
  - Quadratic: *O*(n^2)
  - Polynomial: *O*(n^k), k ≥ 1
  - Exponential: *O*(a^n), a > 1
- Abuse of notation: f(n) = *O*(g(n)) actually means f(n) ε *O*(g(n))

Exercises

1. $f(n) = \sum_{i=0}^{k} a_i n^i, \quad a_k > 0$
   $f(n) \in O(?)$
2. $f(n) = f(n-1) + f(n-2), \quad f(0) = f(1) = 1, \quad f(n) \in O(?)$
3. *f(n) = Log n!, f(n) ε O(?)*

21CSC201J
DATA STRUCTURES AND ALGORITHMS
UNIT-1
Topic : Dynamic Memory Allocation

Dynamic Memory Allocation

Problem with Arrays

- Amount of data cannot be predicted beforehand
- Number of data items keeps changing during program execution

Solution:

- Find the maximum possible value of N and allocate an array of N elements
- Wasteful of memory space, as N may be much smaller in some executions

Dynamic Memory Allocation

Better Solution

- Dynamic memory allocation
  - Know how much memory is needed after the program is run
    - Example: ask the user to enter from keyboard
  - Dynamically allocate only the amount of memory needed

Dynamic Memory Allocation

Memory Allocation Functions

- malloc
  - Allocates requested number of bytes and returns a pointer to the first byte of the allocated space
- calloc
  - Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
- free
  - Frees previously allocated space.
- realloc
  - Modifies the size of previously allocated space.
- We will only do malloc and free

Dynamic Memory Allocation

Allocating a Block of Memory

- A block of memory can be allocated using the function malloc
  - Reserves a block of memory of specified size and returns a pointer of type void
  - The return pointer can be type-casted to any pointer type
- General format:

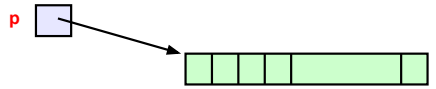
```
type *p;
p =  (type *) malloc (byte_size);
```

Dynamic Memory Allocation

Example

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

- A memory space equivalent to 100 times the size of an int bytes is reserved
- The address of the first byte of the allocated memory is assigned to the pointer p of type int



Dynamic Memory Allocation

Example

- cptr = (char *) malloc (20);

  Allocates 20 bytes of space for the pointer cptr of type char

- sptr = (struct stud *) malloc(10*sizeof(struct stud));

  Allocates space for a structure array of 10 elements. sptr points to a structure element of type struct stud

Dynamic Memory Allocation

- malloc always allocates a block of contiguous bytes
  - The allocation can fail if sufficient contiguous memory space is not available
  - If it fails, malloc returns NULL

```
if  ((p = (int *) malloc(100 * sizeof(int))) == NULL)
 {
      printf ("\n Memory cannot be allocated");
      exit();
 }
```

Dynamic Memory Allocation

Allocating a Block of Memory

calloc() method

- "calloc" or **"contiguous allocation"** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type.

Syntax :

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

Example:

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

Dynamic Memory Allocation

Allocating a Block of Memory
free() method

- "free" method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own.
- It helps to reduce wastage of memory by freeing it.

Syntax of free()

```
free(ptr);
```

Dynamic Memory Allocation

Allocating a Block of Memory
realloc() method

- "realloc" or "re-allocation" method in C is used to dynamically change the memory allocation of a previously allocated memory.
- If the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory.
- re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax :

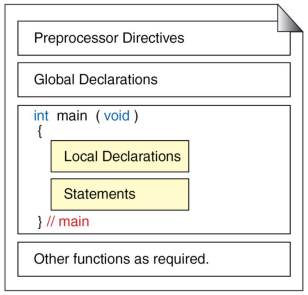
```
ptr = realloc(ptr, newSize);
```

Dynamic Memory Allocation

Allocating a Block of Memory
realloc() method Example:

```
int* ptr;
    int n, i;
    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);
    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));
  // Get the new size for the array
        n = 10;
        printf("\n\nEnter the new size of the array: %d\n", n);
        // Dynamically re-allocate memory using realloc()
        ptr = realloc(ptr, n * sizeof(int));
        // Memory has been successfully allocated
```

Dynamic Memory Allocation



21CSC201J
DATA STRUCTURES AND ALGORITHMS
UNIT-1

MATHEMATICAL NOTATIONS

1. Floor and Ceiling Functions

If x is a real number, then it means that x lies between two integers which are called the floor and ceiling of x. i.e.

```
|_x_| is called the floor of x. It is the greatest integer that is not greater than x.
|  x  | is called the ceiling of x. It is the smallest integer that is not less than x.
```

If x is itself an integer, then |_x_| = | x |, otherwise |_x_| + 1 = | x | E.g.

```
|_3.14_| = 3, |_-8.5_| = -9, |_7_| = 7

|  3.14  |= 4, |  -8.5  | = -8, |   7   |= 7
```

2. Remainder Function (Modular Arithmetic)

If k is any integer and M is a positive integer, then:

```
k (mod M)
```

gives the integer remainder when k is divided by M.
E.g.

```
25(mod 7) = 4
25(mod 5) = 0
```

3. Integer and Absolute Value Functions

If x is a real number, then integer function INT(x) will convert x into integer and the fractional part is removed.
E.g.

```
INT (3.14) = 3
INT (-8.5) = -8
```

The absolute function ABS(x) or | x | gives the absolute value of x i.e. it gives the positive value of x even if x is negative.
E.g.

```
ABS(-15) = 15 or ABS | -15| = 15
ABS(7) = 7 or ABS | 7 | = 7
ABS(-3.33) = 3.33 or ABS | -3.33 | = 3.33
```

4. Summation Symbol (Sums)

The symbol which is used to denote summation is a Greek letter Sigma ?.
Let a1, a2, a3, ... , an be a sequence of numbers. Then the sum a1 + a2 + a3 + ... + an will be written as:

```
n
? aj
j=1
```

where j is called the dummy index or dummy variable.
E.g.

```
n
? j = 1 + 2 + 3 +…..+ n
j=1
```

5. Factorial Function

n! denotes the product of the positive integers from 1 to n. n! is read as 'n factorial', i.e.

```
n! = 1 * 2 * 3 * ….. * (n-2) * (n-1) * n
```

E.g.

```
4! = 1 * 2 * 3 * 4 = 24
5! = 5 * 4! = 120
```

6. Permutations

Let we have a set of n elements. A permutation of this set means the arrangement of the elements of the set in some order.
E.g.

Suppose the set contains a, b and c. The various permutations of these elements can be: abc, acb, bac, bca, cab, cba.

If there are n elements in the set then there will be n! permutations of those elements. It means if the set has 3 elements then there will be 3! = 1 * 2 * 3 = 6 permutations of the elements.

7. Exponents and Logarithms

Exponent means how many times a number is multiplied by itself. If m is a positive integer, then:

```
am = a * a * a * ….. * a (m times)
and
a-m = 1 / am
E.g.
24 = 2 * 2 * 2 * 2 = 16
2-4 = 1 / 24 = 1 / 16
```

The concept of logarithms is related to exponents. If b is a positive number, then the logarithm of any positive number x to the base b is written as logbx. It represents the exponent to which b should be raised to get x i.e. y = logbx and by = x
E.g.

```
log28 = 3, since 23=8

log100.001 = - 3, since 10-3
= 0.001
logb1 = 0, since b0
= 1
logbb = 1, since b1
= b
```

ASYMPTOTIC NOTATIONS BIG O, OMEGA

Asymptotic Analysis

- The time required by an algorithm falls under three types −
  - **Best Case** − Minimum time required for program execution.
  - **Average Case** − Average time required for program execution.
  - **Worst Case** − Maximum time required for program execution.

Asymptotic Analysis (Cont..)

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance.
- Derive the best case, average case, and worst case scenario of an algorithm.
- Asymptotic analysis is input bound.
  - Specify the behaviour of the algorithm when the input size increases
- Theory of approximation.
- Asymptote of a curve is a line that closely approximates a curve but does not touch the curve at any point of time.

Asymptotic notations

- Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.
- ***Asymptotic order*** is concerned with how the running time of an algorithm increases with the size of the input, if input increases from small value to large values

1. Big-Oh notation (O)
2. Big-Omega notation (Ω)
3. Theta notation (θ)
4. Little-oh notation (o)
5. Little-omega notation (ω)

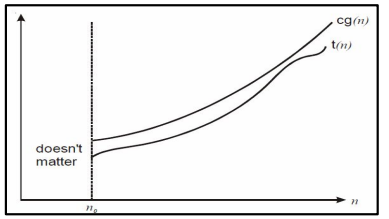
Big-Oh Notation (O)

- Big-oh notation is used to define the worst-case running time of an algorithm and concerned with large values of n.
- ***Definition:*** A function $t(n)$ is said to be in O(g(n)), denoted as t(n) ϵ O(g(n)), if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n. i.e., if there exist some positive constant c and some non-negative integer $n0$ such that

  $t(n) \leq cg(n)$ for all $n \geq n0$

- **O(*g*(*n*)):** Class of functions $t(n)$ that grow no faster than $g(n)$.
- Big-oh puts asymptotic ***upper bound*** on a function.

Big-Oh Notation (O)



Big-Oh Notation (O)

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots\ldots\ldots < 2^n < 3^n < \ldots. < n^n$

- Let t(n) = 2n + 3 upper bound
  2n + 3 ≤ _____??

  2n + 3 ≤ 5n n ≥ 1
  here c = 5 and g(n) = n

  t(n) = O(n)

  2n + 3 ≤ $5n^2$ n ≥ 1
  here c = 5 and g(n) = n^2

  t(n) = O(n^2)

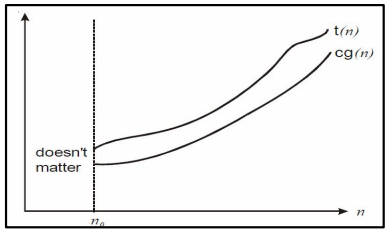
Big-Omega notation (Ω)

- This notation is used to describe the best case running time of algorithms and concerned with large values of *n*.
- ***Definition:*** A function *t*(*n*) is said to be in Ω(*g*(*n*)), denoted as *t*(*n*) ϵ Ω(*g*(*n*)), if *t*(*n*) is bounded below by some positive constant multiple of *g*(*n*) for all large *n*. i.e., there exist some positive constant *c* and some non-negative integer *n*0. Such that

  $t(n) \geq cg(n)$ for all $n \geq n0$

- It represents the ***lower bound*** of the resources required to solve a problem.

Big-Omega notation (Ω)



Big-Omega notation (Ω)

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots\ldots\ldots < 2^n < 3^n < \ldots. < n^n$

- Let t(n) = 2n + 3 lower bound
  2n + 3 ≥ _____??

  2n + 3 ≥ 1n n ≥ 1
  here c = 1 and g(n) = n

  t(n) = Ω(n)

  2n + 3 ≥ 1log n n ≥ 1
  here c = 1 and g(n) = log n

  t(n) = Ω(log n)

Asymptotic Analysis of Insertion sort

- Time Complexity:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

- Best Case: the best case occurs if the array is already sorted, t_j=1 for j=2,3...n.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) .$$

- Linear running time: O(n)

Asymptotic Analysis of Insertion sort

- Worst case : If the array is in reverse sorted order

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \quad \text{and} \quad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8) .$$

- Quadratic Running time. O(n2)

Properties of O, Ω and θ

General property:
If t(n) is O(g(n)) then a * t(n) is O(g(n)). Similar for Ω and θ

Transitive Property :
If *f (n)* ϵ O(*g(n)*) and *g(n)* ϵ O(*h(n)*), then *f (n)* ϵ O(*h(n)*); that is O is transitive.
Also Ω, θ, o and ω are transitive.

Reflexive Property
If f(n) is given then f(n) is O(f(n))

Symmetric Property
If f(n) is θ(g(n)) then g(n) is θ(f(n))

Transpose Property
If f(n) = O(g(n)) then g(n) is Ω(f(n))

Review Questions

- Indicate constant time complexity in terms of Big-O notation.
  A. O(n)
  B. O(1)
  C. O(logn)
  D. O(n^2)
- Big oh notation is used to describe _____
- Big O Notation is a _____function used in computer science to describe an _____
- Big Omega notation (Ω) provides _____bound.
- Given T1(n) =*O*(f(n)) and T2(n)=*O*(g(n).Find T1(n).T2(n)

ASYMPTOTIC NOTATION-THETA MATHEMATICAL FUNCTIONS

Asymptotic Notation – THETA Θ

f(n) = θ (g(n)) if and only if there c_1, c_2 and n_0 such that

$$c_1|g(n)| \le |f(n)| \le c_2|g(n)|$$

for all $n \ge n_0$

$f(n) = O(g(n))$ and $f(n) = \theta(g(n))$

Example: $f(n) = 18n + 9$

If f(n) is between $c_1 g(n)$ and $c_2 g(n)$, $\forall n \ge n_0$, then $f(n) \in \Theta(g(n))$ and g(n) is an asymptotically tight bound for f(n) and f(n) is amongst h(n) in the set.

Example - THETA

- Show that n2 /2 – 2n = Θ(n2).

Solution By the definition, we can write

$c_1 g(n) \le h(n) \le c_2 g(n)$
$c_1 n^2 \le n^2/2 - 2n \le c_2 n^2$

Dividing by n^2, we get

$c_1 n^2/n^2 \le n^2/2n^2 - 2n/n^2 \le c_2 n^2/n^2$
$c_1 \le 1/2 - 2/n \le c_2$

This means $c_2 = 1/2$ because $\lim_{n\to\infty} 1/2 - 2/n = 1/2$ (Big O notation)

To determine c_1 using Ω notation, we can write

$0 < c_1 \le 1/2 - 2/n$

We see that $0 < c_1$ is minimum when n = 5. Therefore,

$0 < c_1 \le 1/2 - 2/5$

Hence, $c_1 = 1/10$

Now let us determine the value of n_0

$1/10 \le 1/2 - 2/n_0 \le 1/2$
$2/n_0 \le 1/2 - 1/10 \le 1/2$
$2/n_0 \le 2/5 \le 1/2$

Example - THETA

$n_0 \ge 5$

You may verify this by substituting the values as shown below.

$c_1 n^2 \le n^2/2 - 2n \le c_2 n^2$
$c_1 = 1/10,\ c_2 = 1/2$ and $n_0 = 5$
$1/10(25) \le 25/2 - 20/2 \le 25/2$
$5/2 \le 5/2 \le 25/2$

Thus, in general, we can write, $1/10n^2 \le n^2/2 - 2n \le 1/2n^2$ for $n \ge 5$.

Mathematical Functions

FUNCTION	REPRESENTATION	EXAMPLE
FLOOR	$\lfloor x \rfloor$	$\lfloor 7.45 \rfloor = 7$ $\lfloor -7.45 \rfloor = -8$
CEIL	$\lceil x \rceil$	$\lceil 7.45 \rceil = 8$ $\lceil -7.45 \rceil = -7$
MODULUS	k (mod M) = r k= Mq + r 0 ≤ r < M	25(mod 7) = 4 25(mod 5) = 0
INTEGER	INT(X)	INT (3.14) = 3 INT (-8.5) = -8
ABSOLUTE	ABS | X|	ABS | -15| = 15 ABS | 7 | = 7

Mathematical Functions (Cont..)

FUNCTION	REPRESENTATION	EXAMPLE
Factorial	n! = 1 * 2 * 3 * ..... * (n-2) * (n-1) * n	4! = 1 * 2 * 3 * 4 = 24
PERMUTATION	Arrangement of the Elements	n elements arranged in n! ways
EXPONENTS	$a^m,\ a^{-m} = \frac{1}{a^m}, a^{m/n} = \sqrt[n]{a^m} = (\sqrt[n]{a})^m$	2^4 = 2 * 2 * 2 * 2 = 16 2^-4 = 1 / 24 = 1 / 16
LOGARITHM	$y = \log_b n \equiv b^y = n$	$\log_{10} 100 = 2\ as\ 10^2 = 100$
SUMMATION	$\sum_{k=1}^{n} a_k$	$a_1 + a_2 + a_3 + \cdots + a_{n-1} + a_n$

Review Questions

1. Give examples of functions that are in Θ notation as well as functions that are not in Θ notation.
2. Which function gives the positive value of the given input?
3. K (mod) M gives the reminder of ____ divided by ____
4. For a given set of number n, the number of possible permutation will be equal to the _____ value of n.
5. _____Notation is used to specify both the lower bound and upper bound of the function.

COMPLEXITY – TIME , SPACE TRADE OFF

SPACE COMPLEXITY

- Space complexity is the **total amount of memory space used by an algorithm/program including the space of input values for execution**. So to find space-complexity, it is enough to calculate the space occupied by the variables used in an algorithm/program.
- Space complexity **S(P)** of any algorithm **P** is,
  - **S(P) = C + SP(I)**
- Where **C** is the fixed part and **S(I)** is the variable part of the algorithm which depends on instance characteristic **I**.

Example:

- Algorithm: SUM(A, B)
- Step 1 - START
- Step 2 - C ← A + B + 10
- Step 3 – Stop

Here we have three variables A, B & C and one constant.
Hence **S(P) = 1+3**.
Space requirement depends on data types of given variables & constants. The number will be multiplied accordingly.

How to calculate Space Complexity of an Algorithm?

- In the Example program, 3 integer variables are used.
- The size of the integer data type is 2 or 4 bytes which depends on the architecture of the system (compiler). Let us assume the size as 4 bytes.
- The total space required to store the data used in the example program is 4 * 3 = 12, Since no additional variables are used, no extra space is required. Hence, **space complexity for the example program is O(1), or constant.**

```
#include<stdio.h>
int main()
{
  int a = 5, b = 5, c;
  c = a + b;
  printf("%d", c);
}
```

EXAMPLE 2

```
#include <stdio.h>
int main()
{
  int n, i, sum = 0;
  scanf("%d", &n);
  int arr[n];
  for(i = 0; i < n; i++)
  {
    scanf("%d", &arr[i]);
    sum = sum + arr[i];
  }
  printf("%d", sum);
}
```

- In the code given above, the array stores a maximum of n integer elements. Hence, the space occupied by the array is 4 * n. Also we have integer variables such as n, i and sum.
- Assuming 4 bytes for each variable, the total space occupied by the program is **4n + 12** bytes.
- Since the highest order of n in the equation 4n + 12 is n, so **the space complexity is O(n) or linear.**

EXAMPLE 3

```
#include<stdio.h>
int main()
 {
  int a = 5, b = 5, c;
  c = a + b;
  printf("%d", c);
 }
```

Space Complexity: size(a)+size(b)+size(c)
=>let sizeof(int)=2 bytes=>2+2+2=6 bytes
=>**O(1) or constant**

Example 4

```
#include <stdio.h>
  int main()
  {
    int n, i, sum = 0;
    scanf("%d", &n);
    int arr[n];
   for(i = 0; i < n; i++)
    {
       scanf("%d", &arr[i]); sum = sum + arr[i];
    }
    printf("%d", sum);
  }
```

Space Complexity:

- **The array consists of n integer elements.**
- **So, the space occupied by the array is 4 * n. Also we have integer variables such as n, i and sum. Assuming 4 bytes for each variable, the total space occupied by the program is 4n + 12 bytes.**
- **Since the highest order of n in the equation 4n + 12 is n, so the space complexity is O(n) or linear.**

Time Complexity

- Time Complexity of an algorithm represents the **amount of time required by the algorithm** to run to completion.
- Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the **number of steps**, provided each step consumes constant time.
- Eg. Addition of two n-bit integers takes n steps.
- Total computational time is T(n) = c*n,
- where **c is the time taken for addition of two bits**.
- Here, we observe that T(n) grows linearly as input size increases.

Time Complexity (Cont..)

Two methods to find the time complexity for an algorithm

1. Count variable method
2. Table method

Count Variable Method

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

int Sum(int A[], int N){
    int s=0;                          ①
    for (int i=0; i< N; i++)          ② ③ ④
        s = s + A[i];                 ⑤ ⑥ ⑦
    return s;                         ⑧
}
```

1,2,8: Once
3,4,5,6,7: Once per each iteration of for loop, N iteration
Total: 5N + 3
The *complexity function* of the algorithm is : $f(N) = 5N + 3$

Table Method

- The table contains s/e and frequency.
- The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.
- Frequency is defined as the total number of times each statement is executed.
- Combining these two, the step count for the entire algorithm is obtained.

Example 1

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0$;	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n + 1$	$n + 1$
5 $s := s + a[i]$;	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n + 3$

Example 2

```
int sum(int A[], int n)

{
int sum = 0, i;
 for(i = 0; i < n; i++)
sum = sum + A[i];
 return sum;
}
```

int sumOfList(int A[ ], int n)	Cost	Repeatation	Total
{			
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1 + 1 + 1	1 + (n+1) + n	2n + 2
sum = sum + A[i];	2	n	2n
return sum;	1	1	1
}			4n + 4

- In above calculation
  Cost is the amount of computer time required for a single operation in each line.
- **Repetition** is the amount of computer time required by each operation for all its repetitions.
- **Total** is the amount of computer time required by each operation to execute.
  So above code requires **'4n+4' Units** of computer time to complete the task. Here the exact time is not fixed. And it changes based on the **n** value. If we increase the **n** value then the time required also increases linearly.

Totally it takes '4n+4' units of time to complete its execution

Example 3

Consider the following piece of code...

```
int sum(int a, int b)
{
return a+b;
}
```

In the above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires the same amount of time i.e. 2 units.

If any program requires a fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity

Example 4

```
                          Cost    Times
i = 1;                    c1      1
sum = 0;                  c2      1
while (i <= n) {          c3      n+1
    i = i + 1;            c4      n
    sum = sum + i;        c5      n
}
```

3n+3

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

The time required for this algorithm is proportional to n

Example 5

```
                          Cost    Times
i=1;                      c1      1
sum = 0;                  c2      1
while (i <= n) {          c3      n+1
    j=1;                  c4      n
    while (j <= n) {      c5      n*(n+1)
        sum = sum + i;    c6      n*n
        j = j + 1;        c7      n*n
    }
    i = i +1;             c8      n
}
```

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

The time required for this algorithm is proportional to n^2

INSERTION SORT -TIME COMPLEXITY

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j - 1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1).$$

$= O(n^2)$

Insertion Sort – Analysis

- Running time depends on not only the size of the array but also the contents of the array.
- ***Best-case:*** ➔ **O(n)**
  - Array is already sorted in ascending order.
  - Inner loop will not be executed.
  - The number of moves: 2*(n-1) ➔ O(n)
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** ➔ **$O(n^2)$**
  - Array is in reverse order:
  - Inner loop is executed i-1 times, for i = 2,3, …, n
  - The number of moves: 2*(n-1)+(1+2+…+n-1)= 2*(n-1)+ n*(n-1)/2 ➔ $O(n^2)$
  - The number of key comparisons: (1+2+…+n-1)= n*(n-1)/2 ➔ $O(n^2)$
- Average-case: ➔ **$O(n^2)$**
  - We have to look at all possible initial data organizations.

So, Insertion Sort is $O(n^2)$

Time Complexity of Bubble Sort

In the case of the standard version of the bubble sort, we need to do N iterations. In each iteration, we do the comparison and we perform swapping if required. Given an array of size N, the first iteration performs $(N-1)$ comparisons. The second iteration performs $(N-2)$ comparisons. In this way, the total number of comparison will be:

$$(N-1)+(N-2)+(N-3)+\ldots\ldots+3+2+1=\frac{N(N-1)}{2}=\mathcal{O}(N^2)$$

Therefore, **in the average case, the time complexity of the standard bubble sort would be $\mathcal{O}(N^2)$.**

Now let's talk about the best case and worst case in bubble sort. The best case would be when the input array is already sorted. In this case, we check all the N elements to see if there is any need for swaps. If there is no swapping still we continue and complete N iterations. Therefore, **in the best scenario, the time complexity of the standard bubble sort would be $\mathcal{O}(N^2)$.**

In the worst case, the array is reversely sorted. So we need to do $(N-1)$ comparisons in the first iteration, $(N-2)$ in the second interactions, and so on. Hence, **the time complexity of the bubble sort in the worst case would be the same as the average case and best case: $\mathcal{O}(N^2)$.**

Review Questions

Sort the following numbers using bubble sort and Insertion sort

- 35,12,14,9,15,45,32,95,40,5
- 3, 1, 4, 1, 5, 9, 2, 6, 5
- 17 14 34 26 38 7 28 32
- 35,12,14,9,15,45,32,95,40,5

Review questions

1. Calculate the time complexity for the below algorithms using table method

```
1  Algorithm Fibonacci(n)
2  // Compute the nth Fibonacci number.
3  {
4      if (n ≤ 1) then
5          write (n);
6      else
7      {
8          fnm2 := 0; fnm1 := 1;
9          for i := 2 to n do
10         {
11             fn := fnm1 + fnm2;
12             fnm2 := fnm1; fnm1 := fn;
13         }
14         write (fn);
15     }
16 }
```

2. Determine the frequency counts for all statements in the following two algorithm segments:

```
1  for i := 1 to n do
2      for j := 1 to i do
3          for k := 1 to j do
4              x := x + 1;
```

(a)

```
1  i := 1;
2  while (i ≤ n) do
3  {
4      x := x + 1;
5      i := i + 1;
6  }
```

(b)

3.

```
1  Algorithm Transpose(a, n)
2  {
3      for i := 1 to n − 1 do
4          for j := i + 1 to n do
5          {
6              t := a[i, j]; a[i, j] := a[j, i]; a[j, i] := t;
7          }
8  }
```

21CSC201J
DATA STRUCTURES AND ALGORITHMS

UNIT-1

DATA STRUCTURE OPERATIONS

Data Structure Operations

- The data in the data structures are processed by certain operations.
  - Traversing
  - Searching
  - Inserting
  - Updating
  - Deleting
  - Sorting
  - Merging

Data Structure Operations (Cont..)

- **Traversing** - Visiting each record so that items in the records can be accessed.
- **Searching** - Finding the location of the record with a given key or value or finding all records which satisfy given conditions.
- **Inserting** - Adding a new record to the data structure.
- **Updating** – Change the content of some elements of the record.
- **Deleting** - Removing records from the data structure.
- **Sorting** - Arranging records in some logical or numerical order.
  - **(Eg: Alphabetic order)**
- **Merging** - Combing records from two different sorted files into a single file.

Example

- An organization contains a membership file in which each record contains the following data for a given member:

  Name, Address, Telephone Number, Age, Sex

a. Suppose the organization wants to announce a meeting through a mailing. Then one would **traverse** the file to obtain Name and Address of each member.

b. Suppose one wants to find the names of all members living in a particular area. Again **traverse and search** the file to obtain the data.

c. Suppose one wants to obtain Address of a specific Person. Then one would **search** the file for the record containing Name.

d. Suppose a member dies. Then one would **delete** his or her record from the file.

Example

e. Suppose a new person joins the organization. Then one would **insert** a record into the file

f. Suppose a member has shifted to a new residence his/her address and telephone number need to be changed. Given the name of the member, one would first need to **search** for the record in the file. Then perform the **update** operation, i.e. change some items in the record with the new data.

g. Suppose one wants to find the number of members whose age is 65 or above. Again one would **traverse** the file, counting such members.

ABSTRACT DATA TYPE

Abstract Data Types

- An Abstract Data type (ADT) is a type for objects whose behavior is defined by a **set of value and a set of operations**.
- ADT refers to a set of data values and associated operations that are specified accurately, independent of any particular implementation.
- The ADT consists of a set of definitions that allow us to use the functions while hiding the implementation.
- Abstract in the context of data structures means everything except the detailed specifications or implementation.
- Data type of a variable is the set of values that the variable can take.

ADT = Type + Function Names + Behaviour of each function

- **Examples: Stacks, Queues, Linked List**

ADT Operations

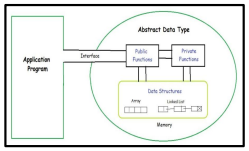
- While modeling the problems the necessary details are separated out from the unnecessary details. This process of modeling the problem is called abstraction.
- The model defines an abstract view to the problem. It focuses only on problem related stuff and that you try to define properties of the problem.
- These properties include
  - The data which are affected.
  - The operations which are identified.

ADT Operations (Cont..)

- Abstract data type operations are
  - **Create -** Create the data structure.
  - **Display -** Displaying all the elements stored in the data structure.
  - **Insert -** Elements can be inserted at any desired position.
  - **Delete -** Desired element can be deleted from the data structure.
  - **Modify -** Any desired element can be deleted/modified from the data structure.

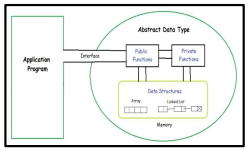
Abstract Data Types Model

- The ADT model has two different **parts functions (public and private) and data structures.**
- Both are contained within the ADT model itself, and do not come within the scope of the application program.
- Data structures are available to all of the ADT's functions as required, and a **function may call any other function** to accomplish its task.
- Data structures and functions are within the scope of each other.



Abstract Data Types Model (Cont..)

- Data are entered, accessed, modified and deleted through the external application programming interface.
- For each ADT operation, there is an algorithm that performs its specific task.
- The operation name and parameters are available to the application, and they provide only an interface to the application.
- When a list is controlled entirely by the program, it is implemented using simple structure.



Review Questions

- _____ are used to manipulate the data contained in various data structures.
- In _____, the elements of a data structure are stored sequentially.
- _____ of a variable specifies the set of values that the variable can take.
- Abstract means _____.
- If the elements of a data structure are stored sequentially, then it is a _____.

ALGORITHMS
SEARCHING TECHIQUES

Algorithms

- An algorithm is a well defined list of steps for solving a particular problem.
- The time and space are two major measures of the efficiency of an algorithm.
- The complexity of an algorithm is the function which gives the running time and / or space in terms of input size.

Searching Algorithms

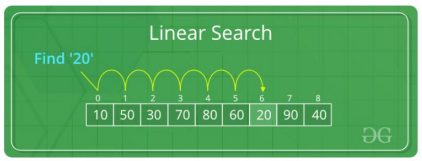
- Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.
- Based on the type of search operation, these algorithms are generally classified into two categories:
  - Linear Search
  - Binary Search

Linear Search

- Linear search is a very basic and simple search algorithm.
- In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found or we can establish that the element is not present.
- It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns the index of the element in the array, else it return -1.
- Linear search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Linear Search - Example

- Search each record of the file, one at a time, until finding the given value.



Algorithm for Linear Search

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:          IF A[I] = VAL
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                 [END OF IF]
                 SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

- In Steps 1 and 2 of the algorithm, we initialize the value of POS and I.
- In Step 3, a while loop is executed that would be executed till I is less than N.
- In Step 4, a check is made to see if a match is found between the current array element and VAL.
- If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL.
- If all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

Program for Linear Search

```c
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 20
int main(int argc, char *argv[])
{
int arr[size], num, i, n, found = 0, pos = -1;
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
printf("\n Enter the elements: ");
for(i=0;i<n;i++)
{
scanf("%d", &arr[i]);
}
printf("\n Enter the number that has to be searched : ");
scanf("%d", &num);
for(i=0;i<n;i++)
{
if(arr[i] == num)
{
found =1;
pos=i;
printf("\n %d is found in the array at position= %d", num,i+1);
/* +1 added in line 23 so that it would display the number in
the first place in the array as in position 1 instead of 0 */
break;
}
}
if (found == 0)
printf("\n %d does not exist in the array", num);
return 0;
}
```

```
Enter the number of elements in the array : 10

Enter the elements: 32
25
69
87
35
56
54
85
95
49

Enter the number that has to be searched : 35

35 is found in the array at position= 5
```

Linear Search - Advantages

- When a key element matches the first element in the array, then linear search algorithm is best case because executing time of linear search algorithm is O(n), where n is the number of elements in an array.
- The list doesn't have to sort. Contrary to a binary search, linear searching does not demand a structured list.
- Not influenced by the order of insertions and deletions. Since the linear search doesn't call for the list to be sorted, added elements can be inserted and deleted.

Linear Search - Disadvantages

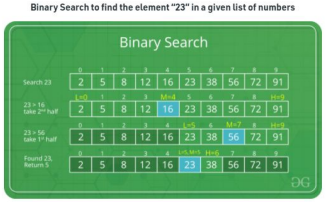
- The drawback of a linear search is the fact that it is time consuming if the size of data is huge.
- Every time a vital element matches the last element from the array or an essential element does not match any element Linear search algorithm is the worst case.

Binary Search

- Binary Search is used with sorted array or list. In binary search, we follow the following steps:
  1) We start by comparing the element to be searched with the element in the middle of the list/array.
  2) If we get a match, we return the index of the middle element and terminate the process
  3) If the element/number to be searched is greater than the middle element, we pick the elements on the left/right side of the middle element, and move to step 1.
  4) If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the right/left side of the middle element, and move to step 1.

Binary Search - Example

- Binary Search is useful when there are large number of elements in an array and they are sorted.



Algorithm for Binary Search

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:          SET MID = (BEG + END)/2
Step 4:          IF A[MID] = VAL
                       SET POS = MID
                       PRINT POS
                       Go to Step 6
                 ELSE IF A[MID] > VAL
                       SET END = MID - 1
                 ELSE
                       SET BEG = MID + 1
                 [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
          PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

- In Step 1, we initialize the value of variables, BEG, END, and POS.
- In Step 2, a while loop is executed until BEG is less than or equal to END.
- In Step 3, the value of MID is calculated.
- In Step 4, we check if the array value at MID is equal to VAL. If a match is found, then the value of POS is printed and the algorithm exits. If a match is not found, and if the value of A[MID] is greater than VAL, the value of END is modified, otherwise if A[MID] is greater than VAL, then the value of BEG is altered.
- In Step 5, if the value of POS = –1, then VAL is not present in the array and an appropriate message is printed on the screen before the algorithm exits.

Program for Binary Search

```c
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 10
int smallest(int arr[], int k, int n); // Added to sort array
void selection_sort(int arr[], int n); // Added to sort array
int main(int argc, char *argv[]) {
int arr[size], num, i, n, beg, end, mid, found=0;
printf("\n Enter the number of elements in the array: ");
scanf("%d", &n);
printf("\n Enter the elements: ");
for(i=0;i<n;i++)
{
scanf("%d", &arr[i]);
}
selection_sort(arr, n); // Added to sort the array
printf("\n The sorted array is: \n");
for(i=0;i<n;i++)
printf(" %d\n", arr[i]);
printf("\n\n Enter the number that has to be searched: ");
scanf("%d", &num);
beg = 0, end = n-1;
while(beg<=end)
{
mid = (beg + end)/2;
if (arr[mid] == num)
{
printf("\n %d is present in the array at position %d", num, mid+1);
found =1;
break;
}
else if (arr[mid]>num)
end = mid-1;
else
beg = mid+1;
}
if (beg > end && found == 0)
printf("\n %d does not exist in the array", num);
return 0;
}
int smallest(int arr[], int k, int n)
{
int pos = k, small=arr[k], i;
for(i=k+1;i<n;i++)
{
if(arr[i]< small)
{
small = arr[i];
pos = i;
}
}
return pos;
}
void selection_sort(int arr[],int n)
{
int k, pos, temp;
for(k=0;k<n;k++)
{
pos = smallest(arr, k, n);
temp = arr[k];
arr[k] = arr[pos];
arr[pos] = temp;
}
}
```

```
Enter the elements: 56
23
14
24
58
56
89
92
31
38

The sorted array is:
14
23
24
31
38
56
56
58
89
92

Enter the number that has to be searched: 92

92 is present in the array at position 10
```

Binary Search (Cont..)

Advantages

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

Disadvantages

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.

Difference between Linear & Binary Search

Linear Search	Binary Search
• Linear Search necessitates the input information to be not sorted	• Binary Search necessitates the input information to be sorted.
• Linear Search needs equality comparisons.	• Binary Search necessitates an ordering contrast.
• Linear search has complexity C(n)= n/2.	• Binary Search has complexity C(n)= $\log_2 n$.
• Linear Search needs sequential access.	• Binary Search requires random access to this information.