

# CS2002D PROGRAM DESIGN

## Lecture - 1

- What ?
- When?
- How?
- Why?

# Why do we learn Program Design course?

1. \_\_\_\_\_

2. \_\_\_\_\_

3. \_\_\_\_\_

Question-1: What are the essential steps in solving a problem using computer?

1.

2.

3.

# Course Outcome - 1

Design, analyse and prove the correctness of simple, iterative and recursive algorithms

Question 2: How efficiently can you search a name in an array consisting of 10 Million names?

1. \_\_\_\_\_

2. \_\_\_\_\_

3. \_\_\_\_\_

Question 3: How efficiently can you arrange 100 Million names in alphabetical order?

1. \_\_\_\_\_

2. \_\_\_\_\_

3. \_\_\_\_\_

## Course Outcome - 2

Analyze algorithms for sorting and searching



Question 4: Consider 10 Million names stored in an alphabetical order. I want to insert a name in this sorted list at location 10,000. How can we do this?

1. \_\_\_\_\_

2. \_\_\_\_\_

3. \_\_\_\_\_

Question 5:How to represent the hierarchy of an institute / organization. How do we systematically access every role in the institute/organization?

1. \_\_\_\_\_

2. \_\_\_\_\_

3. \_\_\_\_\_

## Course Outcome - 3

Select appropriate data structure for solving a given problem

Question 6:How to create a variable sized array?  
How the different memory management functions  
are implemented?

1. \_\_\_\_\_

2. \_\_\_\_\_

3. \_\_\_\_\_

## Course Outcome - 4

Compare different techniques for memory management

# COURSE OBJECTIVES

- To introduce the students to the concept of algorithms, their role in computing and simple data structures.
- To equip the students to design correct and efficient algorithms for computational problems.

# Conditional Control Structures

# Objectives

- To understand how decisions are made in a computer
- To understand the relational operators
- To understand the logical operators and, or and not
- To write programs using relational and logical operators
- To write programs that uses two way selection.  
if...else statements
- To write programs that uses multi way selection if...else  
if ladder and switch case
- To understand classification of characters



- What do Control Structures do?
- What are the different types of execution flow for them?

# Control Structures

- **Control structures** control the flow of execution in a program or function.
- There are three kinds of execution flow:
  - **Sequence:**
    - the execution of the program is sequential. (add/sub/mul/div of two numbers)
  - **Selection:**
    - A control structure which chooses alternative to execute.
  - **Repetition:**
    - A control structure which repeats a group of statements.
- We will focus on the **selection** control structure.

**THANK YOU**

# CS2002D PROGRAM DESIGN

## Lecture 2

# Conditional Control Structures

# Objectives

- To understand how decisions are made in a computer
- To understand the relational operators
- To understand the logical operators and, or and not
- To write programs using relational and logical operators
- To write programs that uses two way selection. if...else statements
- To write programs that uses multi way selection if...else if ladder and switch case

- What do Control Structures do?
- What are the different types of execution flow for a program?

# Control Structures

- **Control structures** control the flow of execution in a program or function.
- There are three kinds of execution flow:
  - **Sequence:**
    - the execution of the program is sequential. (add/sub/mul/div of two numbers)
  - **Selection:**
    - A control structure which chooses alternative to execute.
  - **Repetition:**
    - A control structure which repeats a group of statements.
- We will focus on the **selection** control structure.



- **Selection** control structure?
- Example?

# Conditions

- A program may choose among alternative statements by testing the value of key variables.
  - e.g.,    if    **mark is greater than 50**  
                  print “Pass”
- **Condition** is an expression that is either false (represented by 0) or true (represented by 1).
  - e.g., “**mark is greater than 50**” is a condition.
- Conditions may contain **relational** or **equality operators**, and have the following forms.
  - variable **relational-operator** variable (or constant)
  - variable **equality-operator** variable (or constant)

# Operators Used in Conditions

Operator	Meaning	Type
<	Less than	Relational
>	Greater than	Relational
<=	Less than or equal to	Relational
>=	Greater than or equal to	Relational
==	Equal to	Equality
!=	Not equal to	Equality

# Examples of Conditions

Operator	Condition	Meaning
<code>&lt;=</code>	<code>x &lt;= 0</code>	<code>x</code> less than or equal to 0
<code>&lt;</code>	<code>Power &lt; MAX_POW</code>	<code>Power</code> less than <code>MAX_POW</code>
<code>==</code>	<code>yes_or_no == 'Y'</code>	<code>yes_or_no == 'Y'</code>
<code>!=</code>	<code>num != SETINEL</code>	<code>num</code> not equal to <code>SETINEL</code>

- Logical Operators?
- Examples?

# Logical Operators

- There are three kinds of **logical operators**.
  - **&&**: and
  - **| |**: or
  - **!**: not expression
- **Logical expression** is an expression which uses one or more logical operators, e.g.,
  - (temperature > 90.0 **&&** humidity > 0.90)
  - **!(n <= 0 | | n >= 100)**.


# The Truth Table of Logical Operators

Op 1	Op 2	Op 1 && Op2	Op 1    Op2
nonzero	nonzero	1	1
nonzero	0	0	1
0	nonzero	0	1
0	0	0	0

Op 1	! Op 1
nonzero	0
0	1

# Operator Precedence

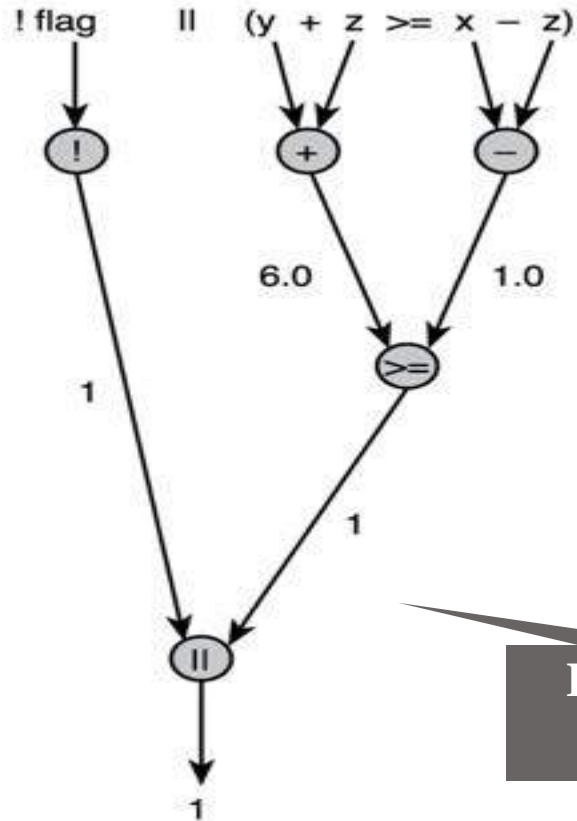
- An operator's **precedence** determines its order of evaluation.
- **Unary operator** is an operator that has only one operand.
  - **!**, **+**(plus sign), **-**(minus sign), and **&**(address of)
  - They are evaluated second only after function calls.

Operator	Precedence
function calls	highest
! + - &	
* / %	
+ -	
< <= >= >	
== !=	
& &	
=	lowest

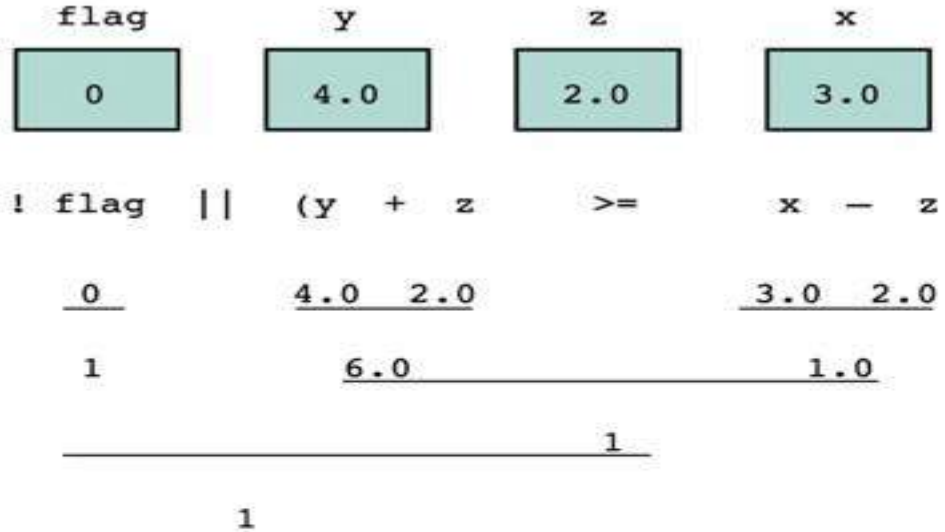


# Evaluation for

**`!flag || (y + z >= x - z)`**



Evaluation tree



The result of this expression is true

# DeMorgan's Theorem

- **DeMorgan's theorem** gives us a way of transforming a logical expression into its complement.
  - The complement of  $\text{expr}_1 \&\& \text{expr}_2$  is  $\text{comp}_1 \parallel \text{comp}_2$ , where  $\text{comp}_1$  and  $\text{comp}_2$  are the complement of  $\text{expr}_1$  and  $\text{expr}_2$ , respectively.
  - The complement of  $\text{expr}_1 \parallel \text{expr}_2$  is  $\text{comp}_1 \&\& \text{comp}_2$ .
- e.g.,  $\text{age} > 25 \&\& (\text{status} == \text{'S'} \parallel \text{status} == \text{'D'})$   
is equal to  
 $(\text{age} \leq 25 \parallel (\text{status} != \text{'S'}) \&\& \text{status} != \text{'D'})$

# Conditional Statements

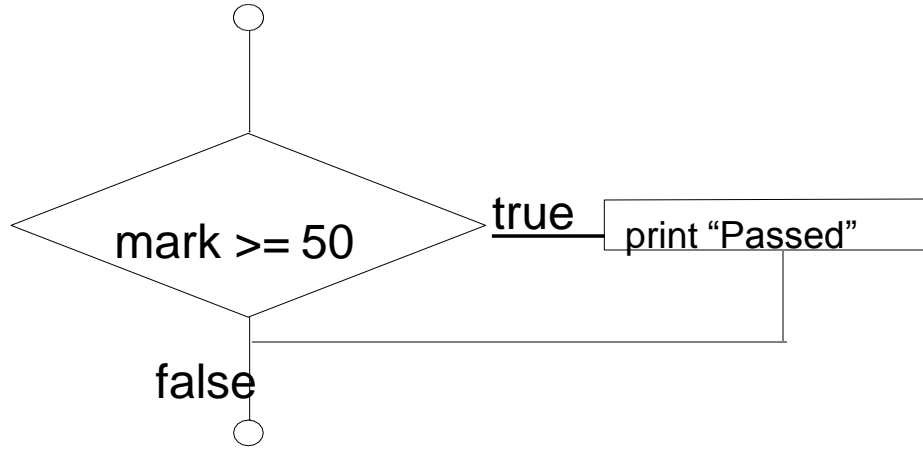
- Pseudocode example:

*If student's mark is greater than or equal to 60*

*Print "Passed"*

- If the condition is **true**
  - Print statement executed, program continues to next statement
- If the condition is **false**
  - Print statement ignored, program continues

# Flowchart of **if** statement



A decision can be made on any expression.

`zero: false`  
`nonzero: true`

Example:

`Mark=60` is `true`

# Translation into C

*If student's mark is greater than or equal to 50*

*Print "You have passed in examination"*

```
if ( mark >= 50 )  
    printf("You have passed in examination!\n");
```

- **if** structure
  - Single-entry/single-exit

# Single way selection `if` Statement

- The `if` statement is the primary **selection** control structure. It has a **null else statement**.

- Syntax:

```
if (condition)  
    statement;
```

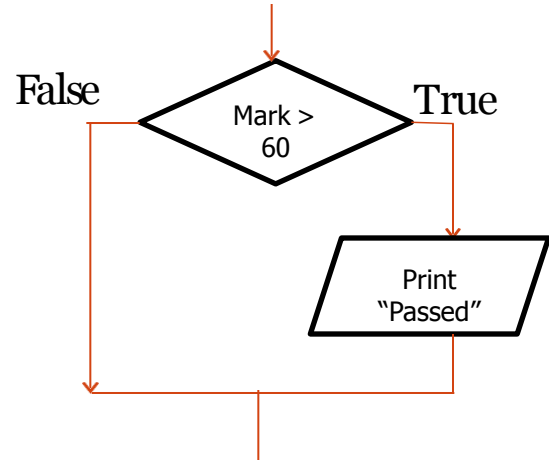
**Eg. 1** *If student's mark is greater than or equal to 50*  
*Print "You have passed in examination"*

```
if(mark > 50)  
    printf("You have passed in examination!\n");
```

**Eg. 2**

*If quantity is not equal to 0*  
*calculate price as price \* quantity*  
*print price*

```
if (quantity != 0.0){  
    price = price * quantity;  
    printf("Price=Rs. %8.2f \n", price);  
}
```



# Two way selection

## if else Statement

- The `if else` statement is the primary **two way selection control structure**.
- Syntax:

**`if(condition)`**

**`statement;`**

**`else`**

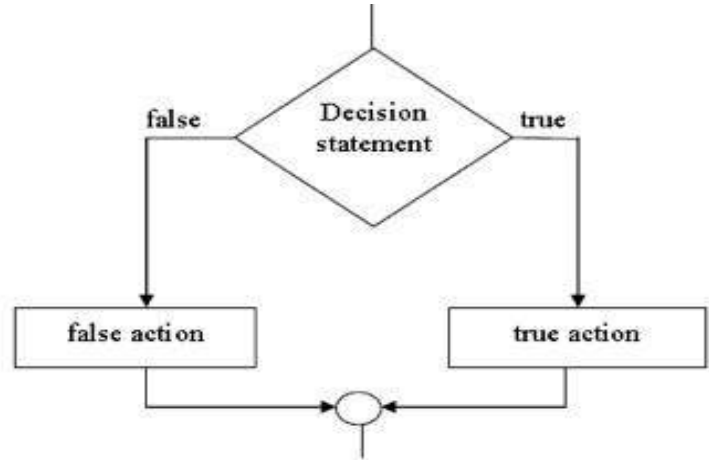
**`statement;`**

Eg.1 `if(mark > 50)`

`printf("You have passed in examination! \n");`

`else`

`printf("Sorry Try again!\n");`



**Eg. 2**

```
if (quantity!= 0.0 ){  
    price = price * quantity;  
    printf(“Price=Rs. %8.2f!\n”,price);  
    }  
  
else  
    printf(“Quantity is less than 0.0\n”);
```



# Single way selection: **if Statement**

- Write a C program to print the number entered by user only if the number entered is negative.

```
void main(){
    int num;
    printf("Enter a number to check.\n");
    scanf("%d",&num);

    if(num<0) {
        /* checking whether number is less than 0 or not. */
        printf("Number = %d\n",num);
    }
    /*If test condition is true, statement above will be executed, otherwise it will not be executed */
    printf("The if statement in C programming is easy.");
    return 0;
}
```

# Two way selection: `if else` Statement

- Write a C program to print the number entered by user only if the number entered is positive or negative.

```
void main(){
    int num;
    printf("Enter a number to check.\n");
    scanf("%d",&num);
    if(num<0) {
        /* checking whether number is less than 0 or not. */
        printf("Negative Number = %d\n",num);
    }
    else{
        /*If test condition is false statement */
        printf("Positive Number = %d\n",num);
    }
    return 0;
}
```

# Two way selection: **if else Statement**

- Write a C program to print the number entered by user only if the number entered is even or odd.

```
void main(){
    int num;
    printf("Enter a number to check.\n");
    scanf("%d",&num);

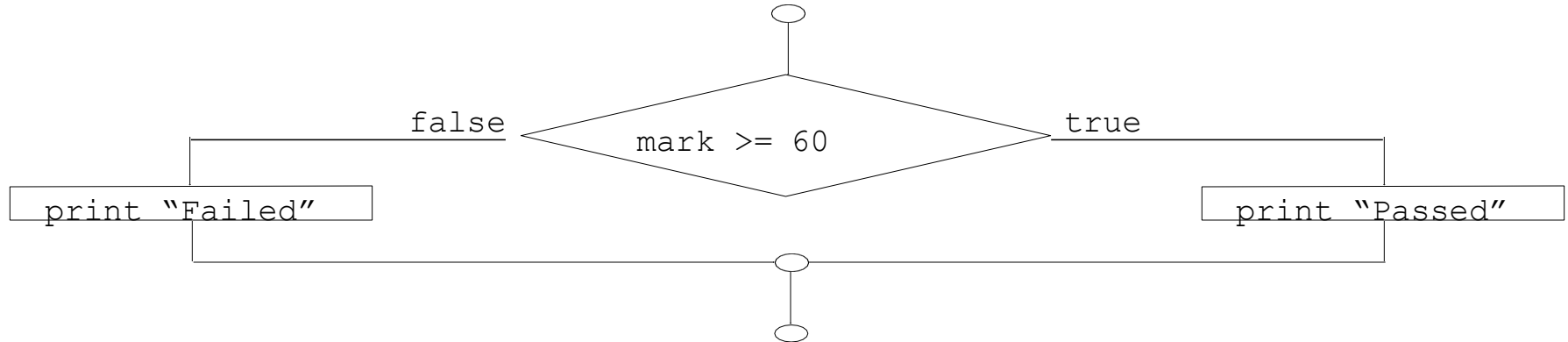
    if(num%2==0) {
        /* checking whether number is odd or even. */
        printf("Even Number = %d\n",num);
    }
    else{
        /*If test condition is false statement */
        printf("OddNumber = %d\n",num);
    }
    return 0;
}
```

# if/else Selection Structure

- Ternary conditional operator(?:)
  - Three arguments (condition, value if true, value if false )
  - Code could be written:

**( mark >= 50 ? printf("Passed") : printf("Failed") )**

Condition                  Value if true      Value if false



# Combining condition with logical operators

- If both conditions need to be true use logical AND (&& )operator
  - If first condition is false second condition not evaluate.
- If only one of the condition true use logical OR (|| ) operator
  - If first condition is true second condition not evaluated.
- Logical AND (&& )and logical OR (|| )act as short circuiting operators.



# Combining condition with logical operators

## Eg.1

```
/* if condition 1 is true condition 2 will be evaluated*/ if (
    (road_status == 'S') &&          (temp>0) )

    printf("Wet roads ahead!\n");

else

    printf("Drive carefully!\n");
```

## Eg.2

```
/* if condition 1 is true condition 2 will not be evaluated*/
if ( (road_status == 'S') ||          (temp>0) )

    printf("Wet roads ahead!\n");

else
```

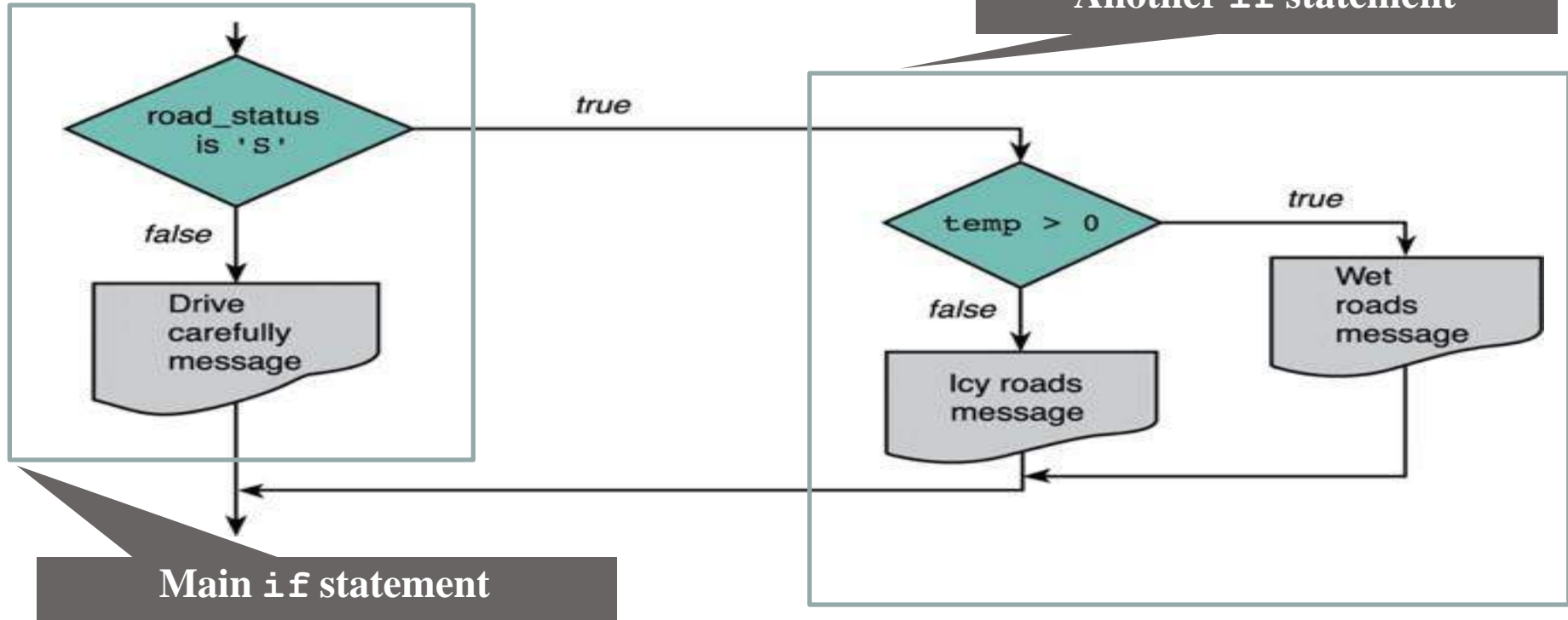
# Nested if Statements

- Nested if statement is an if statement with **another if statement** as its true task or false task.

• e.g.,

```
if (road_status == 'S') if(temp > 0) {  
    printf("Wet roads ahead!\n");  
} else {  
    printf("Icy roads ahead!\n");  
}  
else  
    printf("Drive carefully!\n");
```

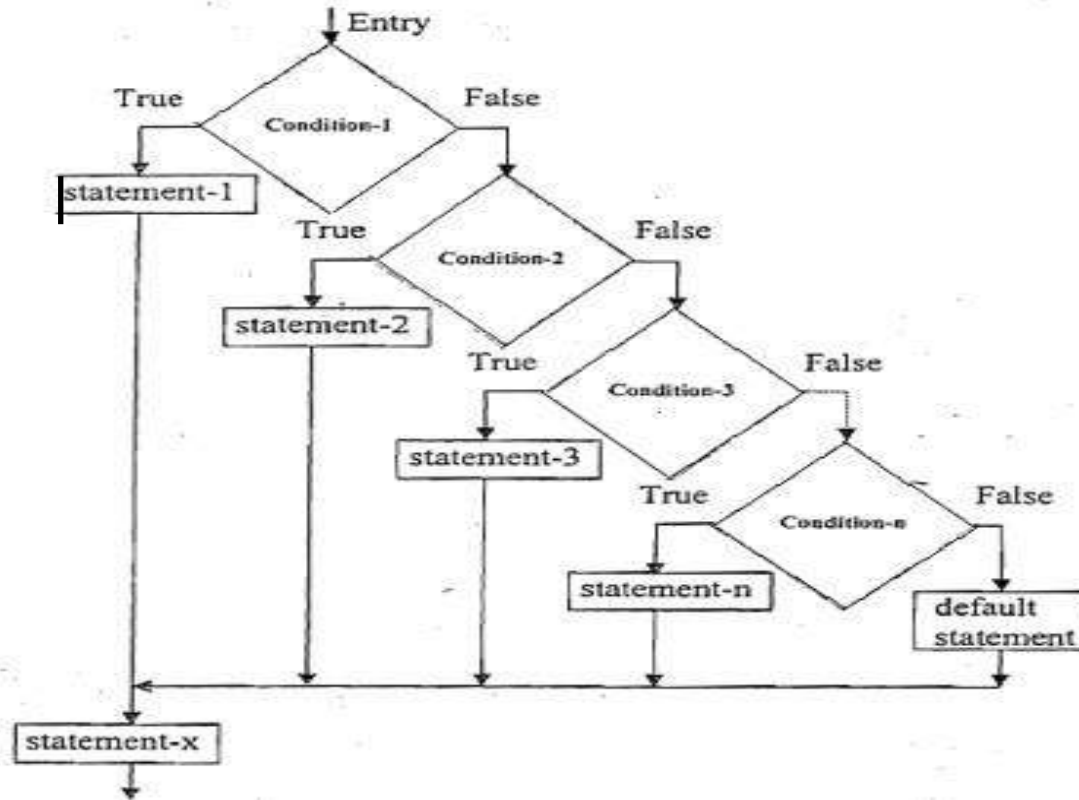
# An Example for the Flowchart of Nested `if` Statements



# Multiway Decision Making statements

- If there are many alternatives, it is better to use the syntax of **multiway decision**
- **C programming provides two types of multiway control structures.**
  - if else if ladder
  - switch case

# Multiway Decision Making statements if else if ladder



# Multiway Decision Making statements if else if ladder

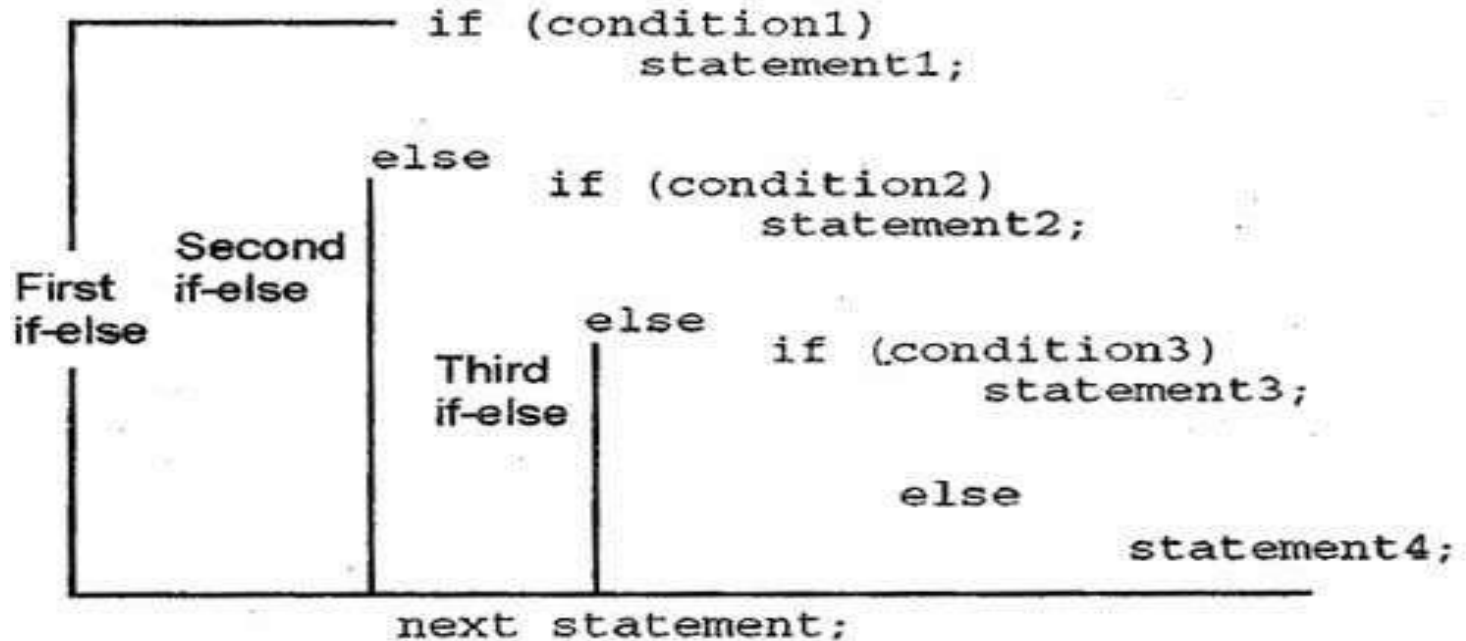
- If there are many alternatives, it is better to use the syntax of **multiway decision**.

- Syntax:

```
if(condition 1) {  
    statements; /* statements will execute if the condition 1 is true */  
else if(condition 2) {  
    statements; /* statements will execute if the condition 2 is true */  
else if(condition 3) {  
    statements; /* statements will execute if the condition 3 is true */  
else if(condition n) {  
    statements; /* statements will execute if the condition n is true */  
  
else {  
    statements; /* statements will execute if all conditions are false */  
}
```

# Multiway Decision Making statements

## if else if ladder



# Multiway Decision Making statements if else if ladder

Eg. Write a C program to check if a number is positive negative or zero



```
void main() {  
    int num;  
    printf("Enter a number = ");  
    scanf("%d",&num);  
    if(num>0) {  
        printf("Number is Positive");  
    }  
    else if(a<0) {  
        printf("Number is Negative");  
    }  
    else {  
        printf("Number is Zero");  
    }  
    return 0;  
}
```

# Multiway-Selection Structure

- **switch case**

- Useful when variable or expression is tested for multiple values
- Consists of a series of **case** labels and an optional **default** case
- **break** is (almost always) necessary
  - It is used to terminate a case in the **switch** statement

# The switch Statement

- The `switch` statement is used to select one of several alternatives when the selection is based on the value of **a single variable** or **an expression**.

```
switch (controlling expression) {  
    case label1:  
        statement1  
        break;  
    case label2:  
        statement2  
        break;  
    ...  
    case labeln:  
        statementn  
        break;  
    default:  
        statementd;  
}
```

If the result of this controlling expression matches *label*<sub>1</sub>, execute statement<sub>1</sub> and then break this switch block.

If the result matches none of all labels, execute the default statement<sub>d</sub>.

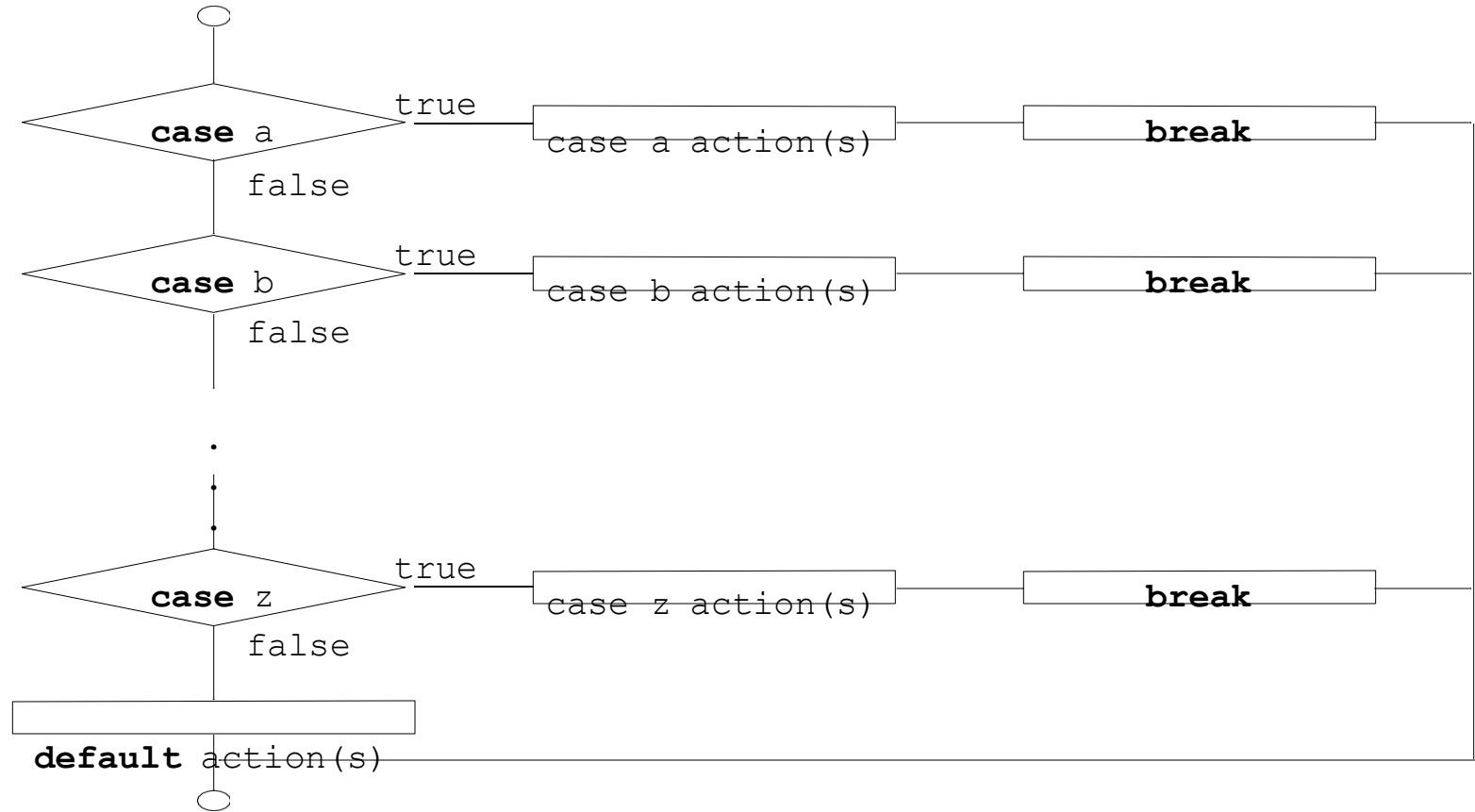
# switch case and if else if ladder

```
switch (expression) {  
    case val1:      statement  
                    break;  
  
    case val2:      statement  
                    break;  
  
    case valn:      statement  
                    break;  
  
    default:        statement  
                    ....  
}
```



```
if (expression == val1)  
    statement  
else if (expression == val2)  
    statement  
  
else if (expression == valn)  
    statement  
  
else  
    statement
```

# Flowchart switch case



## An Example of a `switch` Statement with Type `numeric(int)` Case Labels

Print the day based on the number entered by the user. If any number other than 1-7 is entered say unknown number.

## An Example of a `switch` Statement with Type `numeric (int)` Case Labels

```
/* Print the day based on the number entered*/  
void main() {  
    int day;  
    printf("Enter the day of the week(1-7):");  
    scanf("%d",&day);  
    switch(day) {  
        case 1: printf("Sunday\n");  
                break;  
        case 2: printf("Monday\n");  
                break;  
        case 3: printf("Tuesday\n");  
                break;  
        case 4: printf("Wednesday\n"); break;  
        case 5: printf("Thursday\n");  
                break;  
        case 6: printf("Friday\n");  
                break;  
        case 7: printf("Saturday\n");  
                break;  
        default: printf("Incorrect entry Try again!\n");  
    }  
}
```

## An Example of a `switch` Statement with Type `char` Case Labels

Write a program to enter the ship name based on the character entered.

If it is B or b Battleship

If it is C or c Cruiser

If it is D or d Destroyer

If it is F or f Frigate

If it is not any of the letter print unknown ship



# Try it yourself

1. Write a program to enter the temperature and print the following message according to the given temperature by using if else ladder statement.

- |                        |                       |
|------------------------|-----------------------|
| 1. $T \leq 0$          | "Its very very cold". |
| 2. $0 < T < 10$        | "Its cold".           |
| 3. $10 \leq T \leq 20$ | "Its cool out".       |
| 4. $20 < T \leq 30$    | "Its warm".           |
| 5. $T > 30$            | "Its hot".            |

## Try it yourself

2. Write a program that prompts the user to **input the boiling point** in degree Celsius. Using switch case The program should **output the substance** corresponding to the boiling point listed in the table. The program should output the message **“substance unknown”** when it does not match any substance.

Substance	Boiling point
Water	100°C
Mercury	357°C
Copper	1187°C
Silver	2193°C
Gold	2660°C

# Rules to be followed for switch case

- Case doesn't always need to have order 1, 2, 3 and soon. It can have any integer value after case keyword. Also, case doesn't need to be in an ascending order always, you can specify them in any order as per the need of the program.
- Character labels can be used in switch case.
- **Valid expressions for switch**
  - `switch(1+2+23)`
  - `switch(1*2+3%4)`
- **Invalid switch expressions**
  - `switch(ab+cd)`
  - `switch(a+b+c)`

`switch(ab+cd)` is invalid if `ab+cd` does not evaluate to either integer or character or enumeration
- Nesting of switch statements are allowed, which means you can have switch statements inside another switch. However nested switch statements should be avoided as it makes program more complex and less readable.

## Predict the output

```
void main() {  
    int i=2;  
    switch (i) {  
        case 1: printf("Case1 ");  
        case 2: printf("Case2 ");  
        case 3: printf("Case3 ");  
        case 4: printf("Case4 ");  
        default: printf("Default ");  
    }  
    return 0;  
}
```

# Predict the output

- What will be the output of the following code fragment?

```
int year;
```

```
scanf("%d",&year);
```

```
if(year%100==0)
```

```
{ if(year%400==0)
```

```
printf("leap year\n");
```

```
}
```

```
else
```

```
printf("not leap year\n");
```

- if the input given is (i) 2000 (ii) 1900 (iii) 19

## Predict the output

- What will be the output of the following code fragment?

```
int year;  
scanf("%d",&year);  
if(year%100==0)  
{ if(year%400==0)  
  printf("leap year\n");  
}  
else  
  printf("not leap year\n");
```

- if the input given is (i)2000 (ii)1900 (iii) 19
- Ans.
- (i)leap year
- (ii) No output
- (iii) not leap year

## Predict the output

```
void main() {  
    int i=2; switch  
    (i) {  
        case 1: printf("Case1 "); case 2:  
        printf("Case2 "); case 3:  
        printf("Case3 "); case 4:  
        printf("Case4 "); default:  
        printf("Default ");  
    }  
    return 0;  
}
```

## Predict the output

```
void main() {  
    int i=2; switch  
    (i) {  
        case 1: printf("Case1 "); case 2:  
        printf("Case2 "); case 3:  
        printf("Case3 "); case 4:  
        printf("Case4 "); default:  
        printf("Default ");  
    }  
    return 0;  
}
```

**Output: Case2 Case3 Case4 Default**

**Reason: No break statement .It will execute the first matching case and then all the case statements below it.**



## Try it Yourself

- Convert the following if–else loop into switch...case.

```
if ( grade == 'A' || grade == 'a' )  
    ++aCount;  
else if ( grade == 'B' || grade == 'b' )  
    ++bCount;  
else if ( grade == 'C' || grade == 'c' )  
    ++cCount;  
else if ( grade == 'D' || grade == 'd' )  
    ++dCount;  
else {  
    printf( "Incorrect letter grade entered." ); printf( "  
Enter a new grade.\n" );
```

# Summary

- In this lecture we have seen how to alter the flow of a program based on condition.
- The decisions may be two way or multi way.
- The C constructs used for
  - Single way : if
  - Two way : if else
  - Multiway : Nested if else
    - if else if ladder
    - switch case

**THANK YOU**

## Clarifications : last class

- `printf("Price=Rs. %8.2f\n",price);`
- `switch(ab+cd)`

# Examples for float format specifier

```
float a = 1.12 ;  
printf(“ a = %f ”, a );
```

# Examples for float format specifier

```
float a = 1.12 ;  
printf(“ a = %f”, a );
```

Output :- a = 1.12

# Examples for float format specifier

```
a = 1;  
printf(“ a = %f”, a );
```

# Examples for float format specifier

```
a = 1;
```

```
printf(“ a = %f”, a );
```

**Output :- a = 1.000000**



# Examples for float format specifier

```
float a = 3.122222223;  
printf("a = %.2f ", a);
```

# Examples for float format specifier

```
float a = 3.122222223;  
printf("a = %.2f", a);
```

Output :- a = 3.12

# Examples for float format specifier

```
float a = 1.289999;  
printf(" a = %6.2f", a);
```

# Examples for float format specifier

```
float a = 1.289999;  
printf(" a = %6.2f", a);
```

*Output:- \_ \_1.29 where \_ are spaces*

**%6.2f** :- means output will be in 6 columns

Here, a = 1.28999;

So after applying **%6.2f** it will print two spaces before **1** and **.2f** means two digits after dot. Include dot also in count, as it also requires space to get stored.

# Examples for float format specifier

```
float a = 1111.289999;  
printf(" a = %5.3f", a);
```

# Examples for float format specifier

```
float a = 1111.289999;  
printf(" a = %5.3f", a);
```

Output: a = 1111.290

Before dot it contains enough digits than 5 so it will store 1111 in one block and rest in another.

Space will only be added when there are not enough digits.

# Invalid switch expressions

- `switch(ab+cd)`
- `switch(ab+cd)` is invalid if `ab + cd` does not evaluate to either integer or character or enumeration

# **CS2002D PROGRAM DESIGN**

## **Lecture 3**



# Recall Control Structures

- **Control structures** control the flow of execution in a program or function.
- There are three kinds of execution flow:
  - **Sequence:**
    - the execution of the program is sequential. (add/sub/mul/div of two numbers)
  - **Selection:**
    - A control structure which chooses alternative to execute.
  - **Repetition:**
    - A control structure which repeats a group of statements.
- We will today focus on the **repetition** control structure.

# **ITERATIVE CONTROL STRUCTURES**

## **REPETITION**

# Objectives

- ❑ Concept of loop
- ❑ Loop invariant
- ❑ Pretest and post-test loops
- ❑ Initialization and updating
- ❑ Counter controlled loops
- ❑ Event controlled loops

# In detail

- Introduction to iterative construct
- Counter controlled loops
  - While loop
  - Do-while loop
  - For loop
- Nesting of loops
- Control of loop execution
- Infinite loops
- Event controlled loops
  - Sentinel controlled
  - Flag controlled

# Loops

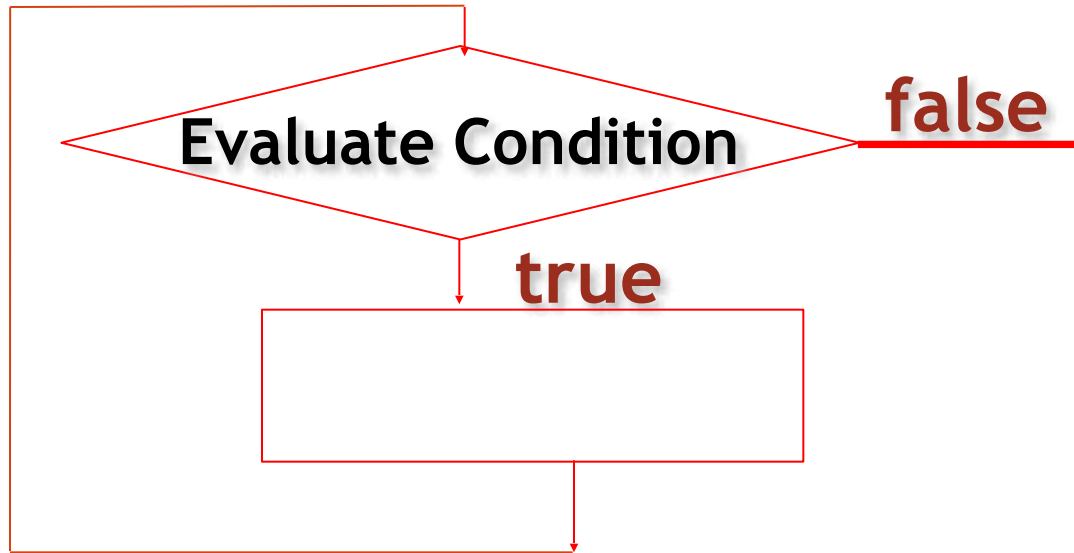
- A loop is a sequence of statements that will be executed repeatedly zero or more times.
  - A loop can be executed a set number of times, or as long as some condition is met.
  - Each single repetition of the loop is known as an iteration of the loop.
- ✓ *Looping until one condition is met is the same as looping as long as the opposite of the condition is met.*
- ✓ *For instance, if you loop until  $x$  equals 5, that is the same as looping as long as  $x$  does not equal 5.*

# Iterative construct: while loop

**while(expression)  
statement s;**

- ✓ *The statement s will be executed as long as the expression remains true, or until a special command is encountered to end the loop.*
- ✓ *The statement s can be a compound statement*

# Looping in a while loop



- ✓ *To repeatedly execute a statement over and over while a given condition is true*
- ✓ *When the condition of the while loop is no longer logically true, the loop terminates and program execution resumes at the next statement following the loop*

# Example #1

```
int x = 3; while (x > 0)
{
    printf("Hello World!\n"); x = x - 1;
}
```

- ✓ The **loop condition** is written first,  
followed by the **body of the loop**
- ✓ The loop condition is evaluated first, and  
if it is true, the loop body is executed
- ✓ After the execution of the loop body, the condition in the while is evaluated again.
- ✓ This repeats until the condition becomes false.



```
int x = 3;
while (x > 0)
{
    printf("Hello World!\n");
    x=x-1;
}
```

❑ This "while" loop will undergo three iterations.

- ✓ During each, the phrase "Hello World!"
- ✓ will be printed on a separate line.

❑ Why does it execute three times?

- ✓ The variable "x" is initialized above the loop with the value of 3.
- ✓ The loop will repeat as long as the expression "x > 0" is true.
- ✓ At the end of each loop iteration, "x" is decreased by 1.
- ✓ After three iterations, "x" will have the value of 0, and the expression will no longer be true, so the loop will end.
- ✓ How many times the condition "x > 0" will be checked?

❑ Note that there is **no semicolon** after the right parenthesis ending the expression that "while" is checking.

- ✓ If there were, it would mean that the program would **repeat the null statement** (statements that do nothing) until the condition were not true.
- ✓ The condition starts off true, it will stay true, and will loop without stopping...

## Example #2

```
int x, y;  
printf("Enter two numbers: ");  
scanf("%d %d", &x, &y);  
while (y != 0)  
{   printf("%d / %d = %d\n", x, y, x/y);  
    printf("Enter two numbers: ");  
    scanf("%d %d", &x, &y);  
}
```

- ✓ *This code repeatedly asks the user to enter two integers.*
- ✓ *As long as the second number is not zero, the program prints the result of dividing the first number by the second.*
- ✓ *If the second number is 0, the program ends.*

## Example #2: Try it yourself

```
int i = 0;
```

```
int loop_count = 5;
```

```
printf("Case1:\n");
```

```
while (i<loop_count) {  
    printf("%d\n",i); i++; }
```

```
printf("Case2:\n");
```

```
i=20;
```

```
while (0) {  
    printf("%d\n",i); i++; }
```

## Example #3 *Contd...*

```
printf("Case3:\n");
```

```
i=0;
```

```
while (i++<5) {  
    printf("%d\n",i); }
```

```
printf("Case4:\n");
```

```
i=3;
```

```
while (i < 5 && i >=2) {  
    printf("%d\n",i); i++; }
```

## Cases:1 and 2

- **Case1 (Normal)** : Variable 'i' is initialized to 0 before 'while' loop; iteration is increment of counter variable 'i'; condition is execute loop till 'i' is lesser than value of 'loop\_count' variable i.e. 5.
- **Case2 (Always FALSE condition)** : Variables 'i' is initialized before 'while' loop to '20'; iteration is increment of counter variable 'i'; condition is FALSE always as '0' is provided that causes NOT to execute loop statements and loop statement is NOT executed.
- ✓ Here, it is noted that as compared to 'do-while' loop, statements in 'while' loop are NOT even executed once which executed at least once in 'do- while' loop because 'while' loop only executes loop statements only if condition succeeds.

## Cases:3 and 4

- **Case3 (Iteration in condition check expression)** :*Variable 'i' is initialized to 0 before 'while' loop; here note that iteration and condition is provided in same expression. Here, observe the condition is execute loop till 'i' is lesser than '5' and loop iterates 5 times.*
  - ✓ *Unlike 'do-while' loop, here condition is checked first then 'while' loop executes statements.*
- **Case4 (Using logical AND condition)** :*Variable 'i' is initialized before 'while' loop to '3'; iteration is increment of counter variable 'i'; condition is execute loop when 'i' is lesser than '5' AND 'i' is greater or equal to '2'.*

# # ./a.out

- Case1: 0 1 2 3 4
- Case2:
- Case3: 1 2 3 4 5
- Case4: 3 4 #

*Find any differences ?*

```
int x = 3;  
while (x-- > 0)  
    printf("Hello World!\n");
```

```
int x = 3;  
while (--x >= 0)  
    printf("Hello World!\n");
```



## *Find any differences ?*

```
int x = 3;  
while (x-- > 0)  
    printf("Hello World!\n");
```

- ✓ *Here, since the decrement operator is placed after the variable, the old value of the variable is used to compare against 0.*
- ✓ *The first time through, this value is 3, then 2, then 1.*
- ✓ *The fourth time, it is 0 so we exit the loop.*

```
int x = 3;  
while (--x >= 0)  
    printf("Hello World!\n");
```

- ✓ *Now, the decrement operator is placed before the variable, so the value of "x" is decreased and then its value is used.*
- ✓ *The first time through, this value is 2, then 1, then 0.*
- ✓ *Then, it is -1 so we exit the loop.*

# Write a program....

- Ask the user to enter a number, and if it is positive, sum the digits.
- The user enters a number, and the value is stored in "x".
- Store sum in the variable "sum\_digits", which is initialized to zero.

## ❖ *Hint:*

- ✓ *As long as "x" is greater than zero, we mod it by 10, which gets the right-most digit of the number, and we add this digit to "sum\_digits".*
- ✓ *Then we divide "x" by 10. Remember, when we do integer division, the fractional part is cut off, so in effect, we are removing the right-most digit of "x" (which we have already added to the sum).*

# Solution

```
int x, digit, sum_digits;
printf("Enter a positive integer: ");
scanf("%d", &x);
sum_digits = 0;
while (x > 0)
{
    digit = x % 10;
    sum_digits = sum_digits + digit;
    x = x / 10;
}
printf("The sum of the digits is %d!\n",
sum_digits);
```

# Iterative construct: do - while loop

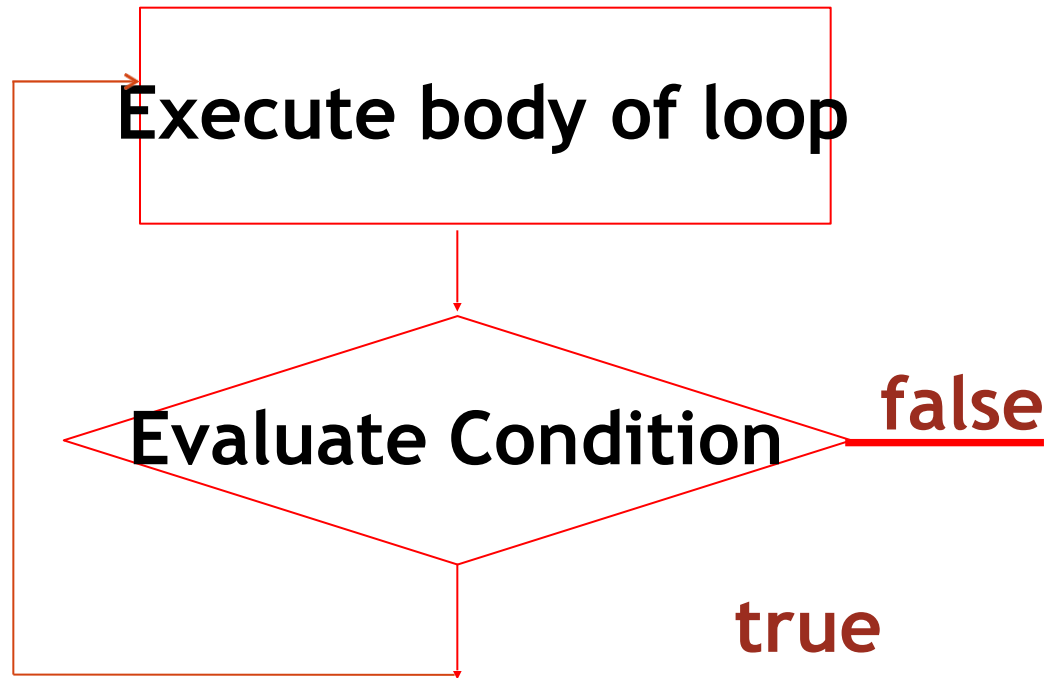
**do**

**statement**

**while(condition);**

✓ *It is similar to the "while" statement, but the condition is checked at the end.*

# Looping in a do-while loop



✓ *Statements inside the statement block are executed once, and then expression is evaluated, in order to determine whether the looping is to continue*

## Example #1

```
int i = 0;
```

```
do{
```

```
    printf("The value of i  %d \n",i);
```

```
    i = i + 1;
```

```
} while (i < 5);
```

```
while (i < 5);
```

- ✓ *The body of the loop comes first, followed by the loop condition at the end*
- ✓ *The loop is entered into straightaway, and after the first execution of the loop body, the loop condition is evaluated*
- ✓ ***The body of the loop is guaranteed to execute at least once***  
*The body of the loop is guaranteed to execute at least once*
- ✓ *Further executions of loop body would be subject to loop condition evaluating to true*

## Example #2

```
int x = 3;  
do  
{  
    printf("Hello World!\n");  
    x = x-1;  
} while (x > 0);
```

✓ Notice that there is a semicolon after the right parenthesis ending the expression while is checking.

✓ If you forget it, you will get a compiler error when you try to compile the program.

## Example #3

```
do
{
    printf("Enter a number from 1 to 100:");
    scanf("%d", &x);
} while ((x < 1) || (x > 100));
```

✓ *If the user does not enter a number in the correct range, the “while” condition will be met, and the loop will undergo another iteration, prompting the user again.*

✓ *Only after the user enters a number from 1 to 100 will the expression be false and the loop end.*



## Example #4: Try it yourself

```
int i = 0;  
int loop_count = 5;  
printf("Case1:\n");  
do {  
    printf("%d\n",i); i++;  
} while (i<loop_count);  
  
printf("Case2:\n");  
i=20;  
do {  
    printf("%d\n",i); i++;  
} while (0);
```

## Example #4 *Contd...*

```
printf("Case3:\n");
```

```
i=0;
```

```
do {
```

```
    printf("%d\n",i);
```

```
} while (i++<5);
```

```
printf("Case4:\n");
```

```
i=3;
```

```
do {
```

```
    printf("%d\n",i); i++;
```

```
} while (i < 5 && i >=2);
```

## Cases: 1 and 2

- **Case1 (Normal)** : *Variable 'i' is initialized to 0 before 'do-while' loop; iteration is increment of counter variable 'i'; condition is to execute loop till 'i' is lesser than value of 'loop\_count' variable i.e. 5.*
  - **Case2 (Always FALSE condition)** : *Variables 'i' is initialized before 'do-while' loop to '20'; iteration is increment of counter variable 'i'; condition is FALSE always as '0' is provided that causes NOT to execute loop statements.*
- ✓ *But, it is noted here in output that loop statement is executed once because do-while loop always executes its loop statements at least once even if condition fails at first iteration.*

## Cases: 3 and 4

- **Case3 (Iteration in condition check expression)** : *Variable 'i' is initialized to 0 before 'do-while' loop; here note that iteration and condition is provided in same expression.*
  - ✓ *Here, observe the condition is to execute loop till 'i' is lesser than '5', but in output 5 is also printed that is because, here iteration is being done at condition check expression, hence on each iteration 'do-while' loop executes statements ahead of condition check.*
- **Case4 (Using logical AND condition)** : *Variable 'i' is initialized before 'do-while' loop to '3'; iteration is increment of counter variable 'i'; condition is execute loop when 'i' is lesser than '5' AND 'i' is greater or equal to '2'.*

# # ./a.out

- Case1: 0 1 2 3 4
- Case2: 20
- Case3: 0 1 2 3 4 5
- Case4: 3 4 #

# Example #5: Programs with Menus

A)dd part to catalog  
R)emove part from catalog  
F)ind part in catalog  
Q)uit

**Select option: A**

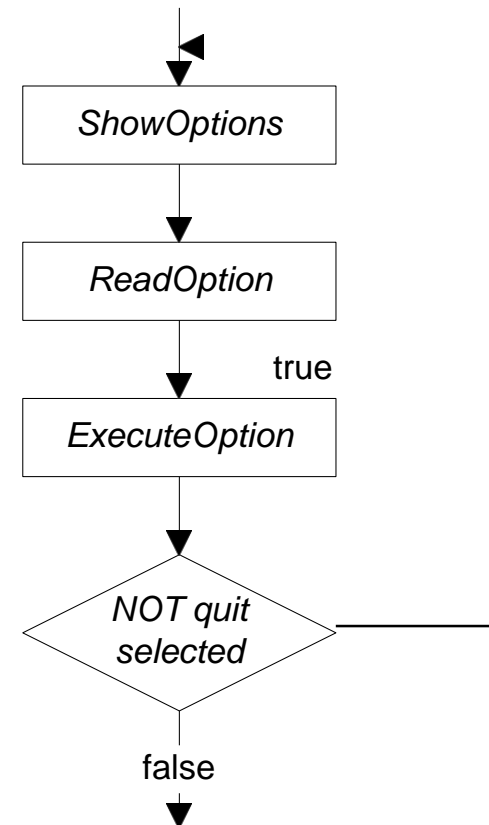
*<interaction to add a part>*

A)dd part to catalog  
R)emove part from catalog  
F)ind part in catalog  
Q)uit

**Select option: *<next option>***

# Menu Loop

```
do {  
    showOptions();  
    printf("Select option:");  
    scanf(" %c",&optn);  
    execOption(optn);  
} while (!((optn == 'Q') || (optn == 'q')));
```



# Menu Options

```
void showOptions() {  
    printf("A)dd part to catalog\n");  
    printf("R)emove part from catalog\n");  
    printf("F)ind part in catalog\n");  
    printf("Q)uit\n");  
}
```



# Executing Options

```
void execOption( char option ) {  
    switch (option) {  
        case 'A': case 'a': addPart(); break;  
        case 'R': case 'r': delPart(); break;  
        case 'F': case 'f': fndPart(); break;  
        case 'Q': case 'q': break;  
        default: printf("Unknown option  
            %c\n",option); break;  
    }  
}
```

# Iterative construct: for loop

```
for (expr1; expr2;expr3)
{
    statement1;
    statement2; . . .
}
```

- ✓ *The for loop construct is by far the **most powerful and compact** of all the loop constructs provided by C.*
- ✓ *This loop keeps **all loop control statements on top of the loop**, thus making it visible to the programmer.*
- ✓ *This loop works well where **the number of iterations of the loop is known before the loop is entered into.***

# for (initialization; condition; update)

✓ The **initialization(expression1)** is usually an assignment of a variable to some starting value, and the **update (expression3)** is often an assignment which changes this variable.

✓ The statement will be executed as long as the **condition(expression2)**, which is an expression, is true.

✓ All three fields are optional.

✓ If the initialization or update are left out, they are considered null statements (statements that do nothing).

✓ If the condition is left out, it is considered to be always true, and the loop will continue until a statement is reached to break out of the loop.

## **The first part :**

- **Expression 1:** is executed before the loop is entered
- ✓ This is usually the initialization of the loop variable

## **The second part :**

- **Expression 2:** is a test, is evaluated immediately after expression1, and then later is evaluated again after each successful looping
- ✓ The loop is terminated when this test returns a false

## **The third part :**

- **Expression 3:** is a statement to be run every time the loop body is completed
- ✓ **It is not evaluated when the for statement is first encountered.**  
However, expression3 is evaluated after each looping and before the statement goes back to test expression2 again.
- ✓ This is usually an increment of the loop counter

# for loop....

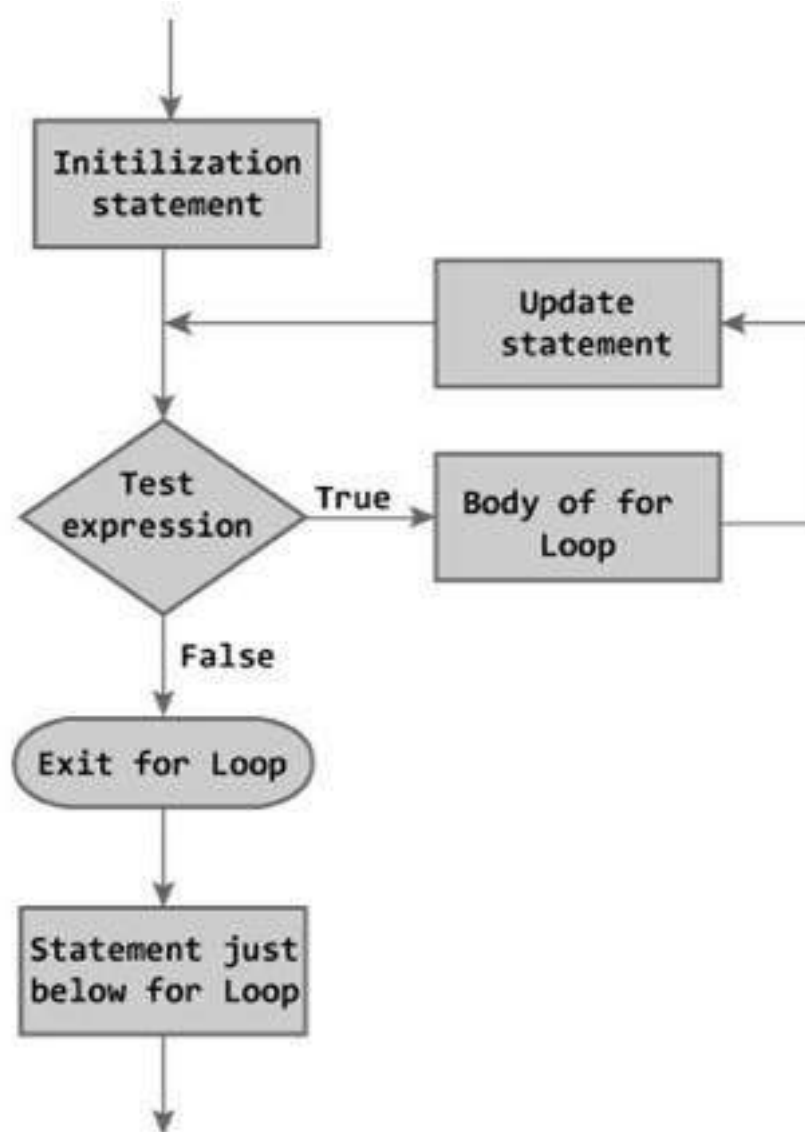


Figure: Flowchart of for Loop

# Example #1

```
int n, count, sum=0;
printf("Enter the value of n.\n");
scanf("%d",&n);
for(count=1;count<=n;++count) //for loop terminates if count>n
{
    sum+=count; // this statement is equivalent to sum=sum+count
}
printf("Sum=%d",sum);
return 0;
```

- ✓ *In this program, the user is asked to enter the value of n.*
- ✓ *Suppose you entered 19 then, count is initialized to 1 at first.*
- ✓ *Then, the test expression in the for loop, i.e., (count <= n) becomes true.*
- ✓ *So, the code in the body of for loop is executed which makes sum to 1.*

- ✓ Then, the expression `++count` is executed and again the test expression is checked, which becomes true.
- ✓ Again, the body of for loop is executed which makes sum to 3 and this process continues.
- ✓ When count is 20, the test condition becomes false and the for loop terminated.
- **Note:** Initial, test and update expressions are separated by semicolon(;).

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

```
for(count=1;count<=n;++count) //for loop terminates if count>n
```

```
{sum+=count;}
```

```
printf("Sum=%d",sum);
```

## Example #2 Try it yourself

```
int i = 0, k = 0; float j = 0;
```

```
int loop_count = 5;
```

```
printf("Case1:\n");
```

```
for (i=0; i < loop_count; i++) {  
    printf("%d\n",i); }
```

```
printf("Case2:\n");
```

```
for (j=5.5; j > 0; j--) {  
    printf("%f\n",j); }
```

```
printf("Case3:\n");
```

```
for (i=2; (i < 5 && i >=2); i++) {  
    printf("%d\n",i); }
```



## Example #2 *Contd...*

```
printf("Case4:\n");  
for (i=0; (i != 5); i++) {  
    printf("%d\n",i); }
```

```
printf("Case5:\n");  
/* Blank loop */ for (i=0; i < loop_count; i++) ;
```

```
printf("Case6:\n");  
for (i=0, k=0; (i < 5 && k < 3); i++, k++) {  
    printf("%d\n",i); }
```

```
printf("Case7:\n");  
i=5;  
for (; 0; i++) { printf("%d\n",i); }
```

# Cases: 1,2 and 3

- **Case1 (Normal)** : Variable 'i' is initialized to 0; condition is to execute loop till 'i' is lesser than value of 'loop\_count' variable; iteration is increment of counter variable 'i'
- **Case2 (Using float variable)** : Variable 'j' is float and initialized to 5.5; condition is to execute loop till 'j' is greater than '0'; iteration is decrement of counter variable 'j'.
- **Case3 (Taking logical AND condition)** : Variable 'i' is initialized to 2; condition is to execute loop when 'i' is greater or equal to '2' and lesser than '5'; iteration is increment of counter variable 'i'.

## Case : 4,5,6 and 7

- **Case4 (Using logical NOT EQUAL condition)** : *Variable 'i' is initialized to 0; condition is to execute loop till 'i' is NOT equal to '5'; iteration is increment of counter variable 'i'.*
- **Case5 (Blank Loop)** : *This example shows that loop can execute even if there is no statement in the block for execution on each iteration.*
- **Case6 (Multiple variables and conditions)** : *Variables 'i' and 'k' are initialized to 0; condition is to execute loop when 'i' is lesser than '5' and 'k' is lesser than '3'; iteration is increment of counter variables 'i' and 'k'.*
- **Case7 (No initialization in for loop and Always FALSE condition)** : *Variables 'i' is initialized before for loop to '5'; condition is FALSE always as '0' is provided that causes NOT to execute loop statement; iteration is increment of counter variable 'i'.*

# # ./a.out

- Case1: 0 1 2 3 4
- Case2: 5.500000 4.500000 3.500000 2.500000  
1.500000 0.500000
- Case3: 2 3 4
- Case4: 0 1 2 3 4
- Case5:
- Case6: 0 1 2
- Case7:

# *Predict the output*



1.

```
int i;
```

```
for (i=0; i<16; i++)
```

```
    printf(“%X %x %d\n”, i, i, i);
```

2.

```
for (i=0; i<8; i++)
```

```
sum += i;
```

3.

```
for (i=0; i<8; i++);
```

```
sum += i;
```

**THANK YOU**

# **CS2002D PROGRAM DESIGN**

## **Lecture 4**

# Recall Control Structures

- **Control structures** control the flow of execution in a program or function.
- There are three kinds of execution flow:
  - **Sequence:**
  - **Selection:**
  - **Repetition:**
- Last class, we have discussed the repetitive structures: **while**, **do while** and few details on **for**
- We will today focus on **for**, **nested loops**, **break**, **continue**, **event controlled loops**
- We will also discuss Data Types



# for loop....

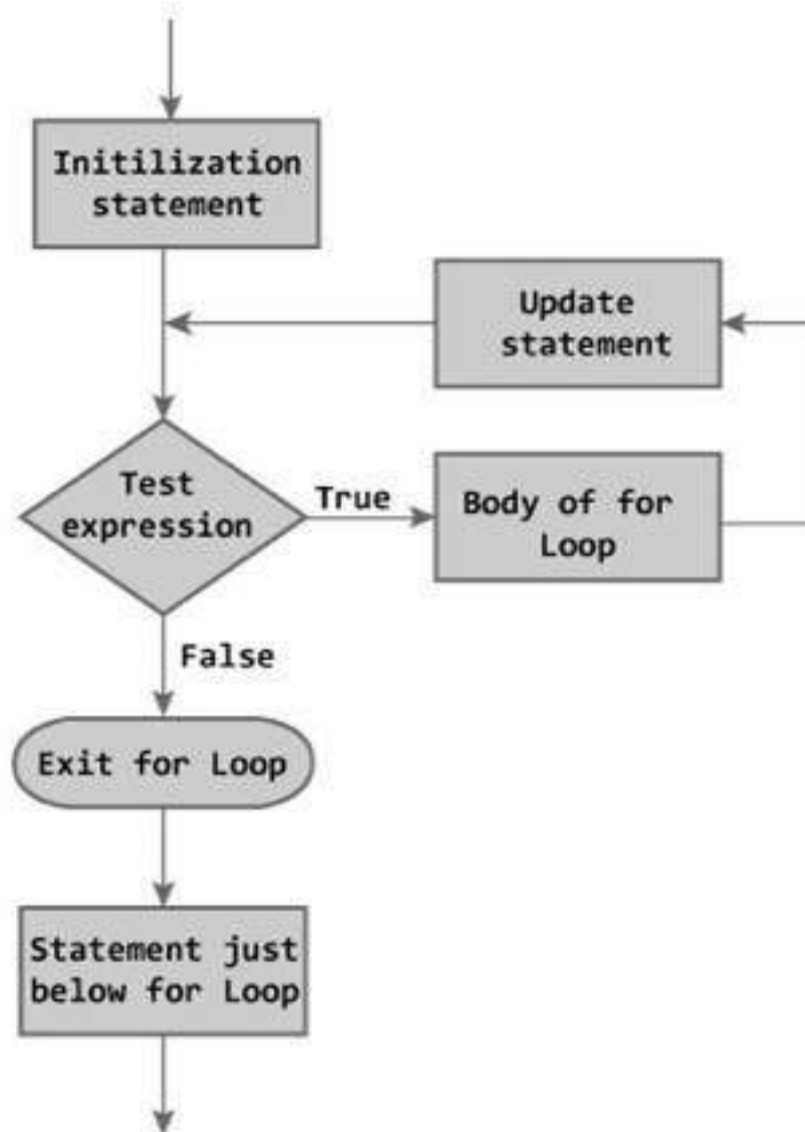


Figure: Flowchart of for Loop

# Conversion

✓ *Anything that can be done with a “for” statement can also be done with an equivalent “while” statement.*

✓ **for (initialization; condition; update)  
statement;**

- The conversion is :

```
initialization;  
while (condition)  
{  
    statement;  
    update;  
}
```

# Comma operator to combine multiple expressions in for loop

**Expr1:** *integer variables **i** and **j** are initialized, respectively, with 0 and 10 when the for statement is first encountered.*

```
for (i=0, j=10; i!=j; i++, j--)  
{  
  
    /* statement block */  
  
}
```

**Expr2:** *relational expressions **i!=j** is evaluated and tested. If it evaluates to zero (false), the loop is terminated.*

**Expr3:** *After each iteration of the loop, **i** is increased by 1 and **j** is reduced by 1.*

*Then the expression **i!=j** is evaluated to determine whether or not to execute the loop again.*

# Nesting of loops

- ✓ *Sometimes, within compound statements that are part of loops, there will be additional loops.*
- **Loops within loops are referred to as nested loops.**
- ✓ *There are outer loops (the loop encountered first) and inner loops (any loops embedded within an outer loop).*

# Nested loops.....While loop

initialize outer loop

while (outer loop condition )

{     ...

    initialize inner loop

    while (inner loop condition )

    {

        inner loop processing and update

    }

    ...

}

## Nested loops .... for loop

```
for (i=1; i<=3; i++) /* outer loop */  
{  
    printf("Start of iteration %d of the outer loop. \n", i);  
  
    for (j=1; j<=4; j++) /* inner loop */  
        printf(" Iteration %d of inner loop.\n",j);  
  
    printf("End of iteration %d of outer loop.\n", i);  
}
```

## Example #1 :

```
int x, sum_digits, digit, temp;
for (x = 1; x <= 1000; x++)
{
    temp = x;
    sum_digits = 0;
    while (temp > 0)
    {
        digit = temp % 10;
        sum_digits = sum_digits + digit;
        temp = temp / 10;
    }
    if (sum_digits == 5)
        printf("%d\n", x);
}
```

## Example #2 : What does the code do?

```
int row, col;  
printf("\t0\t1\t2\t3\t4\t5\t6\t7\t8\t9\n");  
for (row = 0; row <= 9; row++)  
{  
    printf("%d", row);  
    for (col = 0; col <= 9; col++)  
    {  
        printf("\t%d", row*col);  
    }  
    printf("\n");  
}
```



# Screenshot of Output

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

...Program finished with exit code 0  
Press ENTER to exit console.

- ✓ *Using the '\t' symbol within a string passed to "printf" causes a tab to be printed, and the computer skips to the start of the next 8 character column.*
- ✓ *The '\t' symbol is a special way of representing the tab character, in the same way that the '\n' symbol is a special way of representing the newline character.*
- ✓ *The first "printf" in this program skips the first column (since no characters appear to the left of the first tab), then prints out column headers 0 through 0 in the next 9 columns.*

- ✓ *We then loop through 9 rows of the table.*
- ✓ *At the beginning of each row, we print the row number.*
- ✓ *Then, we loop through 9 columns, and for each, we tab over to the column and print the product of the row number and column number.*
- ✓ *After the inner “for” loop (at the end of each iteration of the outer “for” loop), we print a ‘\n’ which causes the program to start the next line.*

*Trace the nested loop given*

```
printf("Max N! to print: ");
scanf("%d",&N);
for (I = 1; I <= N; I++) {
    fact = 1;
    for (J = 2; J <= I; J++)
        fact *= J;
    printf("%d! = %d\n",I,fact);
}
```

# Tracing.....

Stmt	N	I	J	fact	output
1	4				
2		1			
3				1	
4			2		
6					1! = 1
2		2			
3				1	
4			2		
5				2	
4			3		
6					2! = 2
2		3			
3				1	
4			2		
5				2	

Stmt	N	I	J	fact	output
4			3		
5				6	
4			4		
6					3! = 6
2		4			
3				1	
4			2		
5				2	
4			3		
5				6	
4			4		
5				24	
4			5		
6					4! = 24
2		5			

# Control of loop execution:

## Break Statement

- A loop construct, whether while, or do-while, or a for loop continues to **iteratively execute until the loop condition evaluates to false**
- There may be situations where it may be necessary **to exit from a loop even before the loop condition is reevaluated after an iteration**
- The **break statement** is used **to exit early** from all loop constructs (while, do-while, and for)

# Example #1

```
int x, y;  
while (1)
```

```
{
```

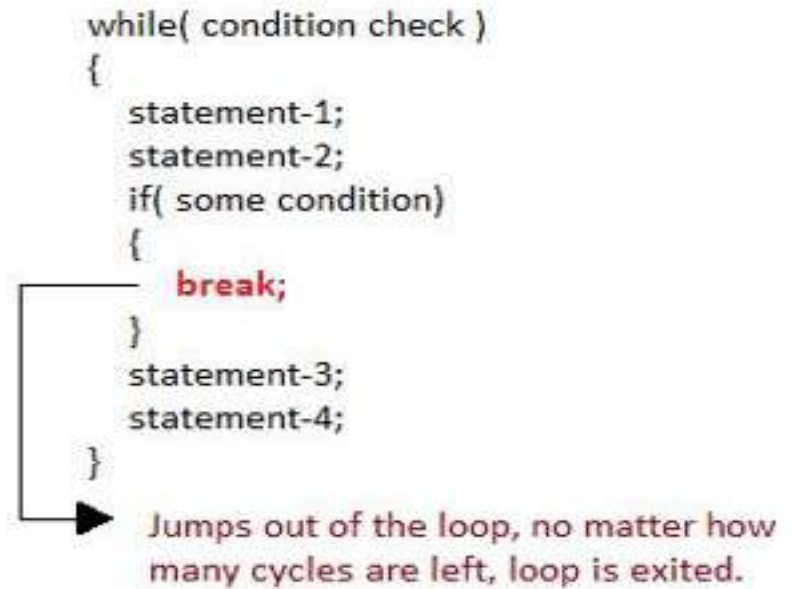
```
    printf("Enter two numbers: ");
```

```
    scanf("%d %d", &x, &y);
```

```
    if (y == 0) break;
```

```
        printf("%d /%d = %d\n", x, y, x/y);
```

```
}
```




- ✓ *When you use "while(1)" (or any other non-zero constant), the loop should continue until some special statement inside the loop stops it.*
  - ✓ *Remember, when a non-zero constant is used as a boolean expression, it is interpreted as true.*
- ✓ *When the **"break"** statement is reached, the computer jumps to the first statement after the "while" loop, regardless of whether or not the condition (expression) being checked by the loop is true.*
- ✓ *The program goes into the loop no matter what and asks the user to enter the two numbers at the start of each iteration of the loop.*
- ✓ *When the user enters 0 as the second number, the loop is exited, and the program ends.*



## Example #2

✓ *Break* - terminates loop, execution continues with the first statement following the loop- for and while loops

```
sum = 0;
for (k=1; k<=5; k++)
{
    scanf("%lf",&x);
    if (x > 10.0)
        break;
    sum += x;
}
printf("Sum = %f\n",sum);
```



```
sum = 0;
k=1;
while (k<=5)
{
    scanf("%lf",&x);
    if (x > 10.0)
        break;
    sum += x;
    k++;
}
printf("Sum = %f\n",sum);
```

# Break statement....

- *If you encounter a “break” statement in the middle of a nested loop, the control of the program jumps to the first statement after the innermost loop surrounding the “break” statement.*
- ✓ *For instance, let’s say in the Example #2 of nested loops, you only want to print out half of the multiplication table, that below the diagonal line from the top left to the bottom right.*
- ✓ *There is no need to print the result of  $2*7$  if you are going to print the result of  $7*2$  anyway!*

# *Multiplication table program*



- ✓ *There is no need to print the result of  $2*7$  if you are going to print the result of  $7*2$  anyway!*
- ✓ *Can you can make this adjustment by adding one “if ” statement to our multiplication program:?*

# *Multiplication table program revisited*

```
int row, col;
printf("\t0\t1\t2\t3\t4\t5\t6\t7\t8\t9\n");
for (row = 0; row <= 9; row++)
{
    printf("%d", row);
    for (col = 0; col <= 9; col++)
    {
        printf("\t%d", row*col);
        if (col == row)
            break;
    }
    printf("\n");
}
```

✓ Now, for each row, once the column equals the row, we skip the rest of the inner “for” loop and jump to the line that prints the newline character. We then move on to the next row!

# Screenshot of the output

```

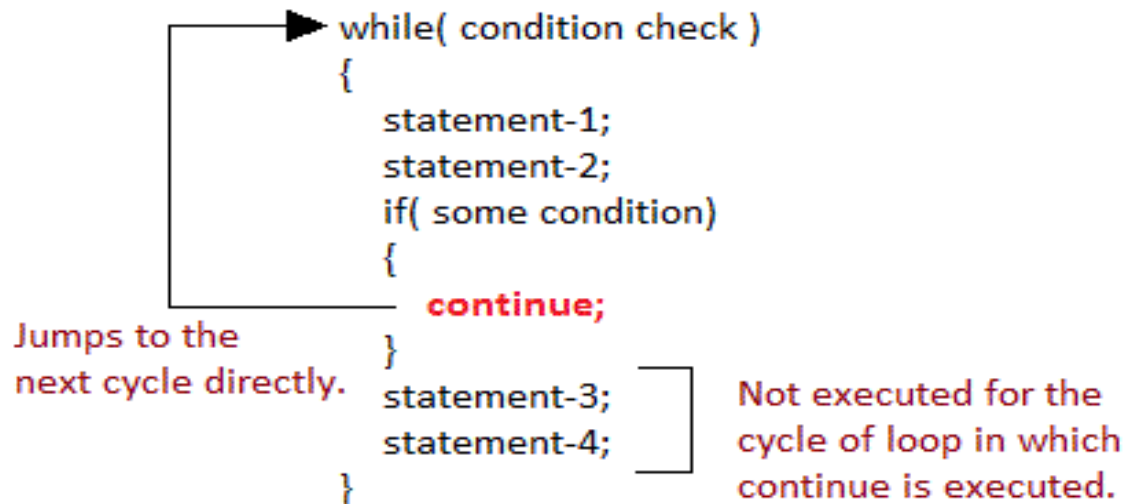
    0      1      2      3      4      5      6      7      8
0      0
1      0      1
2      0      2      4
3      0      3      6      9
4      0      4      8     12     16
5      0      5     10     15     20     25
6      0      6     12     18     24     30     36
7      0      7     14     21     28     35     42     49
8      0      8     16     24     32     40     48     56     64
9      0      9     18     27     36     45     54     63     72

...Program finished with exit code 0
Press ENTER to exit console.
```

# Control of loop execution:

## Continue Statement


- The `continue` statement causes **all subsequent instructions in the loop body** (coming after the `continue` statement) **to be skipped**
- **Control passes back to the top of the loop** where the loop condition is evaluated again
  - ✓ *In case of a `continue` statement in a `for` loop construct, control passes to the reinitialization part of the loop, after which the loop condition is evaluated again.*



# Example #1

✓ ***Continue** forces next iteration of the loop, skipping any remaining statements in the loop- for and while loops*

```
sum = 0;
for (k=1; k<=5; k++)
{
    scanf("%lf",&x);
    if (x > 10.0)
        continue;
    sum +=x;
}
printf("Sum = %f \n",sum);
```



```
sum = 0;
k=1;
while (k<=5)
{
    scanf("%lf",&x);
    if (x > 10.0)
        continue;
    sum +=x;
    k++;
}
printf("Sum = %f \n",sum);
```

## Example #2

```
int x;  
for (x = 1; x <= 20; x++)  
{  
    if (x % 5 == 0)  
        continue;  
    printf("%d\n", x);  
}
```

✓ The first thing to note here is the test for divisibility by 5.

✓ Remember that the "%" is the modulus operator; it returns the remainder when the first operand is divided by the second operand.

✓ ‘

✓ If the remainder when "x" is divided by 5 is 0, then "x" is divisible by 5!

✓ When x is not divisible by 5, the condition of the "if" statement is false, so we reach the "printf" statement and print out the number.

✓ When "x" is 5, 10, or 15, the condition of the "if" statement is true, so we "continue", or skip, to the end of the current iteration of the "for" loop, still do the update "x++", and start the next iteration of the loop.

✓ When "x" is 20, we skip to the end of the current iteration of the "for" loop, still do the update "x++", but now x will be 21, so the condition of the "for" loop is no longer met, and we end the loop.



## Example #3

```
int c;  
printf("Enter a character:\n(enter x to exit)\n");  
while {  
    c = getch();  
    if (c == 'x ')  
        break;  
}  
printf("Break the infinite while loop.Bye!\n");
```

**Enter a character:  
(enter x to exit)**

**H**

**I**

**x**

**Break the infinite while loop.  
Bye!**

## *Rewrite the code using continue statement*

```
for (I = 0; I < 100; I++)  
{  
    if (!(I % 2) == 1)  
        printf("%d is even", I);  
}
```

## *Rewrite the code using continue statement*

```
for (I = 0; I < 100; I++)  
{  
    if (!(I % 2) == 1)  
        printf("%d is even", I);  
}
```

```
for (I = 0; I < 100; I++)  
{  
    if ((I % 2) == 1)  
        continue;  
    printf("%d is even", I);  
}
```

# Infinite loops

```
for ( ; ; )  
{  
    statement1;  
    statement2;  
    ..  
    .  
}
```

```
while {  
    statement1;  
    statement2;  
    ..  
    .  
}
```



# Example #1

✓ Now consider the following program:

```
while (1)
{
    printf("Hello World!\n");
}
```

✓ *It prints "Hello Wold!" forever!*

✓ *This is called an infinite loop. Here, we created one on purpose, but normally, **they are created by accident.***

- *What do you do when you are caught in an infinite loop?*
  - You press **Ctrl-C** on your keyboard.
  - **Pressing Ctrl-C will halt the execution of your C program!**
- *What happens if you use "while(0)" in your program?*
  - The loop is skipped no matter what. It is useless, but valid.

## *Here are five ways to exit a loop:*

- ✓ The condition the loop depends on is not met at the time of the check.*
- ✓ A "break" statement is encountered.*
- ✓ A statement that ends the current function, such as "return", is encountered.*
- ✓ The program crashes.*
- ✓ The user presses Ctrl-C.*

# Event controlled loops

- ✓ *read until input ends*
- ✓ *read until a number encountered*
- ✓ *search through data until item found*

- **Sentinel controlled** Keep processing data until a special value (which is not a possible data value ) is entered to indicate that processing should stop
- **End-of-file controlled** Keep processing data as long as there is more data in the file
- **Flag controlled** Keep processing data until the value of a flag changes in the loop body



# Example #1

✓ *Sentinel is negative blood pressure.*

```
int thisBP;  
int total;  
int count;  
count = 1;  
total = 0;
```

```
scanf( "%d", &thisBP);
```

```
while (thisBP > 0)           // Test expression  
{  
    total = total + thisBP;  
    count++;                 // Update  
}
```

```
printf("The total = %d", total);
```

## Example #2

- ✓ The **value -999** is sometimes referred to as a ***sentinel value***
- ✓ The value serves as the “guardian” for the termination of the loop
- ✓ Often a good idea to make the sentinel a constant

```
#define STOPNUMBER -999  
while (number != STOPNUMBER)  
...
```

## Example#3

✓ **done** is the flag, it controls the looping

```
total = 0;
done = 0; /* done is set to 0 state */
do {
    scanf("%d",&num);
    if (num < 0) done = 1; /* done 1
    */
} while ((num != 0) && (!done));
```

# Loop Testing and Debugging

- Test data should test all sections of the program
- Beware of infinite loops -- the program doesn't stop
- Check loop termination condition
- Use algorithm walk-through to verify that appropriate conditions occur in the right places
- Trace execution of loop by hand with code walk-through
- Use a debugger (if available) to run program in “slow motion” or use debug output statements

# Data types in C

# Data Types

- Data types in C is a system used for declaring variables or functions of different types.
- The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

# The data types in C can be classified as follows:

Sr.No.	Types & Description
1	<b>Basic Types</b> They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.
2	<b>Enumerated types</b> They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
3	<b>The type void</b> The type specifier <i>void</i> indicates that no value is available.
4	<b>Derived types</b> They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

# Basic Types : Integer types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

- To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator.
- The expressions *sizeof(type)* yields the storage size of the object or type in bytes



# Basic Types : Floating point types

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

# Enumerated types

- Enumeration is a user defined datatype in C language.
- It is used to assign names to the integral constants which makes a program easy to read and maintain.
- The keyword “enum” is used to declare an enumeration.
- Syntax of enum in C language:
- *enum enum\_name{const1, const2, ..... };*

# Example of enumerated data types

*/\* Declaration \*/*

***enum sports***

***{***

*cricket, football, hockey, tennis*

***};***

*/\* Define two enums \*/*

***enum sports n1, n2;***

*/\* Assign enum values \*/*

*n1 = cricket;*

*n2 = football;*

# The void Type

The void type specifies that no value is available. It is used in three kinds of situations:

Sr.No.	Types & Description
1	<b>Function returns as void</b> There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, <b>void exit (int status);</b>
2	<b>Function arguments as void</b> There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, <b>int rand(void);</b>
3	<b>Pointers to void</b> A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function <b>void *malloc( size_t size );</b> returns a pointer to void which can be casted to any data type.

# Derived Types

- Array
- Structure
- Union
- Function
- Pointer

# Declaration and initialization of a single dimensional array

```
#include<stdio.h>
int main(){
int i=0;
int marks[5]={ 20,30,40,50,60};//declaration and initialization of
array.
//traversal of array.
for(i=0;i<5;i++){
printf("%d \n",marks[i]);
}
```

# Declaration and initialization of a two dimensional array

```
#include<stdio.h>

int main(){
int i=0,j=0;
int arr[4][3]={ { 1,2,3},{2,3,4},{3,4,5},{4,5,6} };
//traversing 2D array
for(i=0;i<4;i++){
    for(j=0;j<3;j++){
        printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
    }//end of j
} //end of i
return 0;
}
```

# Declaration and initialization of *Structure*

```
struct Point
{
    int x, y, z;
};
```

```
int main()
{
    // Examples of initialization using designated initialization
    struct Point p1 = {.y = 0, .z = 1, .x = 2};
    struct Point p2 = {.x = 20};

    printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
    printf ("x = %d", p2.x);
    return 0;
}
```



# Brief introduction on pointers

- What is a pointer?
- Pointer is a variable that contains the address of a variable
- How do we declare a pointer variable?
- Syntax of pointer declaration:  
`data_type *pointer_name;`
- Example: `int *ptr;`
- `int a;` //declaration of an integer  
`int *ptr;` //declaration of a **pointer variable** which points to an integer variable  
`ptr = &a;` //assigning the address of variable to the pointer **variable—initialization of a pointer variable**

# Pointers

- \* operator : indirection or dereferencing operator
- Every pointer points to a specific data type
- Does not point to an expression

```
void main() {  
    int i=10;  
    printf("\nValue of :%d" ,i);  
    printf("\nAddress of i :%d",&i); }  

```

# Predict the output

```
main(){
    int x=10; int y=20;
    // variable ptr1 is declared as a pointer variable
    int *ptr1=NULL; //it is initialized as 0
    printf( " First: %d %d %d", x,y, ptr1);
    //ptr1 is assigned x's address
    ptr1 = &x;
    // y is assigned the value ptr1 is pointing to
    y = *ptr1;
    printf( "Second: %d %d %d", x,y, *ptr1);
    // value at ptr1 is now 0
    *ptr1 =0;
    printf( "Third: %d %d %d", x,y, *ptr1); }
```

# Predict the output

```
main(){
    int x=10; int y=20;
    // variable ptr1 is declared as a pointer variable
    int *ptr1=NULL; //it is initialized as 0
    printf( " First: %d %d %d", x,y, ptr1);
    //ptr1 is assigned x's address
    ptr1 = &x;
    // y is assigned the value ptr1 is pointing to
    y = *ptr1;
    printf( "Second: %d %d %d", x,y, *ptr1);
    // value at ptr1 is now 0
    *ptr1 =0;
    printf( "Third: %d %d %d", x,y, *ptr1); }
```

- Ans: First: 10 20 0 Second:10 10 10 Third: 0 10 0

Thank You

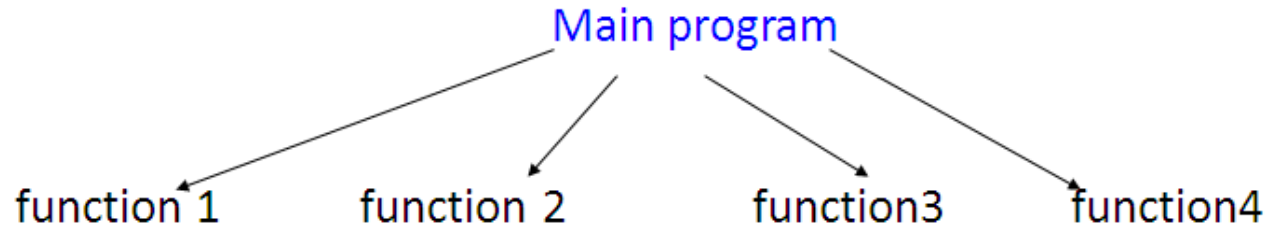
# Functions

# Objectives

- To understand the concept of **modularization**.
- To know about the **types of functions**.
- To study about **formal arguments and actual arguments**.
- To understand the **need of passing arguments** to function.

# Introduction to Functions

Functions are the building blocks of C and the place where all program activity occurs.



## **Benefits of Using Functions:**

- It provides modularity to the program.
- Easy code reusability ( Call the function by its name to use it )
- In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.

Credits: <http://www.studytonight.com/c/types-of-function-calls.php>



# Introduction to Functions

## A function is independent:

- It is “completely” **self-contained**
- It can be **called at any place** of your code and can be **ported** to another program
- ✓ **reusable** - Use existing functions as building blocks for new programs
- ✓ **Readable** - more meaningful
- ✓ **procedural abstraction** - hide internal details
- ✓ **factoring of code** - divide and conquer

# Introduction to Functions

A function:

receives zero or more parameters,  
performs a specific task, and  
returns zero or one value

A function is invoked / called by name and parameters

Communication between function and invoker code is  
through\_\_\_\_\_

➤ In C, whether two functions can have the same name?

# Types of C Functions

- Library function
- User defined function

## Library function

- Library functions are the in-built function in C programming system

For example:

- ❖ `main()` - - The execution of every C program
- ❖ `printf()` - `printf()` is used for displaying output in C.
- ❖ `scanf()` - `scanf()` is used for taking input in C.

## **Some of the math.h Library Functions**

- `sin()` → returns the sine of a radian angle.
- `cos()` → returns the cosine of an angle in radians.
- `tan()` → returns the tangent of a radian angle.
- `floor()` → returns the largest integral value less than or equal to x.
- `ceil()` → returns the smallest integer value greater than or equal to x.
- `pow()` → returns base raised to the power of exponent(xy).

## **Some of the conio.h Library Functions**

- `clrscr()` → used to clear the output screen.
- `getch()` → reads character from keyboard.

# User defined function

- Allows programmer to define their own function according to their requirement.

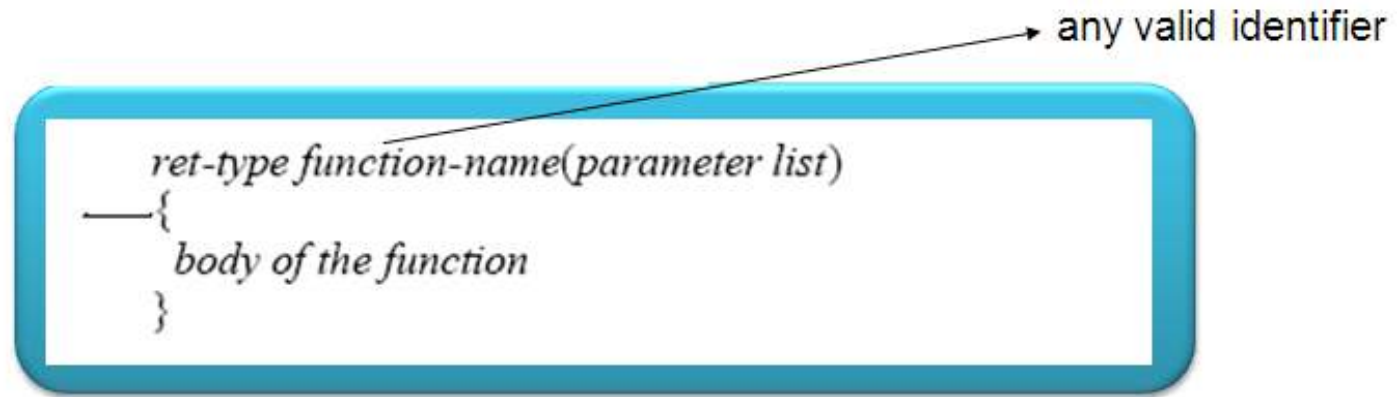
## Advantages of user defined functions

- It helps to **decompose the large program into small segments** which makes programmer easy to understand, maintain and debug.
- If **repeated code** occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmer working on large project can **divide the workload by making different functions.**

# Function naming rule in C

- Name of function includes **only alphabets, digit and underscore**.
- First character of name of any function must be an **alphabet or underscore**.
- Name of function **cannot be any keyword** of C program.
- Name of function **cannot be global identifier**.
- Name of function is **case sensitive**
- Name of function cannot be **register pseudo variables**

# The General Form a Function



The **ret-type** specifies the type of data that the function returns.

A function may **return any type** ( **default: int** ) of data **except an array**

The **parameter (formal arguments) list** is a **comma-separated list of variable names and their associated types**

**Q:** When the parameters **receive the values of the arguments** ?

A function can be without parameters:

**Q:** How do you specify an empty parameter list?

## More about formal argument/parameter list .....

(type varname1, type varname2, . . . , type  
varnameN)

You can declare **several variables to be of the same type**, using a comma separated list of variable names.

In contrast, all function parameters **must be declared individually**, each including both the type and name

- f(int i, int k, int j)
- f(int i, k, float j) Is it correct?



# Scope of a function

Each function is a **discrete block of code**. Thus, a function **defines a block scope**.

A function's code is **private** to that function and cannot be accessed by any statement in any other function except through a call to that function.

- **Variables that are defined within a function are local variables**
- A local variable **comes into existence when the function is entered and is destroyed upon exit**
- A local variable **cannot hold its value between function calls**

# Contd...

The **formal arguments / parameters** to a function also fall **within the function's scope**:

- known throughout the entire function
- comes into existence when the function is called and is destroyed when the function is exited.

Even though they perform the special task of receiving the value of the arguments passed to the function, they **behave like any other local variable**

# Returning value, control from a function

If nothing returned

- return;
- or, until reaches right curly brace

If something returned

- return expression;

**Only one value can be returned from a C function**

A function can **return only one value**, though it can return **one of several values** based on the evaluation of certain conditions.

**Multiple return statements** can be used within a single function (eg: inside an “if-then-else” statement...)

The return statement not only returns a value back to the calling function, it **also returns control back to the calling function**

# Three Main Parts of a Function

- Function Declaration (Function prototype)
- Function Definition
- Function Call

# Structure of a C program with a Function

**Function prototype** //giving the name, return type and the type of formal arguments

```
main()  
{  
.....
```

**Call to the function:**

Variable to hold the value returned by the function = Function name with actual arguments

```
.....  
}
```

**Function definition:**

**Header of function** with name, return type and the type of formal arguments as given in the prototype

**Function body within { }** with local variables declared , statements and return statement

- Functions should be **declared before they are used**
- Prototype only needed if function definition comes after use in program
- Function prototypes are always declared **at the beginning of the program** indicating :

name of the function, data type of its arguments &  
data type of the returned value

```
return_type  function_name ( type1  name1, type2  name2,  
                           ..., typeN  nameN );
```

# Function Definition

Function header

return\_type function\_name ( type1 name1, type2 name2,  
..., typen namen)

{  
    local variable declarations  
    .... otherstatements...  
    return statement  
}

Function Body

# Function call

A function is **called from the main()**

A function can in turn **call another function**

Function call statements **invokes the function** which means the **program control passes to that function**

Once the function completes its task, the **program control is passed back to the calling environment**



# Function call

Variable = function\_name ( actual argument list);

Or

Function\_name ( actual argument list);

Function **name, the type and number of arguments must match** with that of the function declaration stmt (function prototype) and the header of the function definition

Examples:

result = sum( 5, 8 );    display( );    calculate( s, r, t );

result = sum; (Wrong)

# Return statement

To return a value from a C function you must explicitly return it with a return statement

`return <expression>;`

The expression can be **any valid C expression** that resolves to the type defined in the function header

```
add( int a, int b)
```

```
{
```

```
    return (a + b);
```

```
}
```

```
add( int a, int b)
```

```
{
```

```
    int c = a+ b;;
```

```
    return ( c );
```

```
}
```

**Ex: Function call:**    `int value = add(5,8)`

Here, add( ) sends back the value of the expression (a + b) or value of c to main( )

# Examples

## Function Prototype Examples

```
double squared (double number);  
void print_report (int);  
int get _menu_choice (void);
```

## Function Definition Examples

```
double squared (double number)  
{  
    return (number * number);  
}  
  
void print_report (int report_number)  
{  
    if (report_number == 1)  
        printf("Printer Report 1");  
    else  
        printf("Not printing Report 1");  
}
```

# Example C program..

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

```
float average(float, float, float);
```

```
int main( )
```

```
{  
    float a, b, c;  
    printf("Enter three numbers please\n");  
    scanf("%f, %f, %f",&a, &b, &c);  
    printf("Avg of 3 numbers = %.3f\n",  
    return 0;  
}
```

*Function prototype*

**average(a, b, c) );**

*Function call*

The definition of function average:

```
float average(float x, float y, float z) //local variables x, y, z
```

```
{
```

```
    float r; // local variable
```

```
    r = (x+y+z)/3;
```

```
    return r;
```

```
}
```

Function header

Function Body

# Categorization based on arguments and return value

- Function with no arguments and no return value
- Function with no arguments and return value
- Function with arguments but no return value
- Function with arguments and return value.

**Credits :** <http://www.programiz.com/c-programming/types-user-defined-functions>

# Calling Functions – Two Methods

## Call by value

- **Copy of argument passed**
- Changes in function do not effect original
- Use when function does not need to modify argument
  - Avoids accidental changes

## Call by reference

- **Passes original argument**
- Changes in function effect original
- Only used with trusted functions

# Call by Value

When a function is called by an argument/parameter which **is not a pointer** the **copy of the argument is passed to the function.**

Therefore a **possible change on the copy does not change the original value of the argument.**

## Example:

Function call **func1 (a, b, c);**

Function header **int func1 (int x, int y, int z)**

Here, the parameters **x , y and z** are initialized by the values of **a, b and c**

**int x = a**

**int y = b**

**int z = c**



# Example C Program

```
void swap(int, int );
```

```
main()  
{  
    int a=10, b=20;  
    swap(a, b);  
    printf(“ %d %d \n”, a, b);  
}
```

```
void swap (int x, int y)  
{  
    int temp = x;  
    x= y;  
    y=temp;  
}
```

## Intricacies of the preceding example

- In the preceding example, the function `main()` declared and initialized two integers `a` and `b`, and then invoked the function `swap( )` by **passing `a` and `b` as arguments** to the **function `swap( )`**.
- The **function `swap( )` receives the arguments `a` and `b` into its parameters `x` and `y`**. In fact, the function `swap( )` receives a **copy of the values** of `a` and `b` into its parameters.
- The **parameters of a function are local to that function**, and hence, any **changes made by the called function to its parameters affect only the copy received by the called function**, and do not affect the value of the variables in the called function. This is the **call by value mechanism**.

When a function is called by an argument/parameter which **is a pointer** (address of the argument) the **copy of the address of the argument is passed to the function**

Therefore, a **possible change on the data at the referenced address changes the original value of the argument.**

# How to swap two numbers using Call by reference?

```
#include<stdio.h>
void swap(int *,int *);
int main()
{
    int a,b;
    printf("Enter first number : " );
    scanf("%d",&a);
    printf("Enter second number: ");
    scanf("%d",&b);
    printf("Numbers before function call:
%d\t%d\n",a,b);
    swap(&a,&b);
    printf("Numbers after function call :
%d\t%d\n",a,b);
    return 0;
}
```

## Output:-

```
Enter first number : 5
Enter second number: 10
Numbers before function call: 5 10
Numbers before swapping : 5    10
Numbers after swapping : 10    5
Numbers after function call : 10  5
```

```
void swap(int *a, int *b)
{
    int t;
    printf("Numbers before swapping :
%d\t%d\n",*a,*b);
    t = *a;
    *a = *b;
    *b = t;
    printf("Numbers after swapping :
%d\t%d\n",*a,*b);
}
```

# Points to be noted while using Call-by-Reference

	Call-by-Value	Call-by-Reference
Function Declaration	<code>void swap(int ,int );</code>	<code>void swap(int *,int *);</code>
Function Header	<code>void swap(int a, int b)</code>	<code>void swap(int *a, int *b)</code>
Function Call	<code>swap(a,b);</code>	<code>swap(&amp;a,&amp;b);</code>

- Requires '\*' operator along with data type of arguments – in declaration as well as Function header.
- Requires '&' along with actual arguments in Function call.
- Requires '\*' operator inside function body.

# When do you need pointers in functions?

- First scenario: In Call-by-Reference.
  - There is a requirement to modify the values of actual arguments.
- Second scenario: While passing array as an argument to a function.
- Third Scenario: If you need to return multiple values from a function.

# Make Your Own Header File ?

## Step1 : Type this Code

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

- In this Code write only function definition as you write in General C Program

## Step 2 : Save Code

- Save Above Code with [.h ] Extension .
- Let name of our header file be myhead [ myhead.h ]
- Compile Code if required.

### Step 3 : Write Main Program

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

```
#include"myhead.h"
```

```
main() {
```

```
int num1 = 10, num2 = 10, num3;
```

```
num3 = add(num1, num2);
```

```
printf("Addition of Two numbers : %d", num3);
```

```
}
```

Here,

- Instead of writing < myhead.h > use this terminology “myhead.h”
- All the Functions defined in the myhead.h header file are now ready for use .
- Directly call function add(); [ Provide proper parameter and take care of return type ]

**Note :** While running your program precaution to be taken :

Both files [ myhead.h and sample.c ] should be in same folder.



# Storage Classes

- A **storage class** defines the scope (visibility) and life time of variables and/or functions within a C Program.
- Automatic variables → `auto`
- External variables → `extern`
- Static variables → `static`
- Register variables → `register`

## **auto - Storage Class**

**auto is the default storage class** for all local variables.

```
{  
    int Count;  
    auto int Month;  
}
```

The example above defines two variables with the same storage class.

**auto can only be used within functions, i.e. local variables.**

## extern - Storage Class

- These variables are declared **outside any function**.
- These variables are **active and alive throughout the entire program**.
- Also known as **global variables** and **default value is zero**.

## static - Storage Class

- The value of static variables **persists until the end of the program**.
- It is declared using the keyword static like  
static int x;  
static float y;
- It may be of **external or internal type** depending on the place of their declaration.
- Static variables are **initialized only once**, when the program is compiled.

## register - Storage Class

- These variables are stored in one of the **machine's register** and are declared using register keyword.  
eg. `register int count;`
- Since **register access are much faster than a memory access** keeping frequently accessed variables in the register lead to faster execution of program.
- **Don't try to declare a global variable as register.** Because the register will be occupied during the lifetime of the program.

➤ Communication between the function and invoker code is through **the parameters and return value**

➤ Name of function cannot be **register pseudo variable**

**Register pseudo variables** are reserved word of C language.

We Cannot use these words as a name of function, otherwise it will cause compilation error.

Here is an example:

```
#include<stdio.h>
int main() {
    int c;
    c=_AL();
    printf("%d",c);
    return 0;
}
int _AL() {
    int i=5,j=5;
    int k=++j + ++j+ ++j;
    i=++i + ++i+ ++i;
    return k+i;;
}
```

Output: Compilation error

Explanation: `_AL` is register Pseudo variables in c

Ref: <https://www.cquestions.com/2009/05/name-of-function-cannot-be-register.html>

# Summary

- Discussed the modularization techniques in C.
- Illustration of functions with different parts – Prototype, Call and Definition.
- Discussed formal and actual parameters and passing mechanism.

# Introduction to Algorithms

# Agenda

- Computational Problem
- Formal Specification of a problem
  - Examples, Types
- Instance of a problem
- Introduction to Algorithms
  - Definition, Correct Algorithm, Incorrect Algorithm
- Algorithm Representation
- Pseudocode conventions
- Linear Search Algorithm
- Correctness of Linear Search

# Computational Problem

- **Statement of the problem** specifies the desired input/output relationship

Eg: Find the factorial of a given number.

- **Algorithm** describes a specific computational procedure to achieve the input/output relationship

Eg: Steps to find the factorial of the input number



## **Examples of Problems solved by Algorithms:**

- Human Genome Project
- Internet - Search Engine
- Electronic commerce
- Resource allocation
- Shortest Path problem
- Longest common subsequence

Ref: CLRS book (Chapter 1)

## Types of Computational Problems:

**Decision Problems:** Answer for every instance is either YES or NO

**Search Problems:** Searching for a given value in the list of values

**Optimization Problems:** Find a "best possible" solution among the set of all possible solutions

**Counting Problems:** Number of solutions to a search problem

## **Classify the following problems:**

- 1) Find a path between two nodes in a graph
- 2) Find the maximum value in the list of values
- 3) Checking whether the given number is in the list or not
- 4) Find the shortest path between two nodes in the given graph
- 5) Find the number of non-trivial prime factors of the number  $n$
- 6) Check whether the number  $X$  is an Armstrong number or not

# Computational Problem

- Problem specifies the desired input / output relationship

Eg: Find the Prime factors of a given number.

## Formal definition of a **Searching Problem**

**Input:** A sequence of  $n$  numbers  $A = \langle a_1, a_2, a_3, a_4, \dots, a_n \rangle$

and a value  $v$

**Output:** An index  $i$  such that  $v = A[i]$  or the special value NIL if  $v$  does not appear in  $A$ .

Eg:

What is the Input to Searching Problem?

$\langle 10, 29, 65, 23, 12, 15, 78 \rangle, 23$

Output ?

**Input** is referred as **instance of the Searching Problem**

**Instance** consists of the **input** needed to compute the solution to the problem.

**Eg: Find the factorial of a number.**

**Sort the given input.**

What are the reasonable constraints that can be considered?

# Formal definition of a Sorting Problem

**Sorting** - Fundamental Operation in Computer Science

**Input** : A sequence of  $n$  numbers  $\langle a_1, a_2, a_3, a_4, \dots, a_n \rangle$

**Output**: A permutation (reordering)  $\langle a_1', a_2', a_3', a_4', \dots, a_n' \rangle$  of the input sequence such that  $a_1' \leq a_2' \leq a_3' \leq a_4' \leq \dots \leq a_n'$

Eg:

(Input Sequence) **Instance**:  $\langle 31, 41, 59, 26, 41, 58 \rangle$

**Output** Sequence:  $\langle 26, 31, 41, 41, 58, 59 \rangle$

# Algorithms

- **Well defined computational procedure** that **takes input** (some value or set of values) and **produces output** (some value or set of values)
- Sequence of well-defined computational steps that **transform the input to the output**
- **Tool** for solving a well-specified computational problem



# Correct & Incorrect Algorithms

- **Correct** - for every input instance, it halts with the correct output
- Correct algorithm solves the given computational problem
- Incorrect algorithm - might not halt at all on some input instances or it might halt with an incorrect answer
- **Our focus is on correct algorithms**

## Exercises:

**Give Examples for the following:**

Correct algorithm

An algorithm that does not halt on some input instances

An algorithm that gives an incorrect answer

An **algorithm** can be specified,

- English
- Computer program
- Hardware design

**ONLY requirement:**

Precise description of the computational procedure

# Representation - Pseudocode

- **Pseudocode:** An expressive way of making things clear using English sentences
- **Pseudocode** - Not concerned with issues of software engineering such as modularity, error handling etc.

# Pseudocode conventions

- **Indentation** indicates block structure –

Eg: loops and conditional constructs

Eg1:

1. if  $A[i] = \text{key}$
2.            $\text{found} = 1$
3.           return  $i$

Eg2:

1. while  $i > 0$  and  $A[i] > \text{key}$
2.        $A[i+1] = A[i]$
3.        $i = i - 1$

// This symbol indicates that the remainder of the  
line is a comment

2. **for** i = 1 **to** A.length

3.       if A[ i ] = key

4.       found = 1

5.       return i

- Keyword **to** is used when a **for** loop increments its value by 1 in each iteration
- Keyword **downto** is used when a **for** loop decrements its value of loop counter by 1
- If the loop counter changes by an amount **greater than 1**, the amount of change follows the optional keyword **by**

- $i=j=e$  is equivalent to  $j=e$  followed by  $i=j$
- Variables are local to a given procedure
- We shall not use global variables without explicit indication
- $A[i]$  indicates the  $i^{\text{th}}$  element of  $A$
- $A[i..j]$  indicates the element of subarray of  $A$  with elements  $A[i]$ ,  $A[i+1]$ , ...,  $A[j]$
- We access a particular attribute of an object by an object name followed by a dot followed by the attribute name

Eg:  $A.length$

- Parameters are passed to a procedure by **value**
- **called procedure** receives its own copy of the parameters and, if it assigns a value to a parameter, the change is not seen by the calling procedure
- A **return** statement immediately transfers control back to the point of call in the calling procedure.
- Most **return** statements also take a value to pass back to the caller.
- Pseudocode - allows multiple values to be returned in a single **return** statement.



- The boolean operators “and” and “or” are **short circuiting**.
- When we evaluate the expression “x and y”,
  - Possible cases
- While evaluating the expression “x or y”
  - Possible cases

# **First Example - Pseudocode for Linear Search Algorithm**

# Pseudocode of Linear Search

**LINEAR SEARCH(A, key)** // Pseudocode of Linear Search

1. found = 0
2. for i = 1 to A.length
3.     if A[ i ] = key
4.         found = 1
5.     return i
6. if found = 0
7.     return 0

Exercise: Trace the execution of the Linear Search Algorithm for the given list of values and the key=33

Linear Search



Courtesy:

[https://www.tutorialspoint.com/data\\_structures\\_algorithms/linear\\_search\\_algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms/linear_search_algorithm.htm)

# Correctness Proofs

- How do you prove an algorithm is correct ?

# Correctness Proofs

- How do you prove an algorithm is correct ?

Proof by:

- Counter example (indirect proof )
- Induction (direct proof )
- Loop Invariant

Other approaches:

- proof by cases/enumeration
- proof by chain of iffs
- proof by contradiction
- proof by contrapositive

# Counter example (indirect proof )

- Searching for counterexamples is the best way to disprove the correctness of some things.
- Identify a case for which something is NOT true
- If the proof seems hard or tricky, sometimes a counterexample works
- Sometimes a counterexample is just easy to see, and can shortcut a proof
- If a counterexample is hard to find, a proof might be easier

# Counter example (indirect proof )

## Examples

1. Prove or disprove:

“Every positive integer is the sum of two squares of integers”

Proof by counterexample: 3

2. Prove or disprove:

"All shapes that are rectangles are squares."

"All shapes that have four sides of equal length are squares".

Proof by counterexample: Exercise



# Correctness Proof using Loop Invariant:

Useful for algorithms that loop.

Formally: Find a loop invariant, then prove:

1. Initialization phase
2. Maintenance phase
3. Termination phase

Similar to “Proof using Induction”.

Informally:

1. Find  $p$ , a loop invariant
2. Show the base case for  $p$
3. Use induction to show the rest.

# Proof by Loop Invariant Examples

**Invariant:** something that is always true

After finding a candidate loop invariant, we prove:

1. **Initialization:** How does the invariant get initialized?
2. **Loop Maintenance:** How does the invariant change at each pass through the loop?
3. **Termination:** Does the loop stop? When?

## Loop Invariant Proof Examples

- Linear Search
- Insertion Sort
- Bubble Sort

# Pseudocode of Linear Search

## **LINEAR SEARCH(A, v)**

1. found = 0
2. for i = 1 to A.length
3.       if A[ i ] = v
4.       found = 1
5.       return i
6. if found = 0
7.   return 0

Exercise: Trace the execution of the Linear Search Algorithm for the given list of values and the key=33

Linear Search



Courtesy:

[https://www.tutorialspoint.com/data\\_structures\\_algorithms/linear\\_search\\_algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms/linear_search_algorithm.htm)

# Linear Search

**Loop Invariant:** At the start of each iteration of the for loop on line 2, the subarray  $A[1 \dots i-1]$  does not contain the value  $v$

## **LINEAR SEARCH(A, v)**

1. found = 0
2. for i = 1 to A.length
3.       if  $A[i] = v$
4.       found = 1
5.       return i
6. if found = 0
7.   return 0

**Loop Invariant:** At the start of each iteration of the for loop on line 2, the subarray  $A[1 \dots i-1]$  does not contain the value  $v$

**First phase of the proof:**

**Initialization** Prior to the first iteration, the array  $A[1 \dots i-1]$  is empty ( $i == 1$ ).

That (empty) subarray does not contain the value  $v$ .

**Loop Invariant:** At the start of each iteration of the for loop on line 2, the subarray  $A[1 \dots i-1]$  does not contain the value  $v$

**Maintenance:** Line 3 checks whether  $A[i]$  is the desired value ( $v$ ).

If Yes, the algorithm will return  $i$ , thereby terminating and producing the correct behavior - the index of value  $v$  is returned, if  $v$  is present.

If No ( $A[i]$  not equal to  $v$ ), then the loop invariant holds at the end of the loop (the subarray  $A[1 \dots i]$  does not contain the value  $v$ ).

**Loop Invariant:** At the start of each iteration of the for loop on line 2, the subarray  $A[1 \dots i-1]$  does not contain the value  $v$

## **Termination Phase:**

The for loop on line 2 terminates when  $i > A.length$  (that is,  $n$ ). Because each iteration of a for loop increments  $i$  by 1, then  $i = n+1$ .

The loop invariant states that the value is not present in the subarray of  $A[1 \dots i - 1]$ . Substituting  $n+1$  for  $i$ , we have  $A[1 \dots n]$ .

Therefore, the value is not present in the original array  $A$  and the algorithm returns 0.



# References:

1. CLRS Book
2. [https://course.ccs.neu.edu/cs5002f18-seattle/lects/cs5002\\_lect11\\_fall18\\_notes.pdf](https://course.ccs.neu.edu/cs5002f18-seattle/lects/cs5002_lect11_fall18_notes.pdf)
3. [wikipedia.org](https://www.wikipedia.org)

# Insertion Sort Algorithm and its correctness

# Sorting Problem

**Input :** A sequence of  $n$  numbers  $\langle a_1, a_2, a_3, a_4, \dots, a_n \rangle$

**Output:** A permutation (reordering)  $\langle a_1', a_2', a_3', a_4', \dots, a_n' \rangle$

of the input sequence such that  $a_1' \leq a_2' \leq a_3' \leq a_4' \leq \dots \leq a_n'$

- An algorithm for sorting : Insertion sort
- Works in the same way as many people sort in hand while playing cards
  - Start with an empty left hand and all cards face down on the table
  - Remove one card at a time from the table and insert it to the correct position in the left hand

# Observations

- When we take any card (**say *key card***), we **start comparing it with the last card** in our left hand.
- If the value of the ***key card*** is less than the **last card** in our left hand, then only **we proceed with further comparisons**
- If the value of the ***key card*** is equal to or **greater than the last card** in our left hand, then we place the key card as the **last card**.
- This action is performed based on an observation that **cards in the left hand is already sorted**
- At any time, we are **inserting to a sorted list of elements**.

# Working of Insertion Sort algorithm

- What is the input of our algorithm?
- **Input:** array of elements 5, 7, 2, 3, 7, 10 .
- Pass the input array to the function insertion sort

5	7	2	3	7	10
---	---	---	---	---	----

- Keep 5 as such in the first position of the array, assuming that it is sorted by itself

5	7	2	3	7	10
---	---	---	---	---	----

- Take 7, **key=7**, compare it with 5, place it there itself

5	7	2	3	7	10
---	---	---	---	---	----

5	7	2	3	7	10
---	---	---	---	---	----

- Take 2, we put in a temporary variable (say *key*), we shift 7 to 2's position in the array,  
*key=2*

- Now the array looks like,

5	7	7	3	7	10
---	---	---	---	---	----

- Now, compare *key*(2) with 5, we shift 5 to next position in the array
- Now the array looks like,

5	5	7	3	7	10
---	---	---	---	---	----

- Then, place the key value in the first position

2	5	7	3	7	10
---	---	---	---	---	----

2	5	7	3	7	10
---	---	---	---	---	----

- Now **key=3**

2	5	7	7	7	10
---	---	---	---	---	----

2	5	5	7	7	10
---	---	---	---	---	----

2	3	5	7	7	10
---	---	---	---	---	----

2	3	5	7	7	10
---	---	---	---	---	----

- Now **key=7**

2	3	5	7	7	10
---	---	---	---	---	----

- Now **key=10**

2	3	5	7	7	10
---	---	---	---	---	----



## Few tips on the Design of algorithm:

- How many loops we should have?
- One loop for sure which goes from 1 to  $n$
- Another loop which goes from position of *key* element to 1 ?

# Insertion Sort Algorithm:

INSERTION SORT(A)

1. for  $j=2$  to  $A.length$
2.      $key = A[j]$ ;
3.     //Insert  $A[j]$  into the sorted sequence  $A[1...j-1]$
4.      $i = j-1$
5.     while  $i > 0$  and  $A[i] > key$
6.          $A[i+1] = A[i]$
7.          $i = i-1$
8.      $A[i+1] = key$

# Trace INSERTION SORT(A)

## INSERTION SORT(A)

1. for  $j=2$  to  $A.length$
2.      $key = A[j]$ ;
3.     //Insert  $A[j]$  into the sorted  
          sequence  $A[1..j-1]$
4.      $i = j-1$
5.     while  $i > 0$  and  $A[i] > key$
6.          $A[i+1] = A[i]$
7.          $i = i-1$
8.      $A[i+1] = key$

- Input : 5, 2, 4, 6, 1, 3
- First iteration of *for* loop , with  $j=2$
- Key = 2,  $i=1$
- *while* loop is executed as  $i>0$  and  $A[i]>key$
- $A[2] = 5$ ,  $i=0$
- *while* loop fails to execute as  $i=0$
- $A[0+1] = 2$
- Intermediate output after the 1<sup>st</sup> iteration of *for* loop : 2, 5, 4, 6, 1, 3

# Operations of insertion sort on an array

**A = 1, 4, -2, -3**

- After 1<sup>st</sup> iteration of while loop, with  $j = 2$ ,  $i = 1$ : A = 1, 4, -2, -3
- After 1<sup>st</sup> iteration of for loop, with  $j = 2$  : A = 1, 4, -2, -3
- After 1<sup>st</sup> iteration of *while* loop, with  $j = 3$ ,  $i = 2$ : A = 1, 4, 4, -3
- After 2<sup>nd</sup> iteration of *while* loop, with  $j = 3$ ,  $i = 1$ : A = 1, 1, 4, -3
- After 2<sup>nd</sup> iteration of *for* loop, with  $j = 3$  : A = -2, 1, 4, -3
- After 1<sup>st</sup> iteration of *while* loop, with  $j = 4$ ,  $i = 3$ : A = -2, 1, 4, 4
- After 2<sup>nd</sup> iteration of *while* loop, with  $j = 4$ ,  $i = 2$ : A = -2, 1, 1, 4
- After 3<sup>rd</sup> iteration of *while* loop, with  $j = 4$ ,  $i = 1$ : A = -2, -2, 1, 4
- After 3<sup>rd</sup> iteration of *for* loop, with  $j = 4$  : A = -3, -2, 1, 4

## Design Paradigm: Incremental Approach

An algorithm consider the elements **one at a time**, inserting each in its suitable place among those already considered (keeping them sorted).

Insertion sort is an **example of an incremental algorithm**; it builds the sorted sequence one number at a time.

### **Simplest example of the incremental insertion technique**

build up a complicated structure on  $n$  items by first building it on  $n - 1$  items and then making the necessary changes to fix things in adding the last item.

## In place sorting Algorithm

- Insertion sort is an **in place** sorting method i.e it rearranges the numbers within the input array, with at most a **constant number** of them stored outside the array at any time.
- The **input array A contains the sorted** sequence after the procedure is finished

# How do we prove that INSERTION SORT(A) is correct?

- We observe the algorithm critically and try to understand
- We observe that :
  - The index  $j$  indicates:
    - The current number/card being inserted
  - At the beginning of each iteration of *for* loop indexed by  $j$ :  
 $A[1 .. j-1]$  is sorted and  $A[j+1..n]$  is remaining
- **Elements  $A[1..j-1]$  are the elements originally in positions 1 through  $j-1$ , but now in sorted order**
- We state these properties of  $A[1 .. j-1]$  formally as a **loop invariant**

# Loop Invariant of INSERTION SORT(A)

- **At the start of each iteration of the for loop of lines 1-8, the subarray  $A[1 \dots j-1]$  consists of the elements originally in  $A[1 \dots j-1]$ , but in sorted order.**
- We use loop invariant (a property of the algorithm) to prove that the algorithm is correct
- We must show three things about a loop invariant:
  - Initialization
  - Maintenance
  - Termination



# Correctness of Insertion Sort

- We must show three things about a loop invariant :
  - **Initialization:** The loop invariant is true prior to the first iteration of the loop
  - **Maintenance:** If the loop invariant is true before an iteration of the loop, it remains true before the next iteration
  - **Termination:** When the loop terminates, the loop invariant gives us a useful property that helps us to show that the algorithm is correct
- When the first two properties hold, the loop invariant is true prior to every iteration of the loop.

# Loop Invariant vs Mathematical Induction

- Invariant holds before the first iteration ~ base case of mathematical induction
- Invariant holds from iteration to iteration ~ inductive step
- Third property ie. The termination property differs from Mathematical induction
  - In Mathematical induction, we use the inductive step infinitely, whereas here we stop the induction when the loop terminates

# Proof – Correctness of Insertion sort

- **Initialization:** Loop invariant trivially holds before the first loop iteration.  $A[1]$  is sorted by itself.
- **Maintenance:**
  - *for* loop works by moving  $A[j-1]$ ,  $A[j-2]$ ,  $A[j-3]$  and so on by one position to the right and finds the proper position for  $A[j]$  and inserts the value of  $A[j]$ .
  - Hence,  $A[1..j-1]$  consists of the elements originally in  $A[1..j-1]$ , but in sorted order.
  - Incrementing  $j$  for the next iteration preserves the loop invariant

# Proof: Correctness of Insertion sort -contd.

- **Termination:**
  - *for* loop terminates when  $j > A.length = n$  ie.  
 $j = n+1$
  - Substituting  $n+1$  in the wording of loop invariant, the subarray  $A[1..n]$  consists of originally in  $A[1..n]$ , but in sorted order
- Observe that  $A[1..n]$  is the entire array and since the entire array is sorted, the algorithm is correct.

# Binary Search

# Overview

- **Recursive solutions**
- **Implementation**

# Problem Solving

- **Problem: Compute the factorial of a given integer  $n \geq 0$** 
  - If  $n=0$ , factorial =1
  - If  $n > 0$  .....?

# Problem Solving

□ **Problem: Compute the factorial of a given integer  $n \geq 0$**

□ If  $n=0$ , `factorial = 1`

□ If  $n > 0$

1. `factorial = n * (n-1) * (n-2) * ... * 1`

2. `factorial = n * factorial (n-1)`



# Problem Solving – Different Approaches

- **Iterative Solution**

- `factorial = n * (n-1) * (n-2) * ... * 1`

- Use a looping construct

- **Recursive Solution – direct, based on the mathematical formula**

- `factorial = n * factorial (n-1)`

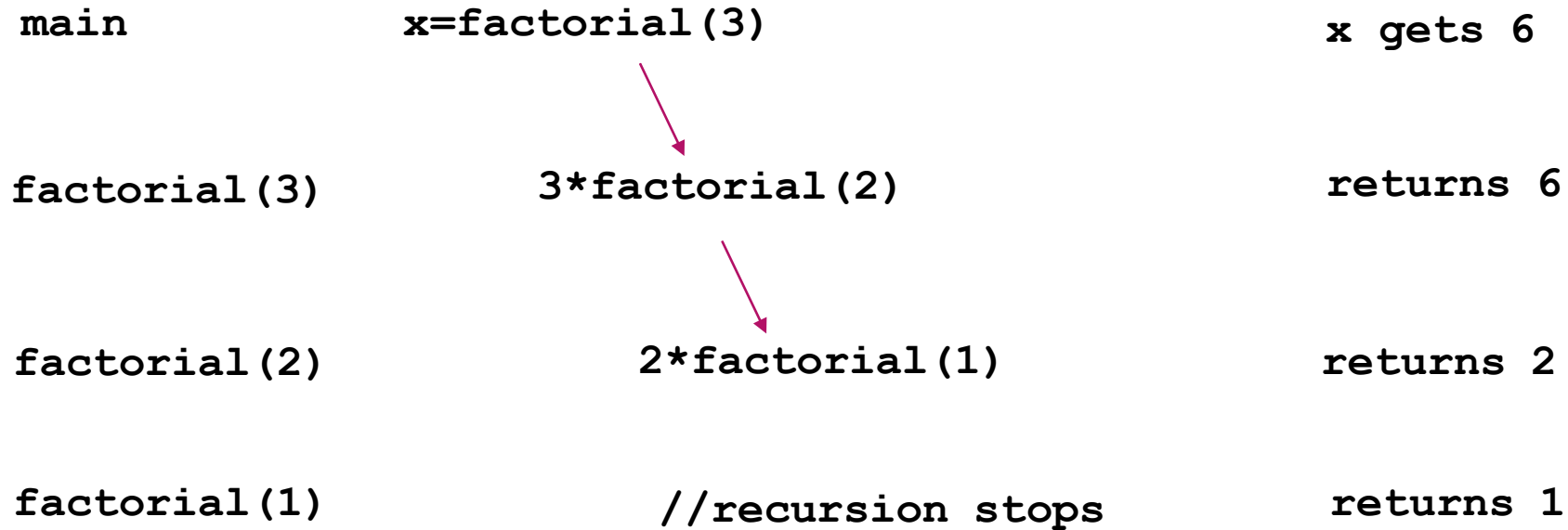
- Recursive function call

# Factorial - Recursive function

```
int factorial (int n){  
    // returns the factorial of n, given n>=0  
    if (n<2)  
        return 1;  
    else  
        return n * factorial (n-1);  
}
```

factorial(3)    3\*factorial(2)    3\*2\*factorial(1)    3\*2\*1

# Factorial - Recursive calls and returns



# Recursive Solution

- Divide into smaller problems
- Solve the sub problems recursively ( direct solution when the sub problem size is small enough)
- Obtain the solution to the original problem from the solutions to the smaller sub problems

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

- Problems with natural recursive solution
- Simple, elegant and concise code

# Recursion – Base Case, Recursive case

- **Recursive Case**

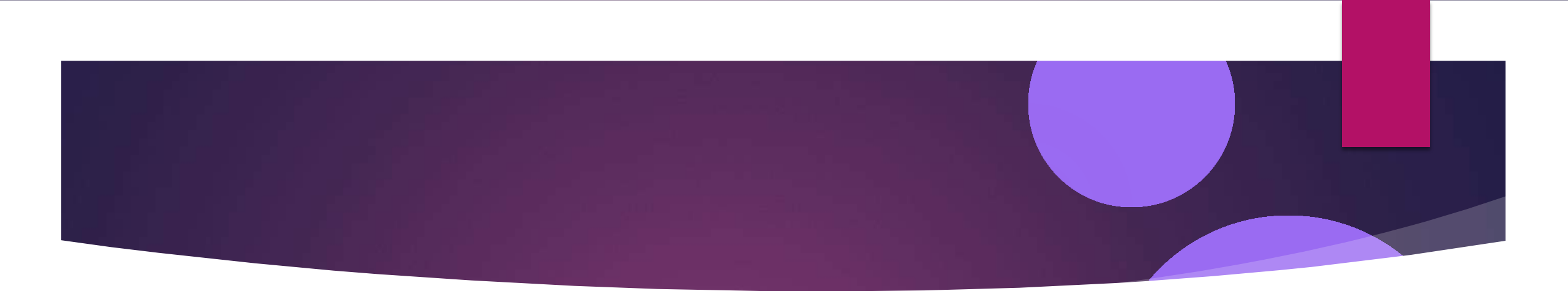
- Subproblems are large enough to solve recursively

- **Base Case**

- Direct solution (without recursion) when the subproblem is small enough

- **More than one base case / recursive case possible as in `fibonacci()`**

- `fib(n) = fib(n-1) + fib(n-2)`



```
int fibonacci(int n)
{
    if(n == 0){
        return 0;
    } else if(n == 1) {
        return 1;
    } else {
        return (fibonacci(n-1) + fibonacci(n-2));
    }
}
```

# Searching Problem : Formal Specification

- Input : A sequence of  $n$  numbers  
 $A = \langle a_1, a_2, \dots, a_n \rangle$  and a key  $k$
- Output : An index  $i$  such that  $k=A[i]$   
or the special value  $-1$  if  $k$  does not  
appear in  $A$

# Searching Problem : Sorted sequence

- Input : A sequence of  $n$  numbers  $A = \langle a_1, a_2, \dots, a_n \rangle$  such that  $a_1 \leq a_2 \leq \dots \leq a_n$  and a key  $k$
- Linear Search ?
  - Number of comparisons?
  - Can we reduce the number of comparisons, given the **input is sorted**?



# Example

A: 3 5 7 9 11 12 35 40 48 52 65

k : 48

- You know how to reduce the number of comparisons, right ?

# Binary Search

- Input : A sequence of  $n$  numbers  $A = \langle a_1, a_2, \dots, a_n \rangle$  such that  $a_1 \leq a_2 \leq \dots \leq a_n$  and a key  $k$
- Binary Search
  - Compare  $k$  with the middle element of the sequence, say  $A[\text{mid}]$
  - If  $k = A[\text{mid}]$  return  $\text{mid}$
  - If  $k < A[\text{mid}]$  **search** in the first half,  $(A[1..\text{mid}-1])$
  - If  $k > A[\text{mid}]$  **search** in the second half,  $(A[\text{mid}+1..n])$
- Number of comparisons?

# Example

A: 3 5 7 9 11 12 35 40 48 52 65      k : 48

A[mid ] is 12 ,    k > 12      search A [7 .. 11]

A: 3 5 7 9 11 12 35 40 48 52 65

A[mid]=48 matches k

Search finished with just 2 comparisons

k : 65 ?      k:3 ?      k:6?

# Binary Search

## □ Binary Search

- If  $k < A[\text{mid}]$  `search` in the first half ( $A[1..\text{mid}-1]$ )
- If  $k > A[\text{mid}]$  `search` in the second half ( $A[\text{mid}+1..n]$ )

## □ Size of the sequence to be searched is reduced to half

□  $n \rightarrow n/2 \rightarrow n/4 \rightarrow \dots \rightarrow 1$

# Binary Search – Algorithm

```
BinarySearch (A, m, n, k)
    if (m>n) .....???? //Base Case
    mid=(m+n)/2
    if A[mid]=k return mid;      //Base Case
    else if k < A[mid]
        BinarySearch(A, m, mid-1, k)
    else if k > A[mid]
        BinarySearch(A, mid+1, n, k)
```

# Binary Search – Algorithm

```
BinarySearch (A, m, n, k)
    if (m>n) return -1;    //Base Case
    mid=(m+n)/2
    if A[mid]=k return mid;    //Base Case
    else if k < A[mid]
        BinarySearch(A, m, mid-1, k)
    else if k > A[mid]
        BinarySearch(A, mid+1, n, k)
```

# Calls / Returns

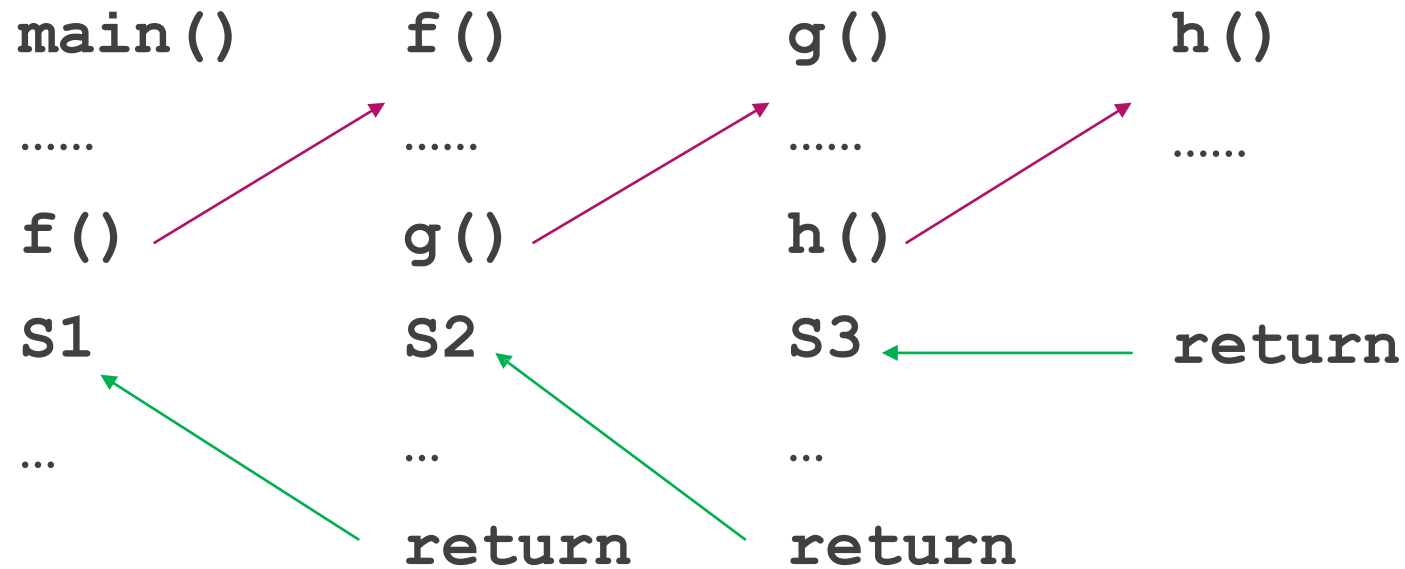
Call Sequence :

`main() → f() → g() → h()`

Return Sequence:

`main() ← f() ← g() ← h()`

# Return Address



**Return Address** : Address of the instruction in the caller function to which control should return



# Passing parameters and return values

```
main()
```

```
x = f(y, z)
```

```
.....
```

```
f(a, b)
```

```
.....
```

```
return(c)
```

- For each active function store return address, parameters, local variable etc.
- Each activation require separate storage area

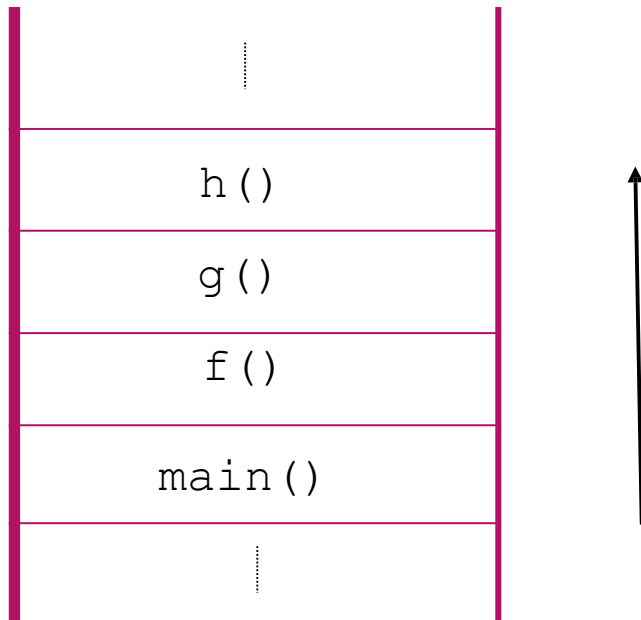
# Activation Records

- Activation Record / Stack Frame
- Data Area in memory for storing the data relevant to one activation of the function (parameters, locals, return address...)
  - Set up on entry to the function
  - Area can be reallocated after return
- Stacked one on top of the other (call stack/control stack)

`main()` → `f()` → `g()` → `h()`

How many activation records when `h()` is active?

# Stack of Activation records



# Activation Records – Recursive Functions

- **Multiple activations of the same function**
  - **Multiple activation records**
- **Time for setting up activation records**
- **Space in stack**

# Conclusion

- ❑ **Recursive solution- simple, elegant and concise code**
- ❑ **Functional Programming Languages – Scheme, ML**
- ❑ **More space requirement – keep track of status**
- ❑ **Common mistake – Missing base case – never stops**

# Reference

- **Text Books and other materials on Programming Language Theory/Compilers/Algorithms**

# Reference

- **Text Books and other materials on Programming Language Theory/Compilers/Algorithms**

# Merge Sort

## Algorithm & Correctness



# Design paradigms

- **Paradigm**

- “In science and philosophy, a paradigm is a **distinct set of concepts or thought patterns**, including theories, research methods, postulates, and standards for what constitutes legitimate contributions to a field”- Wikipedia

# Types of Design Paradigms

- Incremental Approach
- Divide and Conquer
- Greedy approach
- Dynamic Programming

# Incremental approach

- **Example: Insertion sort**

- In the **so far sorted subarray**, insert a new single element into its proper place, resulting in the new sorted subarray

- Example:

[	...	]	[	.....	]
$A[1 .. j-1]$				$A[j .. n]$	
			↑		

Key =  $A[j]$

# Divide and Conquer

*Our life is frittered away by detail. Simplify, simplify.*

— Henry David Thoreau

*The control of a large force is the same principle as the control of a few men:  
it is merely a question of dividing up their numbers.*

— Sun Zi, *The Art of War* (c. 400 C.E.), translated by Lionel Giles (1910)

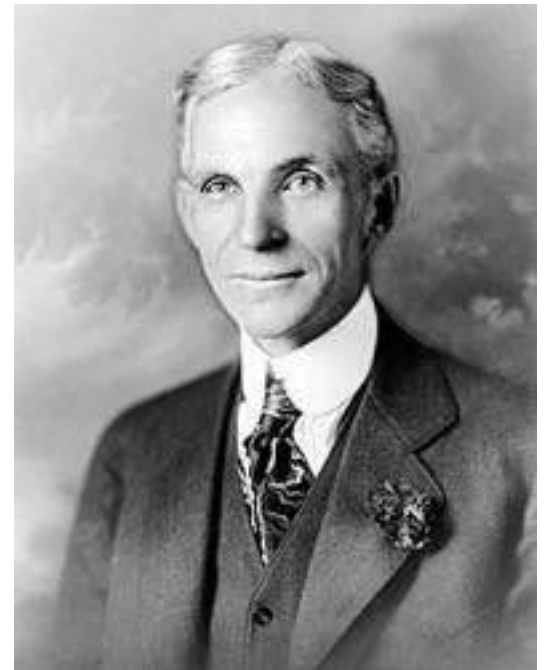
*Nothing is particularly hard if you divide it into small jobs.*

— Henry Ford

Henry Ford (July 30, 1863 - April 7, 1947) was an American captain of industry and a business magnate.

Founder of the Ford Motor company

Sponsor of the development of the assembly line technique of mass production.



# Henry Ford's Assembly line



# Divide and Conquer

- Three crucial steps
  - **Divide** the problem into smaller sub problems
  - **Conquer** the smaller subproblems recursively.
  - **Combine** solutions of the subproblems to get the solution of the original problem

# Divide and conquer - First step

- Divide/Break the problem into smaller sub problems
  - For example, Problem P is divided into subproblems P1 and P2.
  - Also, P1 and P2 resemble the original problem and their size is small



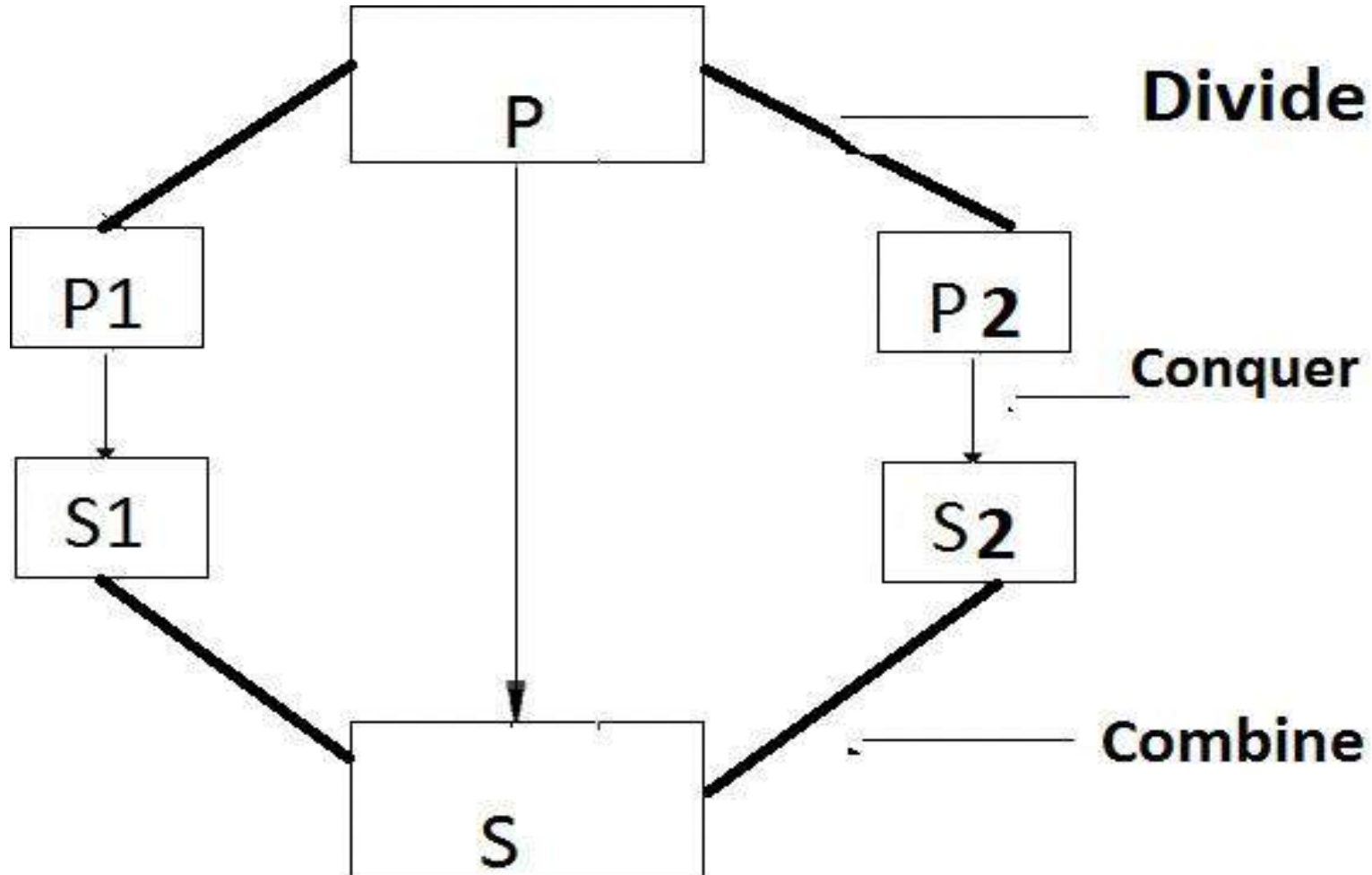
# Divide and conquer - Second step

- Conquer/Solve the smaller subproblems recursively. If the subproblem size is small, solve them directly
  - P1 is solved to give S1, P2 is solved to give S2

# Divide and conquer - Third step

- Merge/Combine these solutions to create a solution to the original problem
  - $S_1$  and  $S_2$  are combined to give the solution  $S$  for the original problem  $P$

# Pictorial Representation: Divide and Conquer



# Divide and Conquer (D & C)

- Most of the algorithms designed using D & C are **recursive** in nature
- **Recursive algorithms:** Call themselves recursively to solve the closely related subproblems
- **Examples**
  - Towers of Hanoi
  - Binary search
  - Merge Sort
  - Quick sort

# Merge Sort

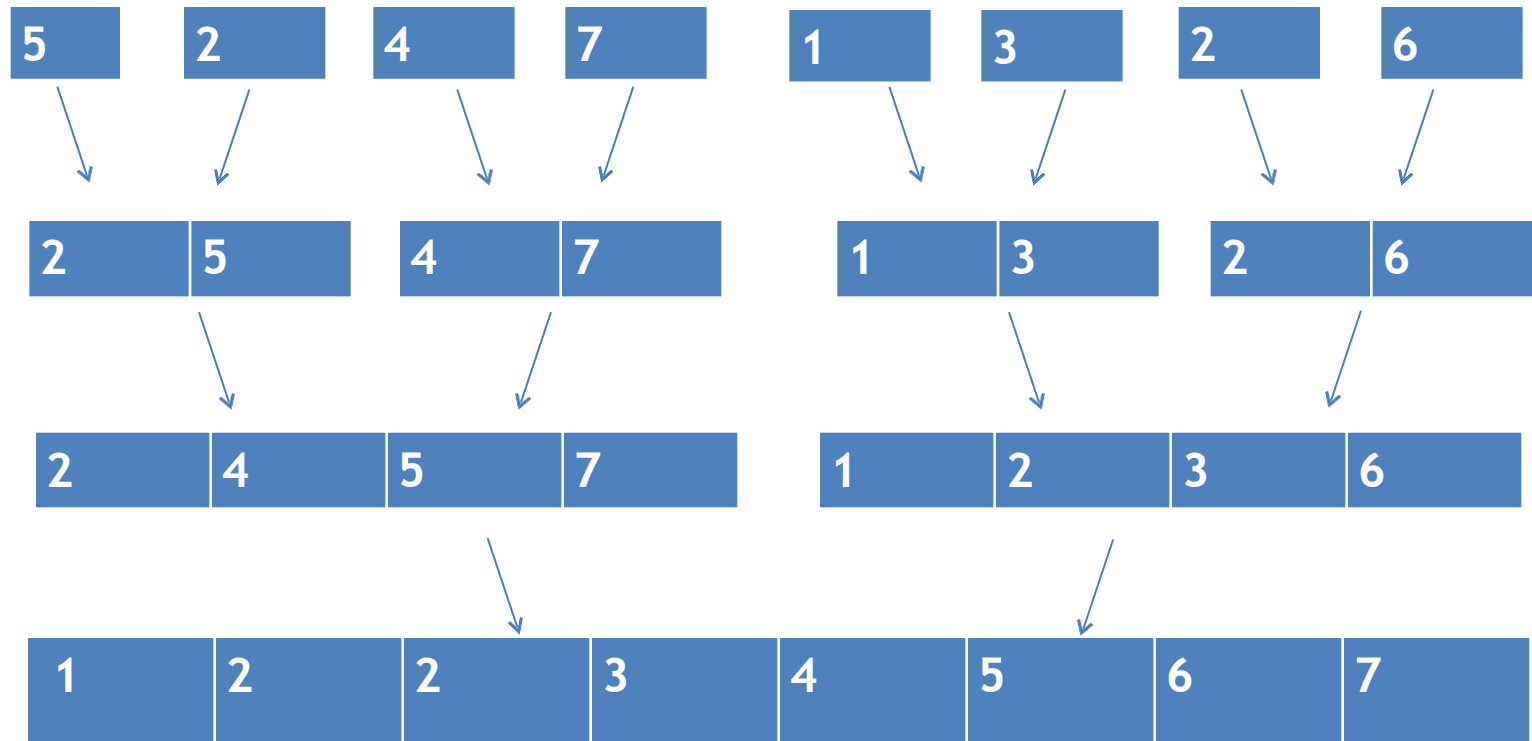
- Follows D & C paradigm
- Divide: Divide the  $n$ -element array into two subarrays of size  $n/2$
- Conquer: Sort the two subarrays recursively
- Combine: Merge the two sorted subarrays to produce the sorted array

# Merge Sort - Example

The operation of merge sort on the array  
 $A = \{5, 2, 4, 7, 1, 3, 2, 6\}$



# Merging of sorted subarrays



# Merge Sort - Recursive Algorithm

MERGE-SORT( $A, p, r$ )

1    **if**  $p < r$

2         $q = \lfloor (p + r) / 2 \rfloor$

3        MERGE-SORT( $A, p, q$ )

4        MERGE-SORT( $A, q + 1, r$ )

5        MERGE( $A, p, q, r$ )

Ref: CLRS Book



# MERGE-SORT

- If  $p \geq r$ , the subarray has at most one element and is therefore already sorted.
- Otherwise, the divide step, computes an index  $q$  that partitions  $A[p \dots r]$  into two subarrays  $A[p \dots q]$  and  $A[q+1 \dots r]$  containing  $(n/2)$  elements

MERGE-SORT( $A, p, r$ )

```
1  if  $p < r$ 
2       $q = \lfloor (p + r) / 2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

Function call:

MERGE-SORT( $A, 1, A.length$ )

# Merge sort - Recursive algorithm

- **Base case:**
  - When the size of the subproblem is 1, we don't need to do any further
  - Its already sorted
- **Key operation:** Merging of two sorted arrays in the combine step
- Merge is done by calling another function **Merge (A,p,q,r)**

# Merge function

- Merge is done by calling another function  
**Merge (A,p,q,r)**

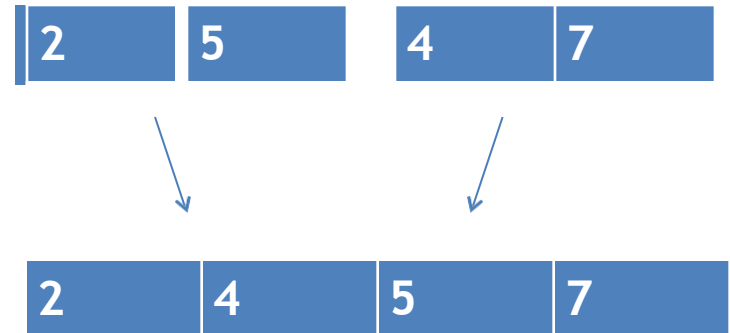
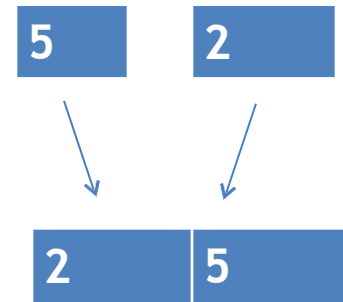
A- Array, p,q,r are indices s.t  $p \leq q < r$

- **Assumption:**  $A[p \dots q]$  and  $A[q+1 \dots r]$  are in sorted order
- **Input:** Array A, indices p, q, and r
- **Output:** Merges  $A[p \dots q]$  and  $A[q+1 \dots r]$  and produce a single sorted subarray  $A[p \dots r]$

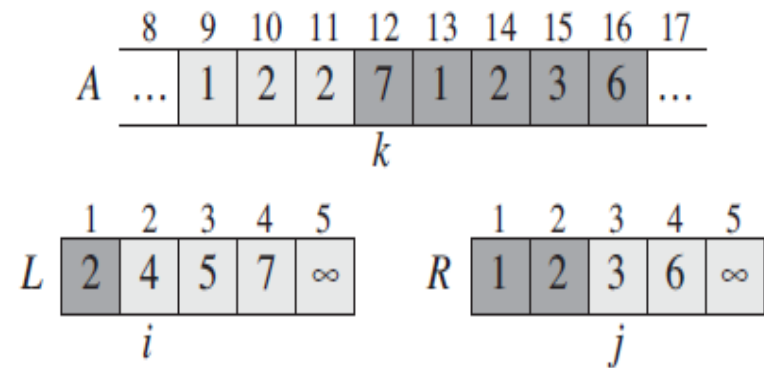
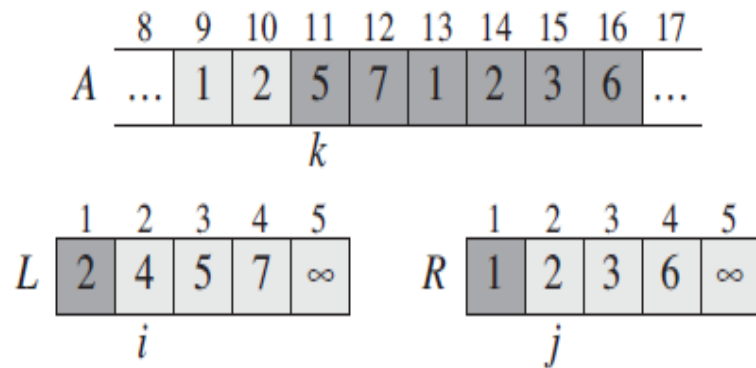
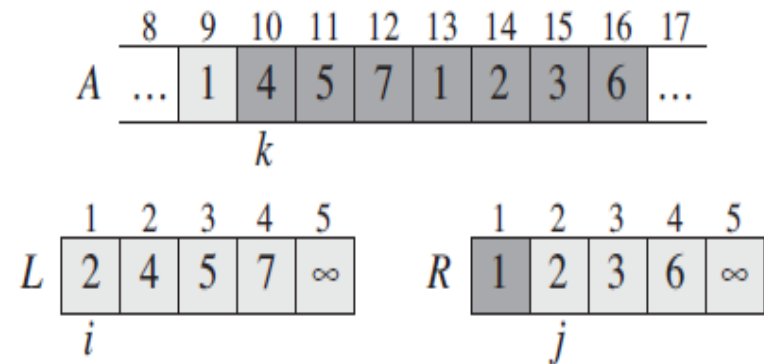
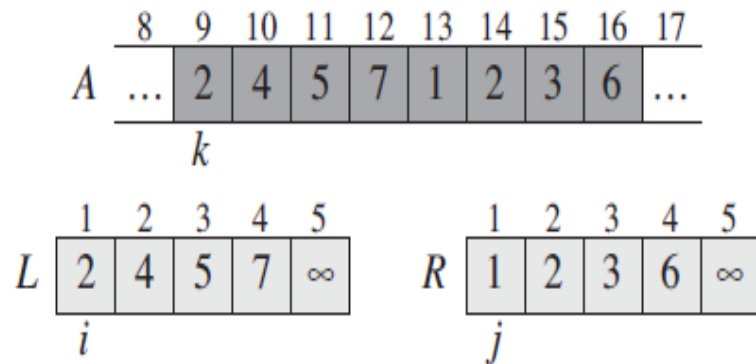
# Merge function

MERGE( $A, p, q, r$ )

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$       Sentinel - a special value
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

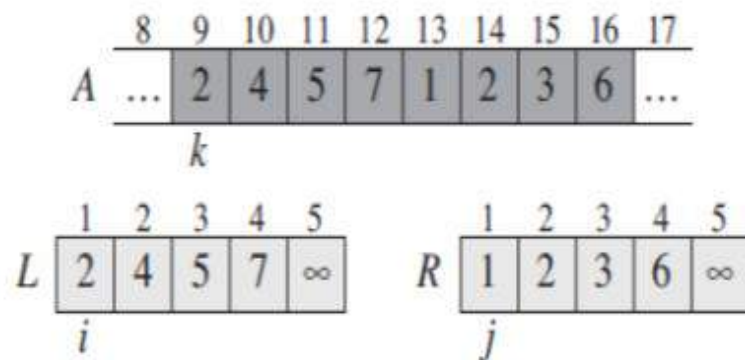


# Working of Merge function

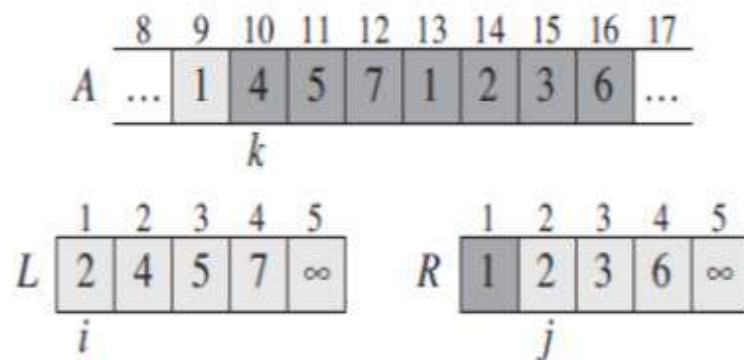


(c)

(d)



(a)



(b)

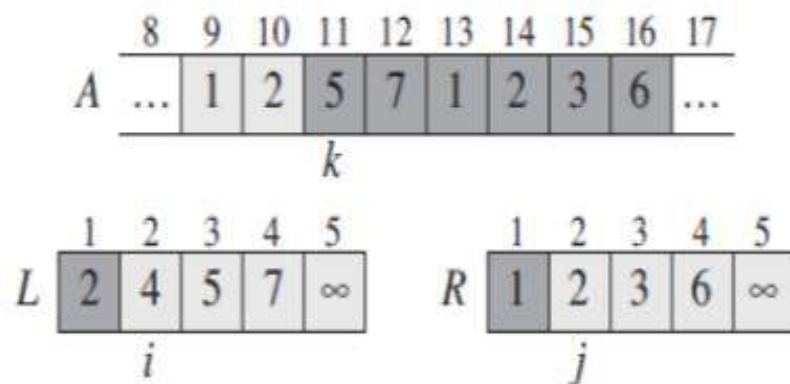
**MERGE**( $A, p, q, r$ )

- 1  $n_1 = q - p + 1$
- 2  $n_2 = r - q$
- 3 let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
- 4 **for**  $i = 1$  **to**  $n_1$
- 5      $L[i] = A[p + i - 1]$
- 6 **for**  $j = 1$  **to**  $n_2$
- 7      $R[j] = A[q + j]$
- 8  $L[n_1 + 1] = \infty$
- 9  $R[n_2 + 1] = \infty$

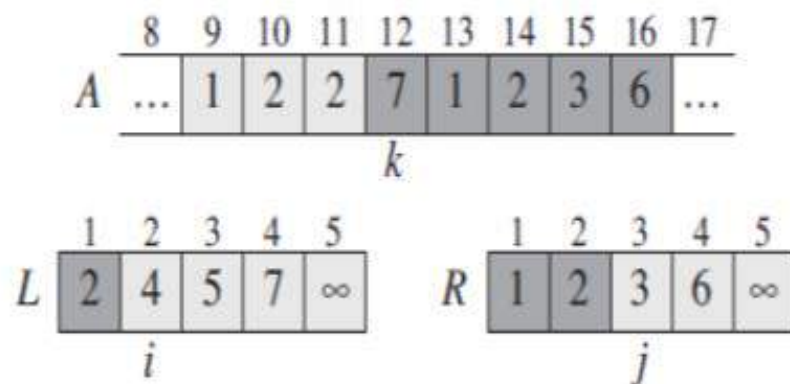
```

10   $i = 1$ 
11   $j = 1$ 
12  for  $k = p$  to  $r$ 
13      if  $L[i] \leq R[j]$ 
14           $A[k] = L[i]$ 
15           $i = i + 1$ 
16      else  $A[k] = R[j]$ 
17           $j = j + 1$ 

```



(c)

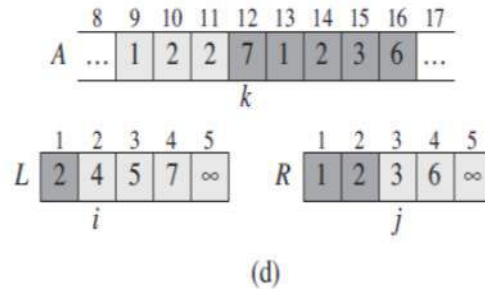
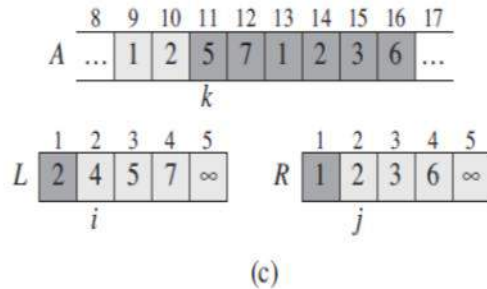


(d)

- Have you understood why do we need to put sentinel values to Left (L) and Right(R) Arrays?



# Correctness of Merge



```

12  for  $k = p$  to  $r$ 
13      if  $L[i] \leq R[j]$ 
14           $A[k] = L[i]$ 
15           $i = i + 1$ 
16      else  $A[k] = R[j]$ 
17           $j = j + 1$ 

```

- **Loop invariant:** At the start of each iteration of the for loop of lines 12-17, the subarray  $A[p \dots k-1]$  contains the  $k - p$  smallest elements of  $L[1.. n_1 + 1]$  and  $R[1.. n_2 + 1]$ , in sorted order.
- Moreover,  $L[i]$  and  $R[j]$  are the smallest elements of their arrays that have not been copied back into A.

# Maintaining the loop invariant

- Show that loop invariant holds **prior to the first iteration** of the *for* loop of lines 12-17
- **Each iteration of the loop** maintains the invariant
- Show correctness when **the loop terminates.**

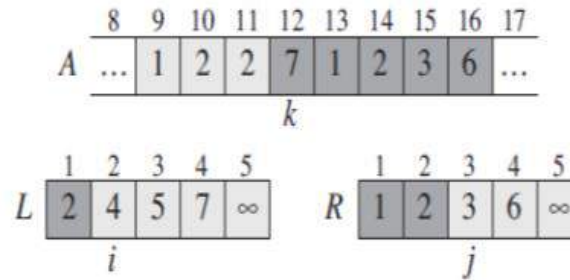
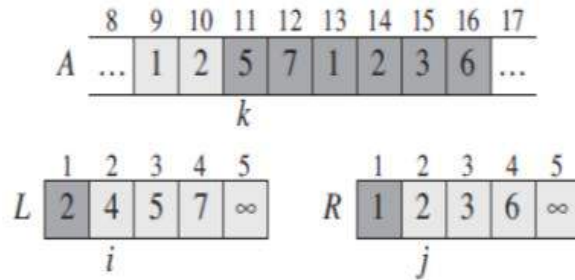
```
12  for  $k = p$  to  $r$   
13      if  $L[i] \leq R[j]$   
14           $A[k] = L[i]$   
15           $i = i + 1$   
16      else  $A[k] = R[j]$   
17           $j = j + 1$ 
```

# Initialization

```

12  for  $k = p$  to  $r$ 
13      if  $L[i] \leq R[j]$ 
14           $A[k] = L[i]$ 
15           $i = i + 1$ 
16      else  $A[k] = R[j]$ 
       $j = j + 1$ 

```



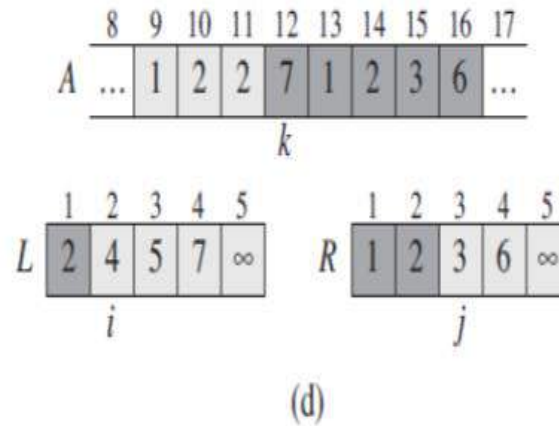
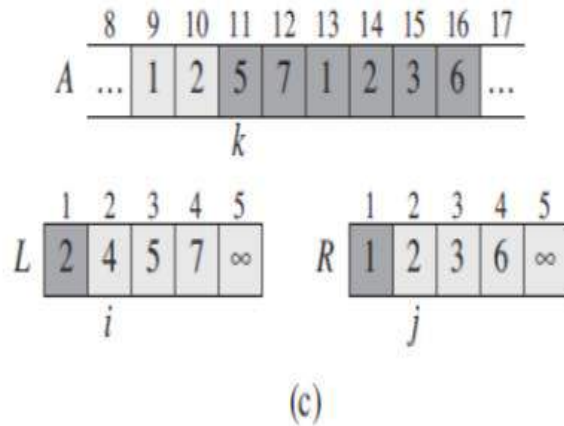
- Prior to the **first iteration** of the loop, we have  $k = p$
- Subarray  $A[p \dots k-1]$  is empty.
- This empty subarray contains the  $k - p = 0$  smallest elements of L and R
- Since  $i = j = 1$ , both  $L[i]$  and  $R[j]$  are the **smallest elements** of their arrays that have not been copied back into A.

**Maintenance:** Each iteration maintains the loop invariant

```

12  for  $k = p$  to  $r$ 
13      if  $L[i] \leq R[j]$ 
14           $A[k] = L[i]$ 
15           $i = i + 1$ 
16      else  $A[k] = R[j]$ 
17           $j = j + 1$ 

```



- First, suppose that  $L[i] \leq R[j]$
- $L[i]$  is the smallest element not yet copied back into A, because  $A[p \dots k-1]$  contains  $k - p$  smallest elements
- After line 14 copies  $L[i]$  into  $A[k]$ , Subarray  $A[p \dots k]$  will contain  $k-p+1$  smallest elements

```

12  for  $k = p$  to  $r$ 
13      if  $L[i] \leq R[j]$ 
14           $A[k] = L[i]$ 
15           $i = i + 1$ 
16      else  $A[k] = R[j]$ 
17           $j = j + 1$ 

```

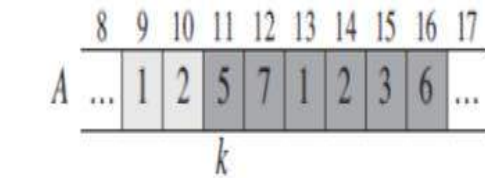
- Incrementing  $k$  (in the **for** loop) and  $i$  (in line 15) reestablishes the loop invariant for the next iteration
- **Suppose if  $L[i] > R[j]$** , lines 16 - 17 perform appropriate action to maintain loop invariant

# Termination

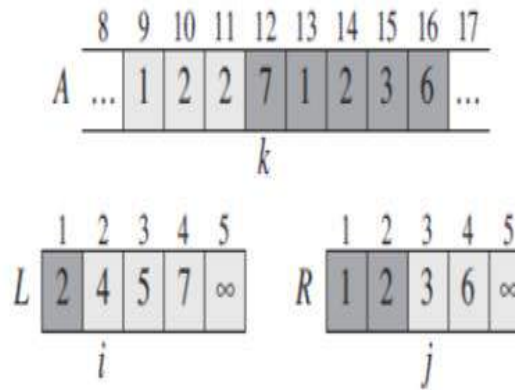
```

12  for  $k = p$  to  $r$ 
13      if  $L[i] \leq R[j]$ 
14           $A[k] = L[i]$ 
15           $i = i + 1$ 
16      else  $A[k] = R[j]$ 
17           $j = j + 1$ 

```



(c)



(d)

- At termination,  $k = r + 1$ .
- By the loop invariant, the subarray  $A[p \dots k-1]$ , which is  $A[p \dots r]$ , contains  $k - p$  smallest elements of  $L[1.. n_1 + 1]$  and  $R[1.. n_2 + 1]$ , in sorted order.
- All but the two largest have been copied back into  $A$ , and these two largest elements are the sentinels.

Reference: CLRS Book

# Recall Merge Sort

**Short comings of Merge sort?**



# Merge Sort - Shortcomings

- Merging L and R arrays create a new array
- No obvious way to efficiently merge *in place*
- Extra storage is required to merge and can be costly
- Merging happens because elements in left half must move right and vice versa

# Motivation - Another Sorting algo / Quick Sort

- Can we divide so that everything to the left is smaller than everything to the right
- No need to Merge

# Without Merging

- Suppose the median value in Array A is  $m$
- Move all values  $\leq m$  to the left half of A
- Move all values  $> m$  to the right half of A
- How much time do you need to do this shifting?

# Without Merging

- Recursively sort left and right halves
- Array A is now sorted
- No need to merge

# Questions

- How do we find the median?
- Sort and pick the middle element
- But, our goal is to sort
- Instead, pick up some value in A i.e **pivot**
- Split A w.r.t the **pivot** element

# Quick Sort Algorithm

# Quick Sort - Introduction

- Tony Hoare - Early 1960's
- Well Known Computer Scientist
- Turing Award Winner

# Quick sort - Idea

- Choose a **pivot** element
- Typically the first/last value in the array is **pivot**
- Divide/Partition the array A into lower and upper parts w.r.t pivot
- Move **pivot** between lower and upper partition
- Recursively sort the two partitions



# Quick Sort : Divide and Conquer

- **Divide :**
  - Partition the array  $A[p..r]$  to two sub arrays  $A[p..q-1]$  and  $A[q+1..r]$ , where  $q$  is computed as part of the *PARTITION* function
  - Each element of  $A[p..q-1]$  is less than or equal to  $A[q]$
  - Each element of  $A[q+1..r]$  is greater than  $A[q]$
  - The sub arrays can be empty or non-empty

# Quick Sort : Divide and Conquer

- **Conquer :**
  - Sort the two sub arrays  $A[p..q-1]$  and  $A[q+1..r]$
  - By recursive calls to quick sort

# Quick Sort : A Divide and Conquer strategy

- **Combine :**
  - The two sub arrays are already sorted
  - The entire array  $A[p..r]$  is already sorted
  - Nothing particular to do in *combine* step

# Pseudo code of Quick Sort

QUICKSORT(A, p, r)

```
1  if p < r
2      then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q-1)
4          QUICKSORT(A, q+1, r)
```

To sort an entire array the initial call is

QUICKSORT(A, 1, A.length)

# How do we partition an array ?

- Suppose array  $A = [9, 7, 5, 11, 12, 2, 14, 3, 10, 6]$
- Array entry  $A[r]$  is selected as the pivot element, where  $r$  is the index of the last element of the array
- $A[r]$  is compared with the array elements until a smaller element  $s$  (less than or equal to pivot) is obtained
- If  $s$  is obtained, it is exchanged with the first element of the array initially
- $A[r]$  is again compared with all other elements of  $A$  and exchange of smaller element is done further

# Working of *PARTITION*

- $A = [9, 7, 5, 11, 12, 2, 14, 3, 10, 6]$
- Pivot =  $A[r] = 6$
- 6 is compared with the elements of A until a smaller element (than 6) is obtained
- In this case, 6 is compared with 9, 7 and 5
- Exchange 9 with 5
- $A = [5, 7, 9, 11, 12, 2, 14, 3, 10, 6]$

# Working of *PARTITION*

- $A = [5, 7, 9, 11, 12, 2, 14, 3, 10, 6]$
- Again 6 is compared with 11, 12 and 2
- 7 and 2 are exchanged
- $A = [5, 2, 9, 11, 12, 7, 14, 3, 10, 6]$
- Again 6 is compared with 14 and 3
- 9 and 3 are exchanged
- $A = [5, 2, 3, 11, 12, 7, 14, 9, 10, 6]$



# Working of *PARTITION*

- $A = [5, 2, 3, 11, 12, 7, 14, 9, 10, 6]$
- Again 6 is compared with 10
- We can see that 5, 2 and 3 are lesser or equal to than the pivot and we have compared pivot with all other elements of A
- Exchange pivot with 11 so that all the elements before pivot is less than or equal to pivot and all the elements after pivot is greater than pivot
- $A = [5, 2, 3, 6, 12, 7, 14, 9, 10, 11]$
- $[5, 2, 3]$  and  $[12, 7, 14, 9, 10, 11]$  are two partitions separated by the pivot 6

# Detailed working of *PARTITION*

p r  
9, 7, 5, 11, 12, 2, 14, 3, 10, 6

p r  
9, 7, 5, 11, 12, 2, 14, 3, 10, 6  
i j

p r  
9, 7, 5, 11, 12, 2, 14, 3, 10, 6  
i j

p r  
9, 7, 5, 11, 12, 2, 14, 3, 10, 6  
i j

p r  
9, 7, 5, 11, 12, 2, 14, 3, 10, 6  
i j

i r  
5, 7, 9, 11, 12, 2, 14, 3, 10, 6  
i j

r  
5, 7, 9, 11, 12, 2, 14, 3, 10, 6  
i j

r  
5, 7, 9, 11, 12, 2, 14, 3, 10, 6  
i j

r  
5, 7, 9, 11, 12, 2, 14, 3, 10, 6  
i j

5, 7, 9, 11, 12, 2, 14, 3, 10, 6  
i j r

5, 2, 9, 11, 12, 7, 14, 3, 10, 6  
i j r

5, 2, 9, 11, 12, 7, 14, 3, 10, 6  
i j r

5, 2, 9, 11, 12, 7, 14, 3, 10, 6  
i j r

5, 2, 9, 11, 12, 7, 14, 3, 10, 6  
i j r

5, 2, 3, 11, 12, 7, 14, 9, 10, 6  
i j r

5, 2, 3, 11, 12, 7, 14, 9, 10, 6  
i j r

5, 2, 3, 6, 12, 7, 14, 9, 10, 11  
i j r

5, 2, 3, 6, 12, 7, 14, 9, 10, 11

5, 2, 3, 6, 12, 7, 14, 9, 10, 11

- [5, 2, 3] and [12, 7, 14, 9, 10, 11] are two partitions separated by the pivot 6
- All the elements before pivot is less than or equal to pivot and all the elements after pivot is greater than pivot

# Design of *PARTITION*

- How many counters/pointers do we need?
  - One counter which goes from  $p$  to  $r-1$  : to compare all other elements of  $A$  with pivot
  - Another counter which keeps track of the position of the smaller element (less than or equal to pivot)

# Pseudo code : *PARTITION*

PARTITION (A, p, r)

1  $x \leftarrow A[r]$

2  $i \leftarrow p - 1$

3 for  $j = p$  to  $r - 1$

4     do if  $A[j] \leq x$

5         then  $i = i + 1$

6             Exchange  $A[i]$  with  $A[j]$

7 Exchange  $A[i + 1]$  with  $A[r]$

8 return  $i + 1$



# Pseudo code of Quick Sort

QUICKSORT(A, p, r)

```
1  if p < r
2      then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q-1)
4          QUICKSORT(A, q+1, r)
```

# Exercise

- Trace the working of *PARTITION* with the sub array [5,2,3]
- Trace the working of *PARTITION* with the sub array [12, 7, 14, 9, 10, 11]
- Trace the working of Quick sort with the array [2 4 6 7 9 23]
- Trace the working of Quick sort with the array [26 24 15 7 3 2]

**Thank You**

# **Algorithm Analysis - Introduction**

# Outline

Introduction

RAM Model

Running time

Analysis - examples

# One Problem - Multiple solutions

- Sorting Problem - Algorithms
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge, Quick, Heap ...
- Which one to choose?
- Basis for your choice?

# Choosing a solution

- Easy to understand and code
- Resource usage - Processor time, Memory Space

# Analyzing the algorithms

- Analyzing the algorithms means predicting the resources the algorithm uses
- What are the resources? - **Computational time and Memory for storage**
- **Why do we analyze algorithms?**
- Analyzing several algorithms for a particular problem results in the most efficient algorithm in terms of computational time/memory



# Calculate Running Time

- Write program, Measure the actual execution time
  - Dependent on implementation - language, machine, compiler,...
  - Nature of input test cases
- Calculate before coding (*Apriori* Analysis)
  - Implementation independent
  - Theoretical-based on a machine model

## Analyzing the algorithms

- A bench mark / model of the implementation technology and the resources of that technology and their costs
- We assume a generic one processor **Random Access Machine** (RAM) model of computation
  - We use RAM as an implementation technology
  - Our algorithms will be implemented as computer programs
  - Instructions are executed one after another, with no concurrent operations

# RAM model

As it will be tedious to define each and every operation of RAM and its costs, we assume a realistic RAM

- RAM contains instructions commonly found in real computers such as:
  - **Arithmetic:** eg: add, subtract, multiply, divide, remainder, floor, ceil
  - **Data movement:** eg: load, store, copy
  - **Control:** conditional & unconditional branch, subroutine call and return

# Random-Access Machine (RAM) model

- Single Processor Machine model
- Instructions for arithmetic, data movement, transfer of control - each taking a constant amount of time
- Instructions executed one after the other, no concurrent operations

## Running time of Instructions:

...

$a = 0$

.....

$b = a$

....

$z = x + y$

....

**return**  $x$

- Each statement is translated to a set of **primitive operations** or **steps**
- Running time depends on the number of primitive operations

## Primitive Operations

Pseudocode statement:  $z = x + y$

Translated code (for a hypothetical machine)

load r1, x // loads contents of memory location x to register r1

load r2, y // loads contents of memory location y to register r2

add r1, r2 // adds contents of r1 and r2, stores result in r1

store z, r1 // // moves data in r1 to memory location z

- $z = x + y$  required 4 machine instructions
- Number of steps different for different types of statements

# Running Time of an Algorithm

- Running time on a particular input is the number of primitive operations or steps executed
- A constant amount of time for each line in the pseudocode
- The  $i^{\text{th}}$  line takes time  $c_i$  where  $c_i$  is a constant

# Example

<b>sample(a, b)</b>	Cost	times
1. $c = a - b$ _____	c1	1
2. $temp = a$ _____	c2	1
3. $a = b$ _____	c3	1
4. $b = temp$ _____	c4	1
5. $return\ c$ _____	c5	1

Running Time =  $c1 + c2 + c3 + c4 + c5$



# Example

Array-Sum(A)	Cost	times
1. sum = 0	c1	1
2. for i = 1 to A.length	c2	n + 1
3. sum = sum + A[i ]	c3	n
4. return sum	c4	1

Running Time = ???

# Example

Array-Sum(A)	Cost	times
1. <code>sum = 0</code> _____	$c_1$	1
2. <code>for i = 1 to A.length</code> _____	$c_2$	$n + 1$
3. <code>sum = sum + A[i]</code> _____	$c_3$	$n$
4. <code>return sum</code> _____	$c_4$	1

Running Time =  $c_1 + c_2(n + 1) + c_3n + c_4$  // *A.length is n*

$$= (c_2 + c_3)n + (c_1 + c_2 + c_4)$$

$$= an + b, \text{ linear}$$

Calculate Running Times of the following algorithms....

### **Linear-Search(A, key )**

1. ***for i = 1 to A.length***
2.       ***if A[i] == key***
3.               ***return i***
4. ***return -1***

Calculate Running Times of the following algorithms....

**Count-Occurrence(A, key )**

1. count = 0
2. **for** i = 1 **to** A. length
3.       if A[i] == key
4.               count = count + 1
5. return count

## Calculate Running Times of the following algorithms....

Count-Occurrence(*A*, *key* )

1 *count* = 0

2 **for** *i* = 1 **to** *A.length*

3     **if** *A*[*i*] == *key*

4         *count* = *count* + 1

5 **return** *count*

$$\text{Running time} = c_1 + c_2(n + 1) + c_3n + c_4n + c_5$$

- ▶ Running time on an input of  $n$  items
- ▶  $T(n)$ , running time as a function of input size,  $n$
- ▶  $T(n) = an + b$ , a linear function of  $n$

## Search-Multiple(A, B)

1. count = 0
2. for i = 1 to B. length
3.     key = B[i ]
4.     for j = 1 to A. length
5.         if A[i] == key
6.             count = count + 1
7. return count

# **Algorithm Analysis: Part-2**

## Calculate Running Times of the following algorithms....

Count-Occurrence( $A$ ,  $key$ )

1  $count = 0$

2 **for**  $i = 1$  **to**  $A.length$

3     **if**  $A[i] == key$

4          $count = count + 1$

5 **return**  $count$

$$Running\ time = c_1 + c_2(n + 1) + c_3n + c_4n + c_5$$

- ▶ Running time on an input of  $n$  items
- ▶  $T(n)$ , running time as a function of **input size**,  $n$
- ▶  $T(n) = an + b$ , **a linear function of  $n$**



## **Search-Multiple(A, B)**

1. count = 0
2. for i = 1 to B. length
3.     key = B[i ]
4.     for j = 1 to A. length
5.         if A[i] == key
6.             count = count + 1
7. return count

## Calculate Running Times of the following algorithms...

Search-Multiple(A, B)	cost	Times
1.   count = 0	c1	1
2.   for i = 1 to B. length	c2	m+1
3.       key = B[i ]	c3	n
4.       for j = 1 to A. length	c4	m(n+1)
5.           if A[i] == key	c5	mn
6.               count = count + 1	c6	mn
7.   return count	c7	1

Note: A.length = n, B.length = m

## Running Time = ?

# Cost and Times

- To analyse an algorithm, sum up the cost of each and every line of the pseudocode
- To compute the **cost** of one line, we will take the time taken to execute that line (i.e the cost incurred to execute that line) multiplied by how many **times** that line is executed
- Usually we write the running time as a function of  $n$ , represented as  $T(n)$ , where  $n$  is the input size of the problem

# Analysis of algorithm

- The **time taken** by an algorithm **grows with the input size**
- **Running time** of an algorithm is described as **a function of its input**
- We will formalize “**input size**” and “**running time**”
- **Input size**
  - Depends on the problem being studied
  - eg: for sorting problem, it is the number of elements
  - eg: for multiplying two numbers, it is the total number of bits

# Pseudocode of Linear Search

## **LINEAR SEARCH(A, key)**

1. found = 0
2. for i = 1 to A.length
3.       if A[ i ] = key
4.             found = 1
5.       return i
6. if found = 0
7.    return 0

What is the running time of Linear Search?

- Consider the different cases of the input
  - Best case
  - Worst case

## Best Case of Linear Search

- **Best Case input of Linear Search** : The element to be searched is in the first position of the list
  - Eg: A= 1, 4, 2, 7, 10, 5 & key = 1
- How do we **analyse the linear search in the best case?**
- Step 1: Cost :  $c_1$ , Times : 1
- Step 2: Cost :  $c_2$ , Times : 1
- Step 3: Cost :  $c_3$ , Times : 1
- Step 4: Cost :  $c_4$ , Times : 1
- Step 5: Cost :  $c_5$ , Times : 1
- $T(n) = c_1 + c_2 + c_3 + c_4 + c_5$

### **LINEAR SEARCH(A,key)**

1. found = 0
2. for i = 1 to A.length
3.           if A[ i ] = key
4.           found = 1
5.           return i
6. if found = 0
7.   return 0

## Worst Case of Linear Search

- **One of the Worst Case input of Linear Search** : The element to be searched is in the last position of the list
  - Eg: A = 1, 4, 2, 7, 10, 5 & key = 5
- How do we analyse the linear search in the worst case – successful search?
- Step 1: Cost :  $c_1$ , Times : 1
- Step 2: Cost :  $c_2$ , Times : n
- Step 3: Cost :  $c_3$ , Times : n
- Step 4: Cost :  $c_4$ , Times : 1
- Step 5: Cost :  $c_5$ , Times : 1
- $T(n) = c_1 + n * c_2 + n * c_3 + c_4 + c_5$

### **LINEAR SEARCH(A,key)**

1. found = 0
2. for i = 1 to A.length
3.           if A[ i ] = key
4.           found = 1
5.           return i
6. if found = 0
7.   return 0

# Analysis of Worst Case of Linear Search- Unsuccessful search

- **One of the Worst Case input of Linear Search** : The unsuccessful search analysis ie. the element is not present
  - Eg:  $A = 1, 4, 2, 7, 10, 5$  &  $key = 0$
- Step 1- Cost :  $c_1$ , Times : 1
- Step 2- Cost :  $c_2$ , Times :  $n+1$
- Step 3- Cost :  $c_3$ , Times :  $n$
- Step 4- Cost :  $c_4$ , Times : 0
- Step 5- Cost :  $c_5$ , Times : 0
- Step 6- Cost :  $c_6$ , Times : 1
- Step 7- Cost :  $c_7$ , Times : 1
- $T(n) = c_1 + (n+1)*c_2 + n*c_3 + c_6 + c_7$

## **LINEAR SEARCH(A,key)**

- 1. found = 0**
- 2. for i = 1 to A.length**
- 3.           if A[ i ] = key**
- 4.           found = 1**
- 5.           return i**
- 6. if found = 0**
- 7.   return 0**



# Running time of Linear Search algorithm

- Considering the different cases of the input
  - Best case running time:  $T(n) = c_1 + c_2 + c_3 + c_4 + c_5$
  - Worst case running time:
    - Successful Search:  $T(n) = c_1 + n * c_2 + n * c_3 + c_4 + c_5$
    - Unsuccessful Search:  $T(n) = c_1 + (n+1) * c_2 + n * c_3 + c_6 + c_7$

# **Insertion Sort - Analysis**

# Analysis of algorithms

- We know : Analysing the algorithms means **predicting the resources the algorithm uses**
- We focus on the resource : **computational time**
- The time taken by the insertion sort depends on what?
  - **input size**
- Does the time taken depend on anything else?
  - **how sorted is the input already**

# Run-time Analysis

- Time taken by Insertion Sort (IS) algorithm depends on the input size
  - Sorting a million numbers takes longer than sorting ten numbers
- IS take different amounts of time to sort two input sequences of the same size
  - Depends on how nearly sorted they already are

# Run-time Analysis

- Time taken by the algorithm grows with the size of the input,

Eg:  $n = 10$ , Running time = 1 unit

$n = 10000$ , Running time = 1000 units

- Running time as a function of the size of its input

# Input Size

- **Sorting and Searching**

Number of elements in the input

- **GCD of two numbers / Number is a prime number or not ?**

Total number of bits needed to represent the input in binary notation

- **Graph problems**

Number of vertices and edges

# Running time

- Number of primitive operations or steps executed
- Steps – machine independent
- Assumption (RAM model)
  1. Constant amount of time is required to execute each line of pseudocode
  2. Each execution of  $i^{\text{th}}$  line takes time  $c_i$ , where  $c_i$  is a constant

## INSERTION-SORT(A)

- |   |       |
|---|-------|
| 1. <b>for</b> $j = 2$ to $A.length$                         | $C_1$ |
| 2. $key = A[j]$ ;   | $C_2$ |
| 3.   // Insert $A[j]$ into the sorted sequence $A[1...j-1]$ |       |
| 4. $i = j-1$  | $C_3$ |
| 5. <b>while</b> $i > 0$ and $A[i] > key$                    | $C_4$ |
| 6. <b>do</b> $A[i+1] = A[i]$                                | $C_5$ |
| 7. $i = i - 1$  | $C_6$ |
| 8. $A[i+1] = key$   | $C_7$ |



# **Algorithm analysis: Part - 3**

- **Insertion Sort Analysis**
- **Asymptotic Notations - Introduction**

# While loop within for loop

- **For/while loop** : Testing condition is executed one time more than the loop body

## INSERTION-SORT(A)

- |   |       |
|---|-------|
| 1. <b>for</b> $j = 2$ to $A.length$                         | $C_1$ |
| 2. $key = A[j]$ ;   | $C_2$ |
| 3.   // Insert $A[j]$ into the sorted sequence $A[1...j-1]$ |       |
| 4. $i = j-1$  | $C_3$ |
| 5. <b>while</b> $i > 0$ and $A[i] > key$                    | $C_4$ |
| 6. <b>do</b> $A[i+1] = A[i]$                                | $C_5$ |
| 7. $i = i - 1$  | $C_6$ |
| 8. $A[i+1] = key$   | $C_7$ |

- **Let  $t_j$  be the number of times** the while loop in line 5 is executed
- Since **while loop** is within a for loop, for each  $j = 2, 3, \dots, n$ , where  $n = A.length$ , **total number of times while loop executed is  $\sum_{j=2 \text{ to } n} t_j$**

# 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  $A[1..j-1]$

4.      $i = j-1$

$c_3$

$n - 1$

5.     **while  $i > 0$  and  $A[i] > key$**

**$c_4$**

**$\sum_{j=2}^n t_j$**

6.          $A[i+1] = A[i]$

$c_5$

$\sum_{j=2}^n (t_j - 1)$

7.          $i = i-1$

$c_6$

$\sum_{j=2}^n$

$(t_j - 1)$

8.      $A[i+1] = key$

$c_7$

$n - 1$

# Running time of an algorithm

- Sum of the running times for each statement executed  
a statement that takes  **$c_i$  steps** to execute and is executed  **$n$  times**, **contribute  $c_i * n$**  to the total running time
- $T(n)$  – running time of IS : sum of the products of the cost and times
- $T(n) = ?$

What do you think is the best case for IS?

Input:

1,2,3,4,5,6,7,8,9,10

Input:

10,9,8,7,6,5,4,3,2,1

# Best case of IS – Already sorted array

- For each  $j = 2, 3, \dots, n$ , we know that  $A[i] \leq \text{key}$  in line 5,  $i$  has its initial value of  $j - 1$   
i.e  $A[1] \leq 2$ , for  $j = 2$ ,  $A[2] \leq 3$ , for  $j = 3$ , .....
- Condition is FALSE and the body of the while loop will not be executed
- But, the condition in while loop alone will be executed, therefore,  $t_j = 1$ , for  $j = 2, 3, \dots, n$

- Best case running time:

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 (n-1) + c_7 (n-1)$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$



# Best case of IS - Linear function

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

=  $a n + b$ , where  $a$  and  $b$  depend on the statement costs  $c_i$

It's a linear function of  $n$

# Worst case of IS - reverse sorted

Input: 10,9,8,7,6,5,4,3,2,1

Compare each element  $A[j]$  with each element in the entire sorted subarray  $A[1 \dots j-1]$

i.e for  $j = 2$ ,  $A[2]$  will be compared with  $A[1]$

Resultant array: 9,10,8,7,6,5,4,3,2,1

for  $j = 3$ ,  $A[3]$  will be compared with  $A[2]$  and  $A[1]$

Resultant array: 8, 9,10,7,6,5,4,3,2,1

$$T(n) = c_1n + c_2(n - 1) + c_3(n - 1) + c_4 \sum_{j=2}^n t_j \\ + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n - 1)$$

- ▶ Compare each  $A[j]$  with each element in  $A[1 \dots j - 1]$
- ▶  $t_j = j$  for  $j = 2, 3 \dots n$
- ▶ Worst-case running time

$$T(n) = an^2 + \underline{\underline{bn}} + c, \text{ a quadratic function of } n$$

What is  $t_j$  for the worst case?

$t_j = j$ , for  $j = 2, 3, \dots, n$

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

and

$$\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

# Worst case running time

$T(n) = a n^2 + b n + c$ , for constants  $a$ ,  $b$  and  $c$   
that depends on the statement costs  $c_i$

$T(n)$  quadratic function of  $n$

# Worst case running time

- Longest running time for any input of size  $n$
- Upper bound on the running time for any input
- Provides a guarantee that the algorithm will not take more than the specified value
- Worst case occurs fairly often – Searching a database, information is not present

- **Best-case running time**
  - Smallest running time for any input of size  $n$
  - Gives a lower bound on the running time of an algorithm

# Average case

- As bad as the worst case
- Randomly choose  $n$  numbers and apply IS
- How long does it take to insert element  $A[j]$  in the sorted subarray  $A[1 \dots j-1]$ ?
- On the average, half the elements are less than  $A[j]$  and half the elements are greater than  $A[j]$
- Hence, we check half of the subarray  $A[1 \dots j-1]$ .
- Therefore,  $t_j$  is  $j/2$
- What is the average case running time?

Quadratic function in the size of the input



# CS2002D Program Design: Monsoon 2021

## Asymptotic Notations

September 10, 2021

Acknowledgements: Dr. Saleena N, CSED

# Rate of Growth or Order of Growth

- Rate/Order of growth – Considering the leading term of a formula
- Ignoring the lower order terms - insignificant for large values of  $n$
- Ignoring leading term's constant coefficient – constant factors are less significant

## Rate of Growth or Order of Growth

- ▶  $T(n) = an^2 + bn + c$
- ▶ we say  $T(n)$  is  $\Theta(n^2)$  (“theta of  $n$ -squared”)
  - simplifying abstraction
  - consider only the leading term of a formula
  - lower order terms- relatively insignificant for large values of  $n$
  - ignore the leading term’s constant coefficients

# Asymptotic Efficiency

- ▶ input size is large enough, only the order of growth is relevant
- ▶ asymptotic efficiency - how the running time increases with the size of the input *in the limit*
- ▶ asymptotically more efficient - best choice for all but very small inputs

# Asymptotic Notations

- ▶ Domain of functions - Set of Natural Numbers  
 $\mathbb{N} = 0, 1, 2, \dots$  (as given by CLRS [1])
  - $T(n)$  usually defined only on integer input sizes
- ▶  $T(n) = an^2 + bn + c$ 
  - $T(n)$  is  $\Theta(n^2)$  ("theta of  $n$ -squared")
  - $T(n)$  is  $O(n^2)$  ("Big - Oh of  $n$ -squared")
  - $T(n)$  is  $\Omega(n^2)$  ("Omega of  $n$ -squared")

## O notation (big-Oh)

- ▶  $T(n) = n^2 + 2n + 1$  for  $n > 1$ ,  $T(1) = 4$
- ▶  $T(n) \leq 4n^2$ , for  $n \geq 1$
- ▶  $T(n) \leq cn^2$ , for  $n \geq n_0$  ( $c=4$  and  $n_0=1$ )
- ▶ we say  $T(n)$  is  $O(n^2)$

## O notation (big-Oh)

- ▶  $T(n) = n^2 + 2n + 1$  for  $n > 1$ ,  $T(1) = 4$
- ▶ How to get  $c$  and  $n_0$  such that  $T(n) \leq cn^2$ , for  $n \geq n_0$ 
  - $c$  should be such that  $n^2 + 2n + 1 \leq cn^2$
  - divide by  $n^2$ ,  $1 + \frac{2}{n} + \frac{1}{n^2} \leq c$
  - for  $n \geq 1$ , we can choose  $c \geq 4$
- ▶  $T(n) \leq cn^2$ , for  $n \geq n_0$  ( $c=4$  and  $n_0=1$ )
- ▶ we say  $T(n)$  is  $O(n^2)$
- ▶  $n^2 + 2n + 1$  is  $O(n^2)$

# O notation

- ▶  $T(n)$  is  $O(n^2)$ 
  - There are positive constants  $c$  and  $n_0$  such that  $T(n) \leq cn^2$  for  $n \geq n_0$



# Some functions:

$$T_1(n) = 5n^2$$

$$T_2(n) = n^2 + 2n$$

$$T_3(n) = n + 5$$

- $T_1(n) \leq 5n^2$  for  $n \geq 1$

- $T_2(n) \leq 2n^2$  for  $n \geq 2$

- $T_3(n) \leq n^2$  for  $n \geq 3$

# Generalizing.....

- *There exists positive constants  $c=5$  and  $n_0=1$  such that  $T_1(n) \leq cn^2$  for  $n \geq n_0$*
- *There exists positive constants  $c=2$  and  $n_0=2$  such that  $T_2(n) \leq cn^2$  for  $n \geq n_0$*
- *There exists positive constants  $c=1$  and  $n_0=3$  such that  $T_3(n) \leq cn^2$  for  $n \geq n_0$*

*There exists positive constants  $c$  and  $n_0$  such that  $f(n) \leq cn^2$  for  $n \geq n_0$*

The set  $O(n^2)$  (read “big oh of  $n^2$ ” or “oh of  $n^2$ ”)

- Set of all  $f(n)$  such that there exists positive constants  $c$  and  $n_0$  such that  $f(n) \leq cn^2$  for all  $n \geq n_0$

- Set is denoted by  $O(n^2)$

- $T_1(n) \in O(n^2)$        $T_2(n) \in O(n^2)$        $T_3(n) \in O(n^2)$

- $O(n^2)$  is a set of functions.

$$O(n^2) = \{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cn^2 \text{ for all } n \geq n_0 \}$$

# The set $O(n^2)$ – contd.

- Give some more functions that belong to the set  $O(n^2)$ .

- $f_1(n) = 100n^2 + n + 5$

- $f_2(n) = 6n + 3$

- $f_3(n) = 10000n^2$

# The set $O(n^3)$

- $O(n^3) = \{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cn^3 \text{ for all } n \geq n_0 \}$
- Some of the elements of  $O(n^3)$ 
  - $f_4(n) = 100n^3 + 3n^2 + 2$
  - $f_5(n) = 6n + 3$
  - $f_6(n) = 10000n^2$

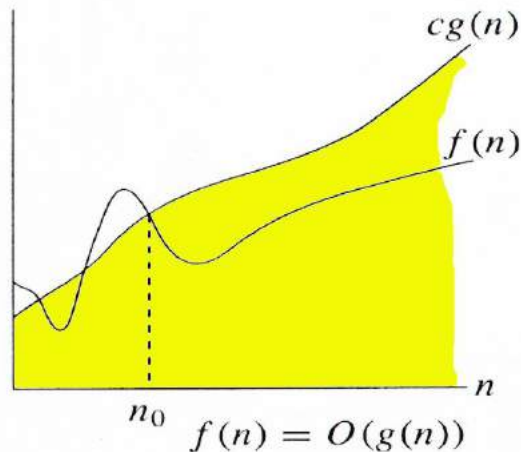
# Generalizing....

- $O(n) = \{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c n \text{ for all } n \geq n_0 \}$
- $O(n \lg n) = \{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c n \lg n \text{ for all } n \geq n_0 \}$

$O(g(n)) = \{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \}$

$O$ -notation gives an **upper bound for a function** to within a constant factor.

$f(n) = O(g(n))$ , if there are positive constants  $c$  and  $n_0$  such that to the right of  $n_0$ , the value of  $f(n)$  always lies on or below  $cg(n)$ .



Source: <http://www.cs.unc.edu/~plaisted/comp122/02-asymp.ppt>

# Going back ....

## ■ Insertion Sort

□ *Worst Case Running time is  $O(n^2)$*

■ *Worst Case Running time,  $T_1(n) \leq cn^2$  for all values of  $n \geq n_0$  where  $c$  and  $n_0$  are positive constants.*

*Normally we write  $f(n)$  is  $O(g(n))$  or  $f(n)=O(g(n))$  to mean “ $f(n)$  is a member of  $O(g(n))$ ”*



Prove  $T(n) = n^3 + 20n + 1$  is  $O(n^3)$

- by the Big-Oh definition,  $T(n)$  is  $O(n^3)$  if  $T(n) \leq c \cdot n^3$  for some  $n \geq n_0$
- Find out  $c$  and  $n_0$

## Exercises

1. Is  $2n + 10 \in O(n^2)$  ?

2. Is  $n^3 \in O(n^2)$ ?

# The set $\Omega(n)$ (*Read big-omega of n*)

- An example[1]

$$T(n) = 2n + 3$$

$$2n \leq T(n) \text{ for } n \geq 1$$

$$cn \leq T(n) \text{ for } n \geq 1 \text{ and } c = 2$$

$T(n)$  belongs to  $\Omega(n)$

$\Omega(n) = \{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cn \leq f(n) \text{ for all } n \geq n_0 \}$

# Exercises

1. Is  $2n + 1 \in \Omega(n)$ ?
2. Is  $2n^2 + 10 \in \Omega(n^2)$  ?
3. Is  $n^3 \in \Omega(n^2)$ ?

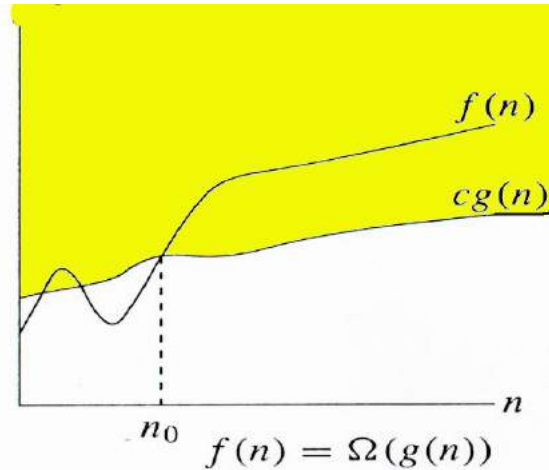
The set  $\Omega(g(n))$

$\Omega(g(n))$

*$= \{ f(n) : \text{there exists positive constants } c$   
 $\text{and } n_0 \text{ such that } 0 \leq c g(n) \leq f(n) \text{ for}$   
 $\text{all } n \geq n_0 \}$*

$\Omega$ -notation gives a **lower bound** for a function to within a constant factor

*$f(n) = \Omega(g(n))$ , if there are positive constants  $c$  and  $n_0$  such that to the right of  $n_0$ , the value of  $f(n)$  always lies on or above  $cg(n)$ .*



Source: <http://www.cs.unc.edu/~plaisted/comp122/02-asympt.ppt>

# The set $\Theta(n)$

- An example[1] -  $T(n) = 2n + 3$

$$T(n) \leq 6n \quad \text{for } n \geq 1 \quad T(n) \text{ is } O(n)$$

$$2n \leq T(n) \quad \text{for } n \geq 1 \quad T(n) \text{ is } \Omega(n)$$

***$T(n)$  belongs to  $\Theta(n)$***

***$\Theta(n) = \{ f(n): \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 n \leq f(n) \leq c_2 n \text{ for all } n \geq n_0 \}$***

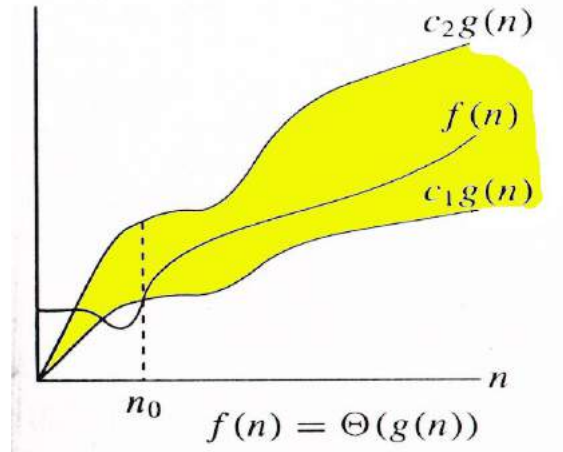
The set  $\Theta(g(n))$

$$\Theta(g(n)) = \{ f(n) : \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all} \\ n \geq n_0 \}$$



**$\Theta$** -notation gives **tight bound** for a function to within constant factors

$f(n) = \Theta(g(n))$ , if there exists positive constants  $c_1$ ,  $c_2$  and  $n_0$  such that to the right of  $n_0$ , the value of  $f(n)$  always lies between  $c_1 g(n)$  and  $c_2 g(n)$  inclusive.

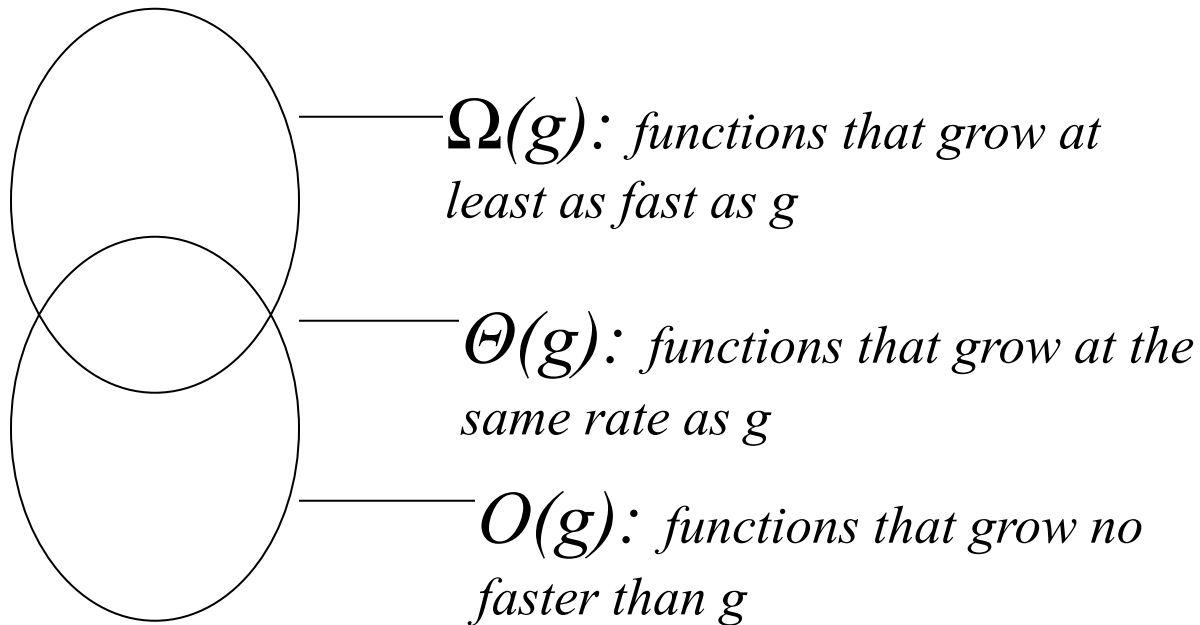


Source: <http://www.cs.unc.edu/~plaisted/comp122/02-asymp.ppt>

# Asymptotic notations – Formal definitions [1]

- $O(g(n)) = \{f(n): \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0\}$
- $\Omega(g(n)) = \{f(n): \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c g(n) \leq f(n) \text{ all } n \geq n_0\}$
- $\Theta(g(n)) = \{f(n): \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$

# To make it more clear [2]



- $f(n) = \Theta(g(n))$ 
  - $g(n)$  is an asymptotically tight bound for  $f(n)$
- $f(n) = O(g(n))$ 
  - $g(n)$  is an asymptotic upper bound for  $f(n)$
- $f(n) = \Omega(g(n))$ 
  - $g(n)$  is an asymptotic lower bound for  $f(n)$

## *Theorem [1]*

For any two functions  $f(n)$  and  $g(n)$ , we have  
 $f(n) = \Theta(g(n))$  if and only if  $f(n) = O(g(n))$  and  
 $f(n) = \Omega(g(n))$ .

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

# Examples[1]

- $f(n) = an^2 + bn + c$ , where  $a$ ,  $b$ , and  $c$  are constants and  $a > 0$
- $f(n) = \Theta(n^2) \rightarrow f(n) = \Omega(n^2) \text{ and } f(n) = O(n^2)$

- For any polynomial,  $p(n)$  of degree  $k$  we have  $p(n) = \Theta(n^k)$
- Any constant function is  $\Theta(n^0)$ , or  $\Theta(1)$ .

# Insertion Sort – Running Time

- Best Case running Time is  $\Omega(n)$ .
  - Implies Running time on any input is  $\Omega(n)$ .
- Running time is not  $\Omega(n^2)$ .
- Worst Case running time is  $\Omega(n^2)$ .
- Is it correct to say best case running Time is  $\Theta(n)$ ?

# Is $O(n \lg n)$ algorithm preferred over $O(n^2)$ ?

- Suppose  $T_1(n) \leq 50 n \lg n$  and  $T_2(n) \leq 2n^2$
- Check the values of  $T_1(n)$  and  $T_2(n)$  when  $n=2$  and  $n=1024$
- For small input sizes, the  $O(n^2)$  algorithm may run faster.
- Once the input size becomes large enough,  $O(n \lg n)$  runs faster
  - irrespective of the constant factors
  - irrespective of the implementation.
- Read the corresponding Sections in CLRS.



# References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein *Introduction to Algorithms*, PHI, 2001.
2. Sara Baase and Allen Van Gelder *Computer Algorithms: Introduction to Design & Analysis*, Pearson Education, third edition, 2000.
3. Donald E Knuth. Big omicron and big omega and big theta. *ACM SIGACT News*, 1976.
4. Gilles Brassard, Paul Bratley, *Fundamentals of Algorithmics*, PHI, 1997.

# Analysis – Recursive Algorithms: Part 1

# Overview

- **Asymptotic Notations – Review**
- **Recurrence relation**
  - **Factorial**
  - **Binary Search**
  - **Merge Sort**
- **Solution of recurrence**

# Asymptotic Notations - Review

- **O (Oh),  $\Omega$  (Omega ),  $\theta$  (Theta)**
- **Definitions (sets)**
- **Insertion Sort – Analysis**
  - **Worst Case**
  - **Best Case**

# Asymptotic Notations - Exercises

1. Is  $3n^2 - 100n + 6$  is  $O(n^3)$  ?
2. Is  $3n^2 - 100n + 6$  is  $\Omega(n^3)$  ?
2. Which algorithm do you prefer?
  - a.  $\Theta(n^2)$  or  $\Theta(n)$
  - b.  $\Theta(n)$  or  $\Theta(\lg n)$

# Factorial - Recursive function

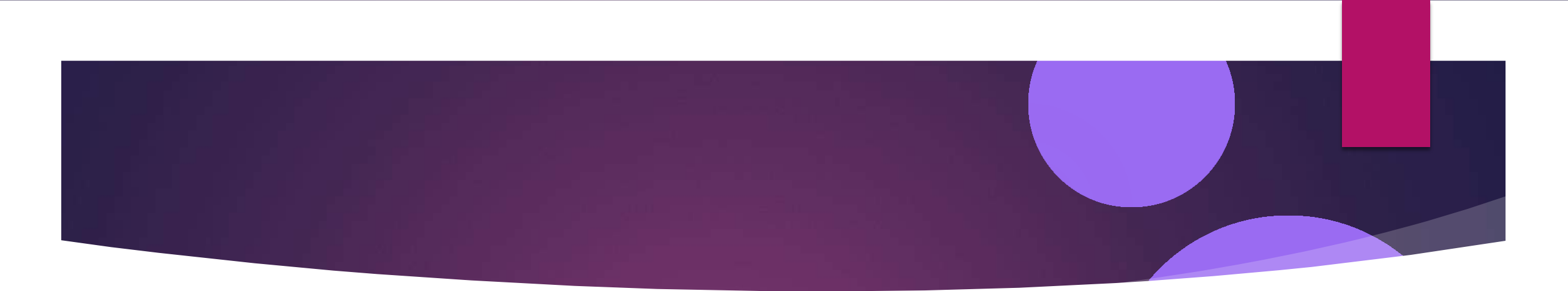
```
int factorial (int n){  
    // returns the factorial of n, given n>=0  
    if (n<=1)  
        return 1;  
    else  
        return n * factorial (n-1);  
}
```

Running Time  $T(n) = ?$

# Factorial – Running Time

$$\begin{aligned} T(n) &= T(n-1) + c && \text{if } n > 1 \\ &= d && \text{if } n \leq 1 \end{aligned}$$

Asymptotic Running Time?



How do we express the running time of binary search?



# Binary Search – Algorithm

```
BinarySearch (A, m, n, k)
    if (m>n) return -1;    //Base Case
    mid=(m+n)/2
    if A[mid]=k return mid;    //Base Case
    else if k < A[mid]
        BinarySearch(A, m, mid-1, k)
    else if k > A[mid]
        BinarySearch(A, mid+1, n, k)
```

# Binary Search – Running Time

$$\begin{aligned} T(n) &= T(n/2) + c && \text{if } n > 1 \\ &= d && \text{if } n \leq 1 \end{aligned}$$



How do we express the running time of merge sort?

# Merge Sort - Recursive Algorithm

MERGE-SORT( $A, p, r$ )

1    **if**  $p < r$

2         $q = \lfloor (p + r) / 2 \rfloor$

3        MERGE-SORT( $A, p, q$ )

4        MERGE-SORT( $A, q + 1, r$ )

5        MERGE( $A, p, q, r$ )

# Merge Sort – Running Time

$$\begin{aligned} T(n) &= 2T(n/2) + cn && \text{if } n > 1 \\ &= c && \text{if } n = 1 \end{aligned}$$

# Running Time - Recurrence equation

- Running time of recursive algorithms described by a **recurrence equation** or **recurrence**
- $T(n)$  in terms of running times of smaller subproblems
- Solve the recurrence using mathematical tools to get bounds on the running time

# Solving recurrence - Factorial

$$\begin{aligned} T(n) &= T(n-1) + c && \text{if } n > 1 \\ &= d && \text{if } n \leq 1 \end{aligned}$$

# Solving recurrence - Factorial

$$T(n) = c + T(n-1) \quad \text{if } n > 1$$



# Solving recurrence - Factorial

$$T(n) = c + T(n-1) \quad \text{if } n > 1$$

$$T(n-1) = c + T(n-2) \quad \text{if } n > 2$$

$$T(n) = c + c + T(n-2) \quad \text{if } n > 2$$

$$= 2c + T(n-2) \quad \text{if } n > 2$$

# Factorial – Running Time

$$T(n) = 2c + T(n-2) \quad \text{if } n > 2$$

$$T(n) = 3c + T(n-3) \quad \text{if } n > 3$$

**In general ?**

# Factorial – Running Time

$$T(n) = 2c + T(n-2) \quad \text{if } n > 2$$

$$T(n) = 3c + T(n-3) \quad \text{if } n > 3$$

In general,

$$T(n) = ic + T(n-i) \quad \text{if } n > i$$

when  $i = n-1$ ,

$$T(n) = (n-1)c + T(1) = (n-1)c + d = cn - c + d$$

**$T(n)$  is  $\Theta(n)$**

# Solving Recurrence – Iteration method

- Expand (iterate) the recurrence
- Express as a summation of terms dependent only on  $n$
- **Recursion Tree** - Visualize the iteration of recurrence

# Divide and Conquer – Recurrence

$$\begin{aligned} T(n) &= \Theta(1) && \text{if } n \leq c \\ &= a T(n/b) + D(n) + C(n) && \text{otherwise} \end{aligned}$$

- Number of subproblems –  $a$
- Each subproblem size is  $1/b$  the size of the original
- $D(n)$  – time to divide the problem into subproblems
- $C(n)$  – time to combine the solutions

# Divide and Conquer – Recurrence

$$\begin{aligned} T(n) &= d && \text{if } n \leq 1 \\ &= 2 T(n/2) + c && \text{otherwise} \end{aligned}$$

Solve using iteration method

# Reference

T H Cormen, C E Leiserson, R L Rivest, C Stein *Introduction to Algorithms*, 3<sup>rd</sup> ed., PHI, 2010