Data Structures (CS 21001)

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY

School Of Computer Engineering

Dr. Pradeep Kumar Mallick
Associate Professor [II]
School of Computer Engineering,
Kalinga Institute of Industrial Technology (KIIT),
Deemed to be University,Odisha

4 Credit

Lecture Note 01

Course Description

- Provide **solid foundations** in **basic concepts of problem solving** – both data structures and algorithms
- **Select and design data structure & algorithms** that are **appropriate** for the given problems
- **Demonstrate** the **correctness** of the algorithm and **analyzing** their **computational complexities**

How?

Blend of Theory and Practical

Prerequisites

- Programming in C (CS 1001)
- Mathematics for Computer Science

What is data?

- **Data**
  - A collection of facts from which conclusion may be drawn.
  - e.g. Data: Temperature 35°C; Conclusion: It is hot.

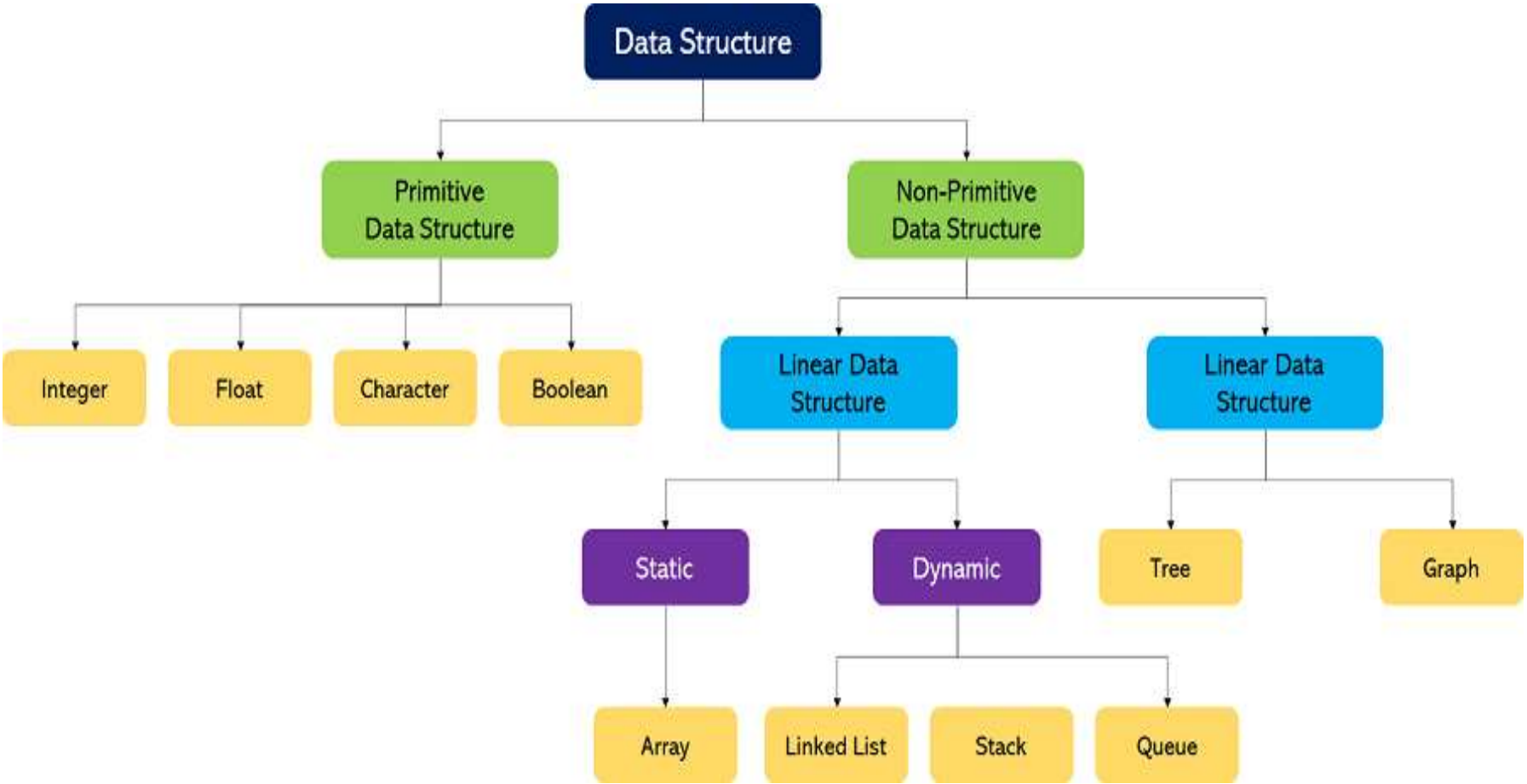
    (Or)
  - Data can be defined as a representation of facts, concepts, or instructions in a formalized manner, which should be suitable for communication, interpretation, or processing by human or electronic machine.
  - Generally data is **raw and unprocessed.**

- **Types of data**
  - Textual: For example, your name (Alya)
  - Numeric: For example, your ID (090254)
  - Audio: For example, your voice
  - Video: For example, your voice and picture
  - (...)

What is data structure?

- A particular way of storing and organizing data in a computer so that it can be used efficiently and effectively.
- Data structure is the logical or mathematical model of a particular organization of data.
- A group of data elements grouped together under one name.
  - For example, an array of integers

Data Structure Classification



Primitive Data Structures

- **Primitive Data Structures** are the data structures consisting of the numbers and the characters that come **in-built** into programs.
- These data structures can be manipulated or operated directly by machine-level instructions.
- Basic data types like **Integer, Float, Character**, and **Boolean** come under the Primitive Data Structures.
- These data types are also called **Simple data types**, as they contain characters that can't be divided further

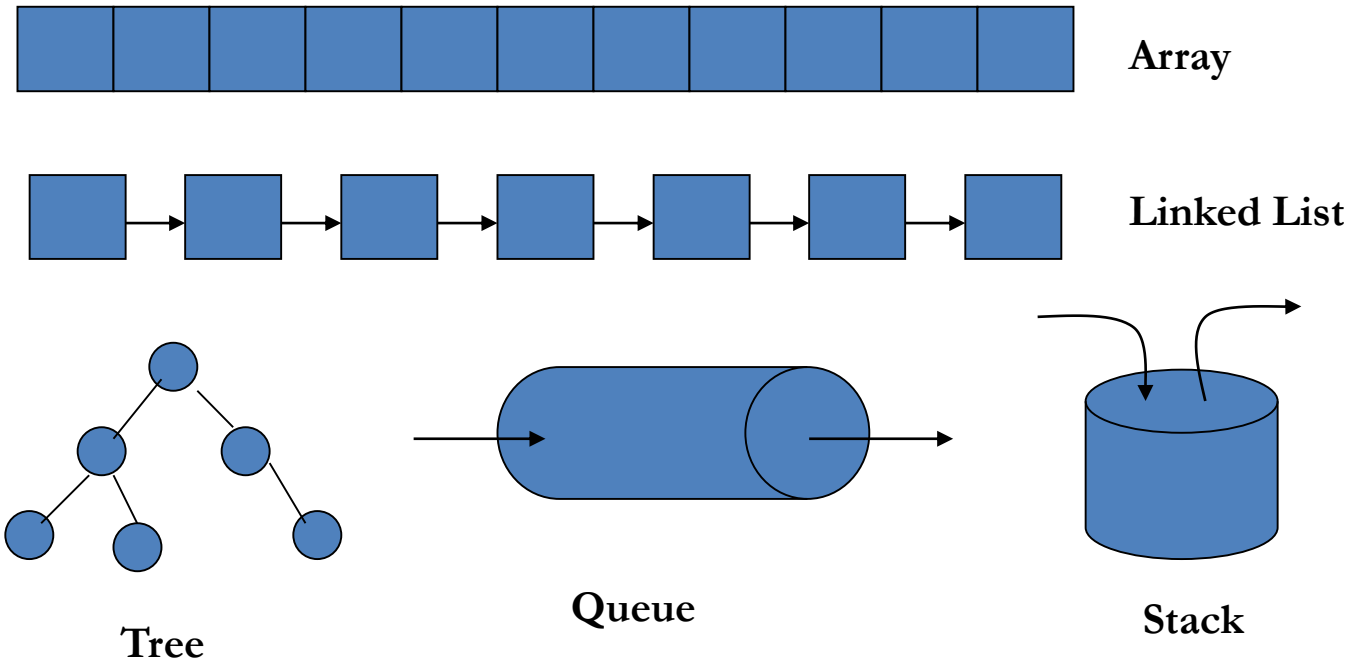
Non-Primitive Data Structures

- **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
- These data structures can't be manipulated or operated directly by machine-level instructions.
- The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).
- Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -
  - Linear Data Structures
  - Non-Linear Data Structures

Data Structure Classification

Characteristics	Description
Linear	Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure. The data items are assessed in a linear sequence, but it is not compulsory to store all elements sequentially. **Example: Array**
Non-Linear	Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only. **Example: Tree, Graph**
Homogeneous	All the elements are of same type. Example: Array
Heterogeneous	The elements are variety or dissimilar type of data Example: Structures
Static	Static data structure has a fixed memory size. It is easier to access the elements in a static data structure. *An example of this data structure is an array.*
Dynamic	In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code. *Examples of this data structure are queue, stack, etc.*

Types of Data Structures



There are many, but we named a few. We'll learn these data structures in great detail!

The Need for Data Structures

- Goal: to **organize data**

- Criteria: to facilitate **efficient**
  - **storage** of data
  - **retrieval** of data
  - **manipulation** of data

- Design Issue:
  - **select and design** appropriate data types
    (This is the main motivation to learn and understand data structures)

Data Structure Operations

- **Navigating**
  - Accessing each data element exactly once so that certain items in the data may be processed
- **Searching**
  - Finding the location of the data element (key) in the structure
- **Insertion**
  - Adding a new data element to the structure
- **Deletion**
  - Removing a data element from the structure
- **Sorting**
  - Arrange the data elements in a logical order (ascending/descending)
- **Merging**
  - Combining data elements from two or more data structures into one

Array

Array is a homogeneous, sequential collection of data items over a single variable name.

Characteristics of Array :

- Arrays are always stored in consecutive memory locations.
- An array can store multiple values of similar type which can be referred by a single name unlike a simple variable which store one value at a time.
- Array name is actually a pointer to the first location of the memory block allocated to the name of the array.
- An array either be a integer, character, or float data type can be initialized only during declaration time, and not afterwards.
- Any particular element of an array can be modified separately without disturbing other elements.
- All elements of an array share the same name, and they are distinguished from one another with the help of an element number.

Structure

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of **different kinds**.

Structures are used to **represent** a **record**. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

Defining the Structure:

```
struct [structure tag]
{

   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

Book Structure

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Access Structure Element:

```
/* Declare Book1 of type Book */
struct Books Book1;

/* Declare Book2 of type Book */
struct Books Book2;
Book1.title ="DSA";
Book2.book_id = 6495700;
```

Union

Unions are quite similar to the structures in C. Union is also a **derived type** as structure. Union can be defined in same manner as structures just the keyword used in defining union in **union** where keyword used in defining structure was **struct**.

You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining the Union:

```
union [union tag]
{

  member definition;
  member definition;
  ...
  member definition;
} [one or more union variables];
```

Data Union

```
union Data
{
  int i;
  float f;
  char str[20];
} data;
```

Access Union Elements

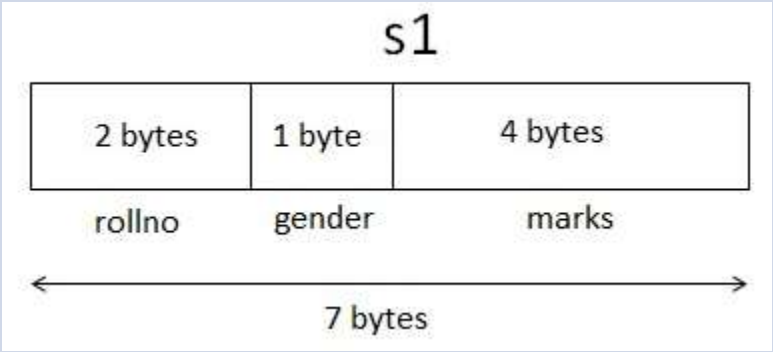
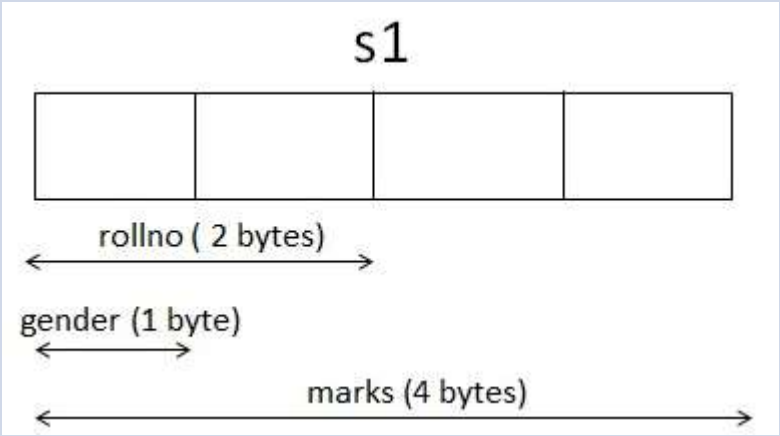
```
/* Declare data of type Data */
union Data data;

data.i =10;
data.f = 34.72;
data.str =“C Programming”
```

Difference b/w Structure & Union

Structure	Union
In structure each member get separate space in memory. Take below example. struct student { int rollno; char gender; float marks; } s1; The total memory required to store a structure variable is equal to the sum of size of all the members. In above case 7 bytes (2+1+4) will be required to store structure variable s1.	In union, the total memory space allocated is equal to the member with largest size. All other members share the same memory space. This is the biggest difference between structure and union. union student { int rollno; char gender; float marks; }s1; In above example variable marks is of float type and have largest size (4 bytes). So the total memory required to store union variable s1 is 4 bytes.

Difference b/w Structure & Union continue…

Pictorial Representation

Structure	Union
	

Difference b/w Structure & Union

Structure	Union

We can access any member in any sequence.
 	
s1.rollno = 20;
s1.marks = 90.0;
printf(“%d”,s1.rollno);
	We can access only that variable whose value is recently stored.
	
	s1.rollno = 20;
	s1.marks = 90.0;
	printf(“%d”,s1.rollno);
	
	The above code will show erroneous output. The value of rollno is lost as most recently we have stored value in marks. This is because all the members share same memory space.

All the members can be initialized while declaring the variable of structure.	Only first member can be initialized while declaring the variable of union. In above example, we can initialize only variable rollno at the time of declaration of variable.

Dynamic Memory Allocation

The exact size of array is **unknown** until the **compile time** i.e. time when a compiler compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes **insufficient** and sometimes **more than required**.

What?

The process of allocating memory during program execution is called dynamic memory allocation. It also allows a program to obtain more memory space, while running or to release space when no space is required.

Difference between static and dynamic memory allocatic

Sr #	Static Memory Allocation	Dynamic Memory Allocation
1	User requested memory will be allocated at compile time that sometimes insufficient and sometimes more than required.	Memory is allocated while executing the program.
2	Memory size can't be modified while execution.	Memory size can be modified while execution.

Dynamic Memory Allocation Functions

There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

Dynamic Memory Allocation Functions

1. malloc()

- The **"malloc"** or **"memory allocation"** method in C is used to dynamically allocate a single large block of memory with the specified size.
- It returns a pointer of type void which can be cast into a pointer of any form.
- It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.
- If space is insufficient, allocation fails and returns a NULL pointer.
- **Syntax of malloc() in C**

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

- **For Example:**

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

Dynamic Memory Allocation Functions

2. C calloc() method

- **“calloc”** or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type.
- it is very much similar to malloc() but has two different points and these are:
- It initializes each block with a default value ‘0’.
- It has two parameters or arguments as compare to malloc().

- **Syntax of calloc() in C**

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

  here, n is the no. of elements and element-size is the size of each element.

  Example:

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

This statement allocates contiguous space in memory for 25 elements each with the size of the float.

Dynamic Memory Allocation Functions

4. C realloc() method

- **“realloc”** or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory.
- In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.
- re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.
- If space is insufficient, allocation fails and returns a NULL pointer.

- **Syntax of realloc() in C**

  ptr = realloc(ptr, newSize);

  where ptr is reallocated with new size 'newSize'.

Dynamic Memory Allocation Functions

3. C free() method

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

Syntax of free() in C:

```
free(ptr);
```

Example : DMA

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *pi;
  float *pj;
  pi = (int *) malloc(sizeof(int));
  pj = (float *) malloc(sizeof(float));
  *pi = 10;
   *pj = 3.56;
  printf("integer = %d, float = %f", *pi, *pj);
  free(pi);
  free(pj);
  return 0;
}
```

IMH

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *pi;
  float  *pj;
  pi = (int *) malloc(sizeof(int));
  pj = (float *) malloc(sizeof(float));
  if (!pi || !pj)
  {
     printf("Insufficient Memory");
     return;
  }
  *pi = 10;
   *pj = 3.56;
    printf("integer = %d, float = %f", *pi, *pj);
    free(pi);
    free(pj);
   return 0;
}
```

IMH : Insufficient Memory Handling

DMA- Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int* ptr;
int n, i;
printf("Enter number of elements:");
scanf("%d",&n);
printf("Entered number of elements: %d\n",
n);
// Dynamically allocate memory using
malloc()
ptr = (int*)malloc(n * sizeof(int));
// Check if the memory has been successfully
if (ptr == NULL) {
printf("Memory not allocated.\n");
                exit(0);
                }
```

IMH

```
else
{
// Memory has been successfully allocated
printf("Memory successfully allocated using
malloc.\n");
// Get the elements of the array
for (i = 0; i < n; ++i)
{
                ptr[i] = i + 1;
}
// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i)
{
   printf("%d, ", ptr[i]);

}
}
return 0;
}
```

IMH : Insufficient Memory Handling

DMA realloc Example

C Program illustrating the usage of realloc

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  char *mem_allocation;
   /* memory is allocated dynamically */
   mem_allocation = malloc( 20 * sizeof(char) );
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n"); return;
   }
   else
   {
     strcpy( mem_allocation,“dynamic memory allocation for realloc
      function");
   }
   printf("Dynamically allocated memory content  : %s\n",
      mem_allocation );
   mem_allocation=realloc(mem_allocation,100*sizeof
      (char));
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested
      memory\n");
   }
   else
   {
      strcpy( mem_allocation,"space is extended upto
      100 characters");
   }
   printf("Resized memory : %s\n", mem_allocation );
   free(mem_allocation);
   return 0;
}
```

Difference between calloc and malloc

Sr #	malloc	calloc
1	It allocates only single block of requested memory	It allocates multiple blocks of requested memory
2	doesn't initializes the allocated memory. It contains garbage values.	initializes the allocated memory to zero
3	int *ptr; ptr = malloc(20 * sizeof(int)); For the above, 20*2 bytes of memory only allocated in one block. Total = 40 bytes	int *ptr; Ptr = calloc(20, 20 * sizeof(int)); For the above, 20 blocks of memory will be created and each contains 20*2 bytes of memory. Total = 800 bytes

Algorithm Specification

An algorithm is a **finite** set of instructions that, if followed, **accomplishes a particular task**. In addition, all algorithms must satisfy the following criteria:

1. **Input**. There are zero or more quantities that are externally supplied.
2. **Output**. At least one quantity is produced.
3. **Definiteness**. Each instruction is clear and unambiguous.
4. **Finiteness**. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness**. Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

Difference between an algorithm & program – program does not have to satisfy the 4th condition.

Describing Algorithm

1. **Natural Language –** e.g. English, Chinese - Instructions must be definite and effectiveness.
2. **Graphics representation – e.g. Flowchart -** work well only if the algorithm is small and simple.
3. **Pseudo Language -**
   - Readable
   - Instructions must be definite and effectiveness
4. **Combining English and C**

Why do we need Algorithms?

We need algorithms because of the following reasons:

- **Scalability:** It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.
- **Performance:** The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

Example- Real world Problem

Suppose we want to make a lemon juice, so following are the steps required to make a lemon juice:

- Step 1: First, we will cut the lemon into half.
- Step 2: Squeeze the lemon as much you can and take out its juice in a container.
- Step 3: Add two tablespoon sugar in it.
- Step 4: Stir the container until the sugar gets dissolved.
- Step 5: When sugar gets dissolved, add some water and ice in it.
- Step 6: Store the juice in a fridge for 5 to minutes.
- Step 7: Now, it's ready to drink.

The above real-world can be directly compared to the definition of the algorithm. We cannot perform the step 3 before the step 2, we need to follow the specific order to make lemon juice. An algorithm also says that each and every instruction should be followed in a specific order to perform a specific task.

Example- Add two numbers entered by the user

Suppose we want to make a lemon juice, so following are the steps required to make a lemon juice:

- Step 1: Start
- Step 2: Declare three variables a, b, and sum.
- Step 3: Enter the values of a and b.
- Step 4: Add the values of a and b and store the result in the sum variable, i.e., sum=a+b.
- Step 5: Print sum
- Step 6: Stop

A simple algorithm

- Problem: Find maximum of a, b, c
- Algorithm
  - Input = a, b, c
  - Output = max
  - Process
    - Let max = a
    - If b > max then
      - max = b
    - If c > max then
      - max = c
    - Display max

• Order is very important!!!

Algorithm Specification cont…

Describing Algorithm – Natural Language

Problem - Design an algorithm to add two numbers and display result.

Step 1 − START
Step 2 − declare three integers a, b & c
Step 3 − define values of a & b
Step 4 − add values of a & b
Step 5 − store output of step 4 to c
Step 6 − print c
Step 7 − STOP

Note *- Writing step numbers, is optional.*

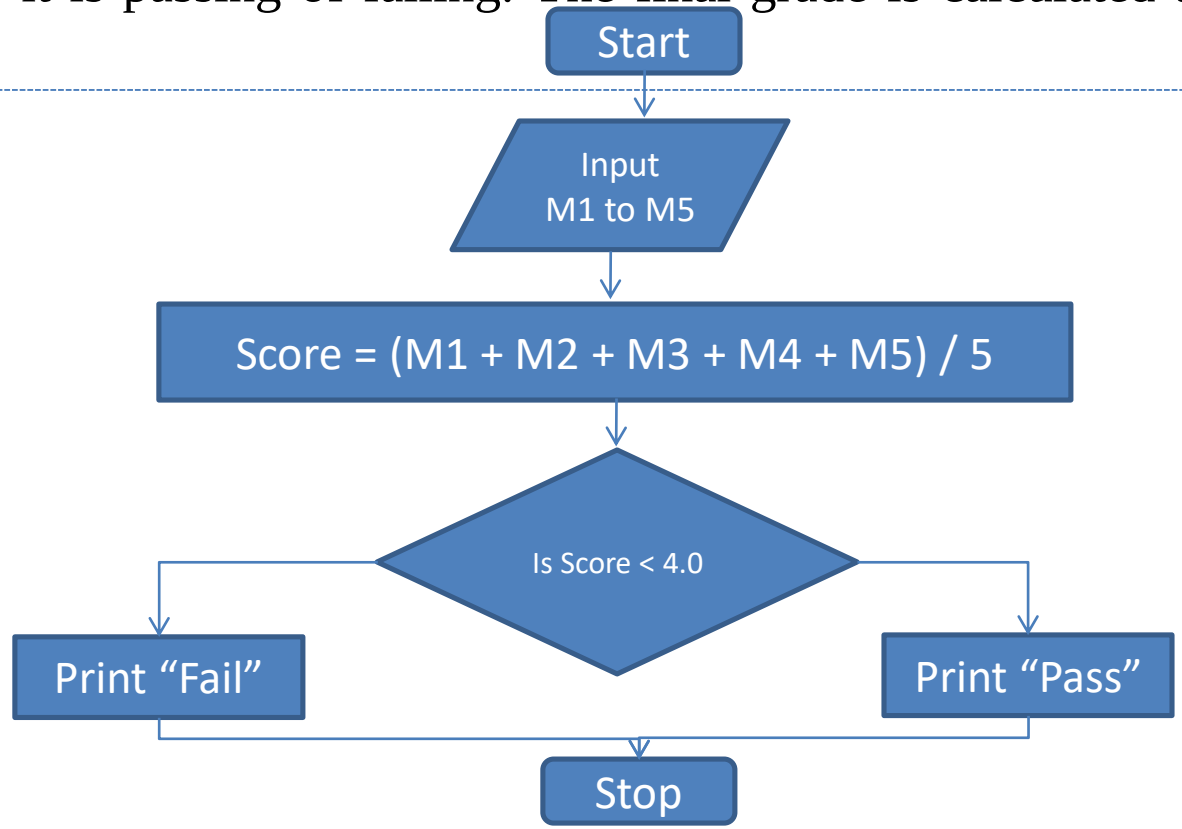
Problem - Design an algorithm to find the largest data value of a set of given positive data values.

Step 1 − START
Step 2 − input NUM
Step 3 – LARGE = NUM
Step 4 − While (NUM >=0)
    if (NUM > LARGE) then
        LARGE = NUM
    input NUM
Step 5 – display “Largest Value is:”, LARGE
Step 6 − STOP

Algorithm Specification cont…

Describing Algorithm – Flowchart

Problem – Write an algorithm to determine a student's final grade and indicate whether it is passing of failing. The final grade is calculated as the average of five marks.



How to express an algorithm?

- A sequence of steps to solve a problem
- We need a way to express this sequence of steps

1. **Natural language (NL) is an obvious choice, but not a good choice. Why?**
   - NLs are notoriously ambiguous (unclear)
2. **Programming language (PL) is another choice, but again not a good choice. Why?**
   - Algorithm should be PL independent

- **We need some balance**
  - We need PL independence
  - We need clarity
  - Pseudo-code provides the right balance

What is Pseudo-code?

- Pseudo-code is a short hand way of describing a computer program
- Rather than using the specific syntax of a computer language, more general wording is used
- It is a mixture of NL and PL expressions, in a systematic way
- Using pseudo-code, it is easier for a non-programmer to understand the general workings of the program

Pseudo-code: General Guidelines

- Use PLs construct that are consistent with modern high level languages, e.g. C++, Java, ...
- Use appropriate comments for clarity
- Be simple and precise

Components of Pseudo-code

- **Expressions**
  - Standard mathematical symbols are used
    - Left arrow sign (←) as the assignment operator in assignment statements (equivalent to the = operator in Java)
    - Equal sign (=) as the equality relation in Boolean expressions (equivalent to the "= =" relation in Java)
    - For example

      Sum ← 0

      Sum ← Sum + 5

      What is the final value of sum?

Components of Pseudo-code (cont.)

- **Decision structures (if-then-else logic)**

  - **if** condition **then**
    true-actions
    [**else**
    false-actions]
  - We use indentation to indicate what actions should be included in the true-actions and false-actions
  - For example

```
if marks > 50 then
  print "Congratulation, you are passed!"
else
  print "Sorry, you are failed!"
  end if
```

What will be the output if marks are equal to 75?

Components of Pseudo-code (cont.)

- Loops (Repetition)
  - **Pre-condition loops**
    - **While loops**
      - **while** condition **do** actions
      - We use indentation to indicate what actions should be included in the loop actions
      - For example

```
while counter < 5 do
        print "Welcome to CS204!"
        counter ← counter + 1
end while
```

What will be the output if counter is initialised to 0, 7?

Components of Pseudo-code (cont.)

- Loops (Repetition)
  - **Pre-condition loops**
    - For loops
      - **for** variable-increment-definition **do** actions
      - For example

```
for counter ← 0; counter < 5; counter ← counter + 2 do
        print "Welcome to CS204!"
end for
```

What will be the output?

Components of Pseudo-code (cont.)

- Loops (Repetition)
  - **Post-condition loops**
    - Do loops
      - **do** actions **while** condition
      - For example

```
do
        print "Welcome to CS204!"
        counter ← counter + 1
while counter < 5
```

What will be the output, if counter was initialised to 10?

The body of a post-condition loop must execute at least once

Components of Pseudo-code (cont.)

- **Method declarations**
  - Return_type method_name (parameter_list) method_body
  - For example
    - **integer *sum* (integer num1, integer num2)**
    - **start**
    - **result ← num1 + num2**
    - **end**
- **Method calls**
  - object.method (args)
  - For example

    mycalculator.sum(num1, num2)

Components of Pseudo-code (cont.)

Method returns

- **return** value
- For example

```
integer sum ( integer num1, integer num2)
start
                    result ← num1 + num2
                    return result
end
```

Pseudo Language Example

1. Problem - Design the pseudo code to add two numbers and display the average.

INPUT: x, y
sum ← x + y
average ← sum / 2
OUTPUT: 'Average is:' average

2. Problem - Design the pseudo code to calculate & display the area of a circle

INPUT: radius
area← 3.14 * radius * radius
OUTPUT: 'Area of the circle is ' area

2. Problem - Design the pseudo code to calculate & display the largest among 2 numbers

INPUT: num1, num2
max ← num1
IF (num2 > num 1) THEN
  max ← num2
ENDIF
OUTPUT: 'Largest among 2 numbers is' max

Recursive Algorithm

A recursive algorithm is an algorithm which calls itself. In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem. There are 2 types of recursive functions.

Direct recursion Function	Indirect recursion Function
Functions call themselves e.g. function α calls α	Functions call other functions that invoke the calling function again e.g. a function α calls a function β that in turn calls the original function α.

Example –
int func (int n)
{
if (n <= 1)
return 1;
else
return (func(n-1)+func(n-2));
}
	Example –
	int func1(int n)
	{
	if (n<=1)
	return 1;
	else
	return func2(n);
	}
	
	int func2(int n)
	{
	return func1(n);
	}

Recursive Algorithm cont…

- When is **recursion an appropriate mechanism**?
  - ✓ The problem itself is defined recursively
  - ✓ Statements: if-else and while can be written recursively
  - ✓ Art of programming
- **Why recursive algorithms** ?
  - ✓ Powerful, express an complex process very clearly
- **Properties** - A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −
  - ✓ **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
  - ✓ **Progressive criteria**− The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.
- **Implementation** - Many programming languages implement recursion by means of **stack.**

Recursive Implementation of Fibonacci


```
#include<stdio.h>
void Fibonacci(int);
int main()
{
   int k,n;
   long int i=0,j=1,f;

   printf("Enter the range of the Fibonacci series: ");
   scanf("%d",&n);

   printf("Fibonacci Series: ");
   printf("%d %d ",0,1);
   Fibonacci(n);
   return 0;
}
```

```
//continuation of program
void Fibonacci(int n)
{
   static long int first=0,second=1,sum;

   if(n>0)
   {
      sum = first + second;
      first = second;
      second = sum;
      printf("%ld ",sum);
      Fibonacci(n-1);
   }
}
```

Base Criteria

Progressive Criteria

Algorithm Analysis

- **Time complexity:** The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution.

- **Space complexity:** An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.

What is a Good Algorithm?

- Efficient:
  - Running time (small)
  - Space used (less)

- Efficiency as a function of input size:
  - The input size
  - Number of data elements

Space Complexity of Algorithms

For any algorithm memory may be used for the following:

- Variables (includes constant values, temporary values)
- Program Instruction
- Execution

- Space complexity: Amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the output.
- While executing, algorithm uses memory space for three reasons:
  - Instruction Space: memory used to save the compiled version of instructions.
  - Environmental Stack: In function call, a system stack is maintained.
  - Data Space: Amount of space used by the variables and constants.

- while calculating the Space Complexity of any algorithm,
  - only Data Space is considered (neglecting Instruction Space and Environmental Stack.)

Calculating the Space Complexity

Example:

```
{
    int z = a + b + c;
    return(z);
}
```

In this expression,

- variables a, b, c and z are all integer types, take 4 bytes each.
- Total memory = (4(4) + 4) = 20 bytes,
- Additional 4 bytes is for return value.
- Because this space requirement is fixed, called Constant Space Complexity.

Example:

```
// n is the length of array a[]
int sum(int a[], int n) {
    int x = 0;                          // 4 bytes for x
    for(int i = 0; i < n; i++)  {       // 4 bytes for i
        x  = x + a[i];
    }
    return(x);
}
```

- In this code, 4*n bytes of space is required for the array a[] elements.
- 4 bytes each for x, n, i and the return value.
- Total memory requirement is (4n + 16), increasing linearly with increase in the input value n. Called as Linear Space Complexity.

Time Complexity

- Time complexity of an algorithm signifies the total time required by the program to run till its completion.
- Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution.
- Any problem can have number of solutions.
- Two different algorithms to find square of a number:

Asymptotic Notation

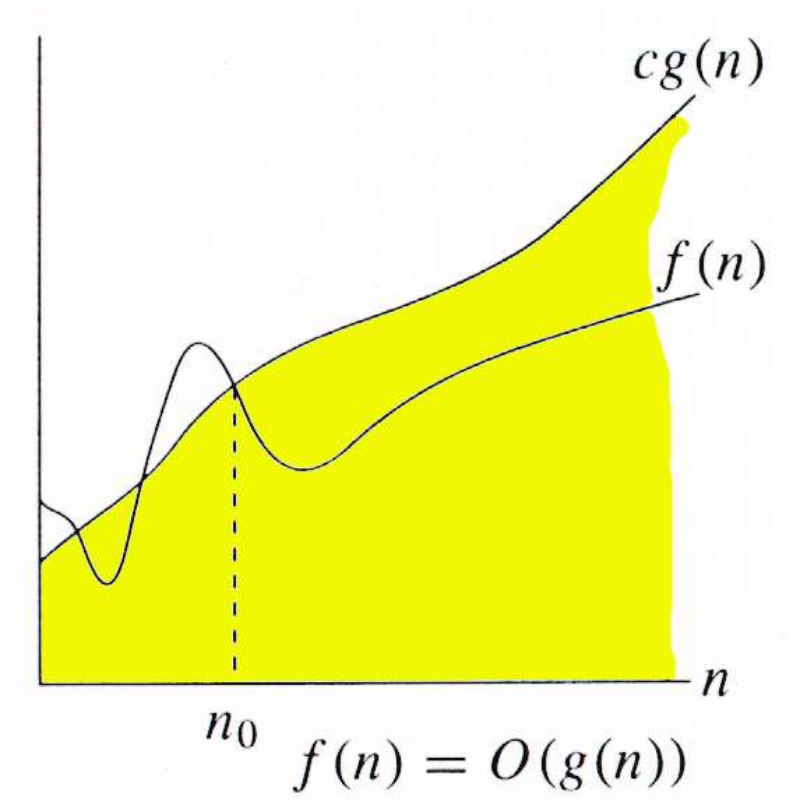
- Commonly used asymptotic notations to calculate the running time complexity of an algorithm.
  - O Notation
  - Ω Notation
  - θ Notation

Order of Time Complexity :

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

Big Oh Notation, O

- The Big Oh notation is the formal way to express the upper bound of an algorithm's running time.
- It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.
- Definition: $O(g(n)) = \{ f(n) :$ $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,
- we have $0 \leq f(n) \leq cg(n) \}$
- Intuitively: Set of all functions whose rate of growth is the same as or lower than that of g(n).



Big Oh Notation: Example

Example:

Compute Big-Oh notation for

$f(n) = 2n+3$

Given $f(n)= 2n+3$

$f(n) \leq c * g(n)$

$2n +3 \leq 10n, \forall\ n \geq 1,\ f(n)= O(n)$

Or

$2n +3 \leq 2n +3n , \forall\ n \geq 1,\ f(n)= O(n)$

Or

$2n+3 \leq 2\ n^2 +3\ n^2\ , \forall\ n \geq 1,\ f(n)= O(n^2)$

This function $f(n)=2n+3$ satisfies $O(n)$ means it will also satisfy $O(n^2)$

So

$f(n)= O(n)$

$f(n)= O(n^2)$

$f(n)\ \ O(2^n)$

But it will false $f(n)= O(\text{Log } n)$, Because it is lower bound.

When you write the O . Write the closest function i.e. $O(n)$.

Big Oh Notation: Example

Example:

Compute Big-Oh notation for

$f(n) = 10n^2+4n+2$

Given $f(n)= 10n^2+4n+2$

$f(n) \leq c * g(n)$

$10n^2+4n+2 \leq 10n^2+4n+n$, for $n \geq 2$

$10n^2+4n+2 \leq 10n^2+5n$

$10n^2+4n+2 \leq 10n^2+n^2$, for $n \geq 5$

$10n^2+4n+2 \leq 11n^2$ where c= 11,

$g(n) = n^2$ and $n_0=5$

Hence $f(n) = O(n^2)$

Example:

Compute Big-Oh notation for

$f(n) = 1000n^2+100n- 6$

Given $f(n) = 1000n^2+100n-6$

$f(n) \leq c * g(n)$

$1000n^2+100n-6 \leq 1000\ n^2+100n$ for all values of n

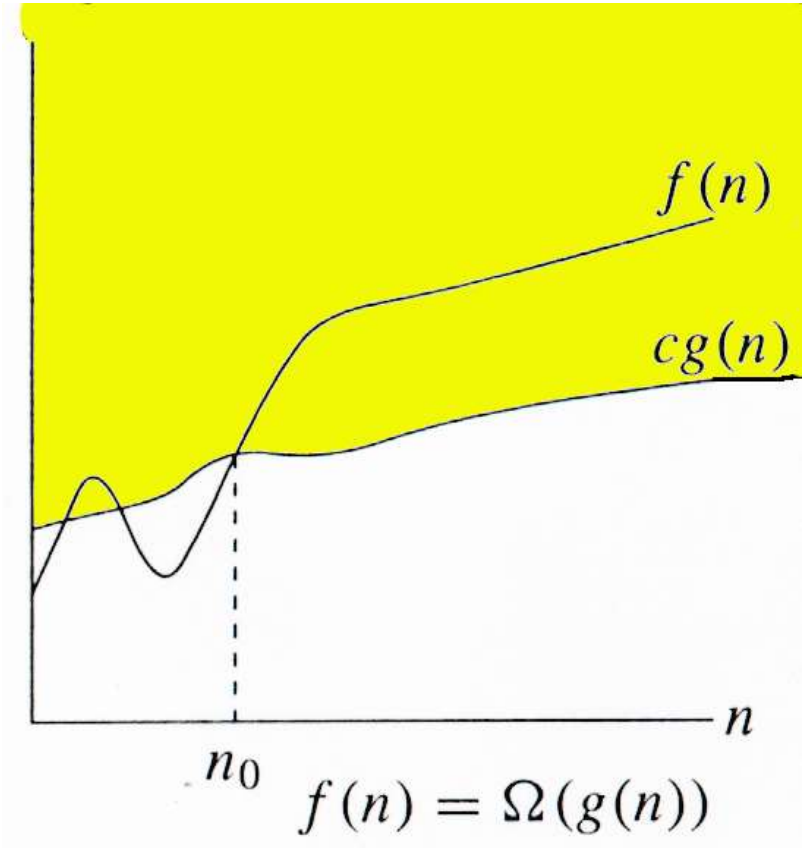
$1000n^2+100n-6 \leq 1000\ n^2+n^2$, for $n \geq 100$

$1000n^2+100n-6 \leq 1001\ n^2$, where c =1001, $g(n) = n^2$ and $n_0=100$

Hence $f(n)=O(n^2)$

Omega Notation, Ω

- The Omega notation is the formal way to express the lower bound of an algorithm's running time.
- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.
- Definition: $\Omega(g(n)) = \{ f(n) :$ $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,
- we have $0 \leq cg(n) \leq f(n)\}$
- Intuitively: Set of all functions whose rate of growth is the same as or higher than that of g(n).



Omega Notation, Ω

Example:

Compute omega notation for

f(n) = 2n+3

Given f(n)= 2n+3

f(n) ≥ c * g(n)

2n +3 ≥ 1*n, ∀ n ≥1,

Hence f(n) = Ω(n) Its use full because its closest

Also

2n +3 ≥ 1*log n, n ≥1

Hence f(n) = Ω(logn)

This function f(n)=2n+3 satisfies O(n) means it will also satisfy $O(n^2)$

So

f(n) = Ω(n) We can choose this one.

f(n) = Ω(√n)

f(n) = Ω(log n)

f(n) = Ω(1)

But it will false f(n)= Ω (n^2) , Because it is upper bound.

Omega Notation, Ω

Example:

Compute omega notation for f(n)= $10n^2+4n+2$

Given f(n)= $10n^2+4n+2$

$f(n) \geq c * g(n)$

$10n^2+4n+2 \geq 10n^2$ for all values of n ($n \geq 0$)

where c=10, $g(n)=n^2$ and $n_0=0$

Hence $f(n) = \Omega(n^2)$

Example:

Compute omega notation for f(n)= $4n^3+2n+3$

Given f(n)= $4n^3+2n+3$

$f(n) \geq c * g(n)$

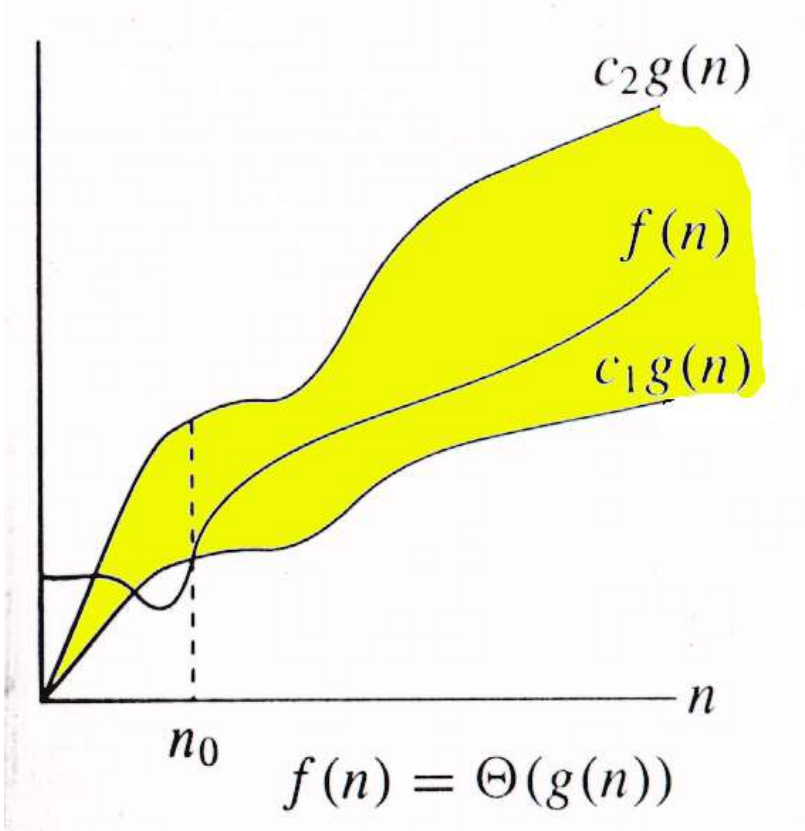
$4n^3+2n+3 \geq 4n^3$ for all values of n ($n \geq 0$)

where c=4, $g(n)=n^3$

and $n_0=0$

Theta Notation, θ

- The Theta notation is the formal way to express both the lower bound and the upper bound of an algorithm's running time.
- Definition: $\Theta(g(n)) = \{f(n) :$ $\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$
- *Intuitively*: Set of all functions that have the same *rate of growth* as $g(n)$.
- g(n) is an asymptotically tight bound for f(n).



Theta Notation, θ

Example:

Compute theta notation for f(n)=3n+2

Given f(n)=3n+2

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

$1*n \leq 2n+3 \leq 5*n$

Here c1=1, g(n)=n, c2= 5

So f(n)= ɵ(n)

Example:

Compute theta notation for f(n)=10n2+4n+2

Given $f(n)=10n^2+4n+2$

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

Compute $f(n) \leq c_2 * g(n)$

$10n^2+4n+2 \leq 10n^2+4n+n$, for $n \geq 2$

$10n^2+4n+2 \leq 10n^2+5n$

$10n^2+4n+2 \leq 10n^2+n^2$, for $n \geq 5$

$10n^2+4n+2 \leq 11n^2$, where $c_2=11$ and $g(n)=n^2$

Compute $c_1 * g(n) \leq f(n)$

$10n^2 \leq 10n^2+4n+2$ for all values of n

where c1=10, $g(n)=n^2$

Hence, $f(n)= ɵ(n^2)$

Common Asymptotic Notations

Following is a list of some common asymptotic notations:

- constant – $O(1)$
- logarithmic – $O(\log n)$
- linear – $O(n)$
- n log n – $O(n \log n)$
- quadratic – $O(n^2)$
- cubic – $O(n^3)$
- polynomial – $n^{O(1)}$
- exponential – $2^{O(n)}$

Analysis of Algorithms

1. O(1): Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

```
// set of non-recursive and non-loop statements
```

- Example: swap() function has O(1) time complexity.
- A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
    // some O(1) expressions   }
```

Analysis of Algorithms

2) O(n): Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount.

Example:

```
// Here c is a positive integer constant
   for (int i = 1; i <= n; i += c) {
        // some O(1) expressions
   }
   for (int i = n; i > 0; i -= c) {
        // some O(1) expressions    }
```

Analysis of Algorithms

3. $O(n^c)$: Time complexity of nested loops is equal to the number of times the innermost statement is executed.

Example:

```
for (int i = 1; i <=n; i += c) {
      for (int j = 1; j <=n; j += c) {
         // some O(1) expressions   }
  }

 for (int i = n; i > 0; i -= c) {
      for (int j = i+1; j <=n; j += c) {
         // some O(1) expressions
  }
```

For example Selection sort and Insertion Sort have $O(n^2)$ time complexity.

Analysis of Algorithms

4) O(Logn): Time Complexity of a loop is considered as O(log n) if the loop variables is divided / multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c) {
        // some O(1) expressions     }
for (int i = n; i > 0; i /= c) {
        // some O(1) expressions     }
```

Example: Binary Search (refer iterative implementation) has O(log n) time complexity.

Analysis of Algorithms

5. O(LogLogn) Time Complexity of a loop is considered as O(log log n) if the loop variables is reduced / increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
}
//Here func() is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = func(i)) {
    // some O(1) expressions
}
```

Analysis of Algorithms

How to combine time complexities of consecutive loops?

When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <=m; i += c) {
    // some O(1) expressions
}
for (int i = 1; i <=n; i += c) {
    // some O(1) expressions
}
```

- Time complexity of above code is O(m) + O(n) which is O(m+n)
- If m == n, the time complexity becomes O(2n) which is O(n).

Question Discussion

What is time complexity of func()?

```
int func(int n)  {
  int count = 0;
  for (int i = n; i > 0; i /= 2)
     for (int j = 0; j < i; j++)
        count += 1;
  return count; }
```

(A) $O(n^2)$
(B) O(nLogn)
(C) O(n)
(D) O(nLognLogn)

Question Discussion

What is time complexity of func()?

```
int func(int n)  {
  int count = 0;
  for (int i = n; i > 0; i /= 2)
     for (int j = 0; j < i; j++)
        count += 1;
  return count; }
```

(A) $O(n^2)$
(B) O(nLogn)
(C) O(n)
(D) O(nLognLogn)

Answer: (C)

Explanation:

- For a input integer n, the innermost statement of func() is executed following times.

  $n + n/2 + n/4 + \ldots 1$

- So time complexity T(n) can be written as: $T(n) = O(n + n/2 + n/4 + \ldots 1) = O(n)$

Notation-Links

Notation:
https://www.youtube.com/watch?v=A03oI0znAoc
https://www.youtube.com/watch?v=Nd0XDY-jVHs
https://www.youtube.com/watch?v=NI4OKSvGAgM

Complexity :
https://www.youtube.com/watch?v=9TlHvipP5yA
https://www.youtube.com/watch?v=1U3Uwct45IY
https://www.youtube.com/watch?v=p1EnSvS3urU

THANK YOU!