CPE112

Programming with Data Structures

การโปรแกรมด้วยโครงสร้างข้อมูล

Assoc. Prof. Dr. Natasha Dejdumrong

COURSE MODULES

- ★ MODULE CPE11201 - Linear Data Structures and Algorithms
- ★ MODULE CPE11202 - Non-Linear Data Structures and Algorithms
- ★ MODULE CPE11203 - Algorithms for Data Structures and Applications

MODULE 1 [CPE11201]

Linear Data Structures and Algorithms

โครงสร้างข้อมูลแบบเชิงเส้นและอัลกอริทึม

Assoc. Prof. Dr. Natasha Dejdumrong

OVERVIEW

- ★ Introduction to Data Structures
- ★ Classification of Data Structures
- ★ Operations on Data Structures
- ★ Abstract Data Type
- ★ Algorithm
- ★ Different Approaches to Designing an Algorithm
- ★ Control Structures used in Algorithms
- ★ Time and Space Complexity
- ★ Big-O Notation

Basic Terminology

- ★ Our aim has been to design good programs, where a good program is defined as a program that
  - ★ runs correctly
  - ★ is easy to read and understand
  - ★ is easy to debug and
  - ★ is easy to modify.
- ★ A program should undoubtedly give correct results, but along with that it should also run efficiently.

Basic Terminology

- ★ A program is said to be efficient when it executes in minimum time and with minimum memory space.
- ★ In order to write efficient programs we need to apply certain data management concepts.
- ★ The concept of data management is a complex task that includes activities like data collection, organization of data into appropriate structures, and developing and maintaining routines for quality assurance.

Data Structure

- ★ A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.
- ★ Data structures are used in almost every program or software system.
- ★ Some common examples of data structures are
  arrays, linked lists,
  queues, stacks,
  binary trees, and hash tables.

Data Structure

- Data structures are widely applied in the following areas:
  - Compiler design
  - Statistical analysis package
  - Numerical analysis
  - Artificial intelligence
  - Operating system
  - DBMS
  - Simulation
  - Graphics.

Data Structure

★ When you will study DBMS as a subject, you will realize that the major data structures used in

- ★ the Network data model is graphs,
- ★ Hierarchical data model is trees, and
- ★ RDBMS is arrays.

★ Specific data structures are essential ingredients of many efficient algorithms as they enable the programmers to manage huge amounts of data easily and efficiently.

★ Some formal design methods and programming languages emphasize data structures and the algorithms as the key organizing factor in software design.

Data Structure

★ Some formal design methods and programming languages emphasize data structures and the algorithms as the key organizing factor in software design.

★ This is because representing information is fundamental to computer science.

★ The primary goal of a program or software is not to perform calculations or operations but to store and retrieve information as fast as possible.

Data Structure

- ★ If a program meets its performance goals with a data structure that is simple to use, then it makes no sense to apply another complex data structure just to exhibit the programmer's skill.
- ★ When selecting a data structure to solve a problem, the following steps must be performed.
  - ★ 1. Analysis of the problem to determine the basic operations that must be supported. For example, basic operation may include inserting/deleting/searching a data item from the data structure.
  - ★ 2. Quantify the resource constraints for each operation.
  - ★ 3. Select the data structure that best meets these requirements.

Elementary Data Structure Organization

★ Data structures are building blocks of a program.

★ A program built using improper data structures may not work as expected.

★ As a programmer it is mandatory to choose most appropriate data structures for a program.

Elementary Data Structure Organization

★ The term data means a value or set of values. It specifies either the value of a variable or a constant
(e.g., marks of students, name of an employee, address of a customer, value of pi, etc.).

★ While a data item that does not have subordinate data items is categorized as an elementary item, the one that is composed of one or more subordinate data items is called a group item.
For example, a student's name may be divided into three sub-items–first name, middle name, and last name – but his roll number would normally be treated as a single item.

Elementary Data Structure Organization

★ A record is a collection of data items.

For example, the name, address, course, and marks obtained are individual data items. But all these data items can be grouped together to form a record.

★ A file is a collection of related records.

For example, if there are 60 students in a class, then there are 60 records of the students. All these related records are stored in a file.

★ Similarly, we can have a file of all the employees working in an organization, a file of all the customers of a company, a file of all the suppliers, so on and so forth.

Classification of Data Structure

★ Data structures are generally categorized into two classes: primitive and non-primitive data structures.

★ Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms "data type", "basic data type", and "primitive data type" are often used interchangeably.

★ Non-primitive data structures are those data structures which are created using primitive data structures.
Examples of such data structures include linked lists, stacks, trees, and graphs.

Non-Primitive Data Structures

- ★ Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.
- ★ If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure.
  Examples include arrays, linked lists, stacks, and queues.
- ★ Linear data structures can be represented in memory in two different ways.
  - ★ One way is to have to a linear relationship between elements by means of sequential memory locations.
  - ★ The other way is to have a linear relationship between elements by means of links.

Non-Linear Data Structures

★ Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.

★ However, if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure.

★ The relationship of adjacency is not maintained between elements of a non-linear data structure.

Examples include trees and graphs.

Array

- ★ An array is a collection of similar data elements.
- ★ These data elements have the same data type.
- ★ The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).

In C, arrays are declared using the following syntax:

```
type name[size];
```

For example,

```
int marks[10];
```

1st element	2nd element	3rd element	4th element	5th element	6th element	7th element	8th element	9th element	10th element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

Figure 2.1 Memory representation of an array of 10 elements

Array

- ★ Arrays are generally used when we want to store large amount of similar type of data. But they have the following limitations:
  - ★ Arrays are of fixed size.
  - ★ Data elements are stored in contiguous memory locations which may not be always available.
  - ★ Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

Linked List

- ★ A linked list is a very flexible, dynamic data structure in which elements (called nodes) form a sequential list.
- ★ In contrast to static arrays, a programmer need not worry about how many elements will be stored in the linked list. This feature enables the programmers to write robust programs which require less maintenance. In a linked list, each node is allocated space as it is added to the list.
- ★ Every node in the list points to the next node in the list. Therefore, in a linked list, every node contains the following two types of data:
  - ★ The value of the node or any other data that corresponds to that node
  - ★ A pointer or link to the next node in the list.

Linked List

★ The last node in the list contains a NULL pointer to indicate that it is the end or tail of the list.

★ Since the memory for a node is dynamically allocated when it is added to the list, the total number of nodes that may be added to a list is limited only by the amount of memory available.

Figure 2.2 Simple linked list

Note

Advantage: Easier to insert or delete data elements
Disadvantage: Slow search operation and requires more memory space

Stack

★ A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack.

★ Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.

★ In the computer's memory, stacks can be implemented using arrays or linked lists.

A	AB	ABC	ABCD	ABCDE					
0	1	2	3	top = 4	5	6	7	8	9

Figure 2.3 Array representation of a stack

Stack

- ★ Every stack has a variable top associated with it. top is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted.
- ★ There is another variable MAX, which is used to store the maximum number of elements that the stack can store. If top = NULL, then it indicates that the stack is empty and if top = MAX–1, then the stack is full.

A	AB	ABC	ABCD	ABCDE					
0	1	2	3	top = 4	5	6	7	8	9

Figure 2.3 Array representation of a stack

Stack

- ★ Stack supports three basic operations: push, pop, and peep.
- ★ The push operation adds an element to the top of the stack.
- ★ The pop operation removes the element from the top of the stack.
- ★ The peep operation returns the value of the topmost element of the stack (without deleting it).

However, before inserting an element in the stack, we must check for overflow conditions. An overflow occurs when we try to insert an element into a stack that is already full.

Similarly, before deleting an element from the stack, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a stack that is already empty.

Queue

- ★ A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out.
- ★ The elements in a queue are added at one end called the rear and removed from the other end called the front.
- ★ Like stacks, queues can be implemented by using either arrays or linked lists.
- ★ Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.

Queue

	Front					Rear				
Value	12	9	7	18	14	36				
Index	0	1	2	3	4	5	6	7	8	9

Figure 2.4 Array representation of a queue

Here, `front = 0` and `rear = 5`. If we want to add one more value to the list, say, if we want to add another element with the value 45, then the `rear` would be incremented by 1 and the value would be stored at the position pointed by the `rear`. The queue, after the addition, would be as shown in Fig. 2.5.

Here, `front = 0` and `rear = 6`. Every time a new element is to be added, we will repeat the same procedure.

	Front						Rear			
Value	12	9	7	18	14	36	45			
Index	0	1	2	3	4	5	6	7	8	9

Figure 2.5 Queue after insertion of a new element

Now, if we want to delete an element from the queue, then the value of `front` will be incremented. Deletions are done only from this end of the queue. The queue after the deletion will be as shown in Fig. 2.6.

		Front					Rear			
Value		9	7	18	14	36	45			
Index	0	1	2	3	4	5	6	7	8	9

Figure 2.6 Queue after deletion of an element

Tree

★ A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order. One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.

★ The simplest form of a tree is a binary tree.

★ A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees. Each node contains a data element, a left pointer which points to the left sub-tree, and a right pointer which points to the right sub-tree.

Tree

★ A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees.

★ The root element is the topmost node which is pointed by a 'root' pointer. If root = NULL then the tree is empty.

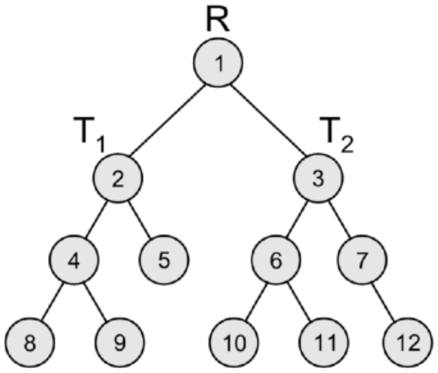

Figure 2.7 Binary tree

Note *Advantage*: Provides quick search, insert, and delete operations
Disadvantage: Complicated deletion algorithm

Graph

★ A graph is a non-linear data structure which is a collection of vertices (also called nodes) and edges that connect these vertices.

★ A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist.

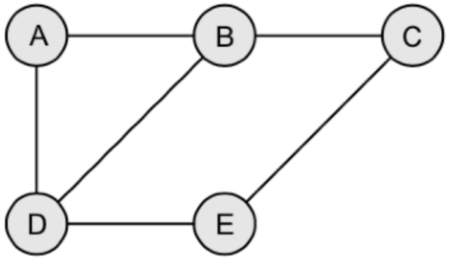

Figure 2.8 Graph

Graph

★ In a tree structure, nodes can have any number of children but only one parent, a graph on the other hand relaxes all such kinds of restrictions.

★ A node in the graph may represent a city and the edges connecting the nodes can represent roads. A graph can also be used to represent a computer network where the nodes are workstations and the edges are the network connections.

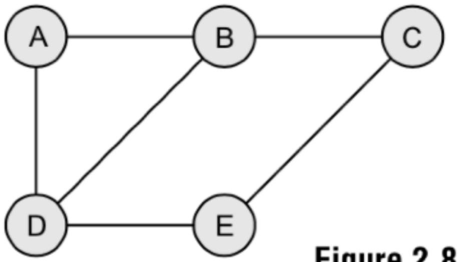

Figure 2.8 Graph

Graph

- ★ Note that unlike trees, graphs do not have any root node.
- ★ Rather, every node in the graph can be connected with every another node in the graph.
- ★ When two nodes are connected via an edge, the two nodes are known as neighbors. For example, in Fig. 2.8, node A has two neighbors: B and D.

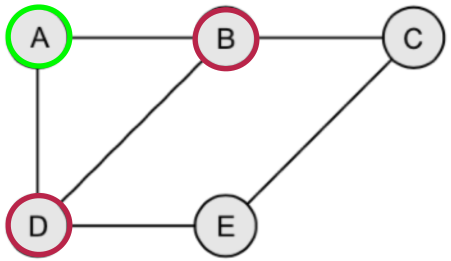

Figure 2.8 Graph

Note *Advantage*: Best models real-world situations
Disadvantage: Some algorithms are slow and very complex

Operation on Data Structures

Traversing It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.

Searching It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.

Inserting It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.

Deleting It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

Operation on Data Structures

Sorting Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

Merging Lists of two sorted data items can be combined to form a single list of sorted data items.

Many a time, two or more operations are applied simultaneously in a given situation. For example, if we want to delete the details of a student whose name is X, then we first have to search the list of students to find whether the record of X exists or not and if it exists then at which location, so that the details can be deleted from that particular location.

Abstract Data Type

★ An abstract data type (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job.

For example, stacks and queues are perfect examples of an ADT.

★ We can implement both these ADTs using an array or a linked list. This demonstrates the 'abstract' nature of stacks and queues.

★ To further understand the meaning of an abstract data type, we will break the term into 'data type' and 'abstract', and then discuss their meanings.

Data Type

★ Data type of a variable is the set of values that the variable can take. We have already read the basic data types in C include

int, char, float, and double.

★ When we talk about a primitive type (built-in data type), we actually consider two things: a data item with certain characteristics and the permissible operations on that data.

For example, an int variable can contain any whole-number value from –32768 to 32767 and can be operated with the operators +, –, *, and /.

Abstract

★ The word 'abstract' in the context of data structures means considered apart from the detailed specifications or implementation.

★ In C, an abstract data type can be a structure considered without regard to its implementation.

★ It can be thought of as a 'description' of the data in the structure with a list of operations that can be performed on the data within that structure.

The end-user is not concerned about the details of how the methods carry out their tasks. They are only aware of the methods that are available to them and are only concerned about calling those methods and getting the results. They are not concerned about how they work.

Abstract

★ For example, when we use a stack or a queue,
the user is concerned only with the type of data and the operations that can be performed on it.

★ Therefore, the fundamentals of how the data is stored should be invisible to the user. They should not be concerned with how the methods work or what structures are being used to store the data.

★ They should just know that to work with stacks, they have push() and pop() functions available to them. Using these functions, they can manipulate the data (insertion or deletion) stored in the stack.

Advantages of Using Abstract Data Type

- ★ In the real world, programs evolve as a result of new requirements or constraints, so a modification to a program commonly requires a change in one or more of its data structures. For example, if you want to add a new field to a student's record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program's efficiency.
- ★ In such a scenario, rewriting every procedure that uses the changed structure is not desirable.
- ★ Therefore, a better alternative is to separate the use of a data structure from the details of its implementation. This is the principle underlying the use of abstract data types.

Algorithms

★ The typical definition of algorithm is 'a formally defined procedure for performing some calculation'.

★ If a procedure is formally defined, then it can be implemented using a formal language, and such a language is known as a programming language.

★ In general terms, an algorithm provides a blueprint to write a program to solve a particular problem.

★ It is considered to be an effective procedure for solving a problem in finite number of steps. That is, a well-defined algorithm always provides an answer and is guaranteed to terminate.

Algorithms

- ★ Algorithms are mainly used to achieve software reuse.
- ★ Once we have an idea or a blueprint of a solution, we can implement it in any high-level language like C, C++, or Java.
- ★ An algorithm is basically a set of instructions that solve a problem.
- ★ It is not uncommon to have multiple algorithms to tackle the same problem, but the choice of a particular algorithm must depend on the time and space complexity of the algorithm.

Different Approaches to Designing an Algorithm

★ Algorithms are used to manipulate the data contained in data structures. When working with data structures, algorithms are used to perform operations on the stored data.

★ A complex algorithm is often divided into smaller units called modules.

★ This process of dividing an algorithm into modules is called modularization.

Different Approaches to Designing an Algorithm

- The key advantages of modularization are as follows:
  - It makes the complex algorithm simpler to design and implement.
  - Each module can be designed independently.

    While designing one module, the details of other modules can be ignored, thereby enhancing clarity in design which in turn simplifies implementation, debugging, testing, documenting, and maintenance of the overall algorithm.

Two Main Approaches to Designing an Algorithm

★ There are two main approaches to design an algorithm,

  ★ top-down approach and bottom-up approach, as shown in Fig. 2.9.

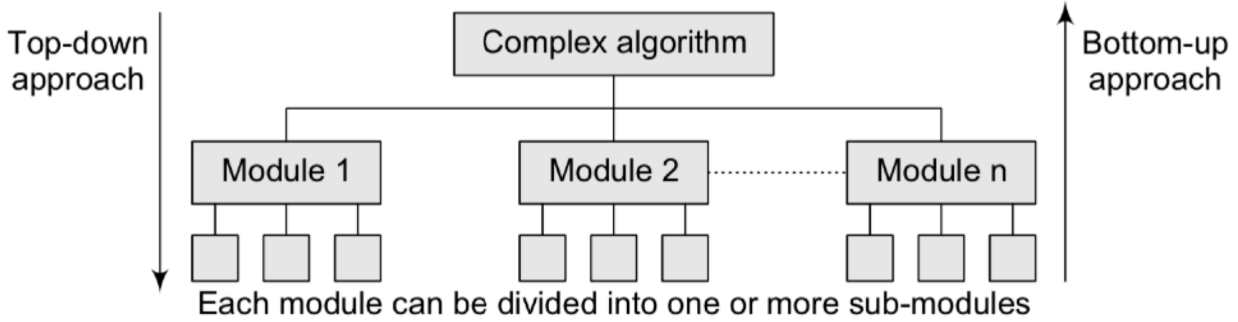

Figure 2.9 Different approaches of designing an algorithm

Top-Down Approach

★ A top-down design approach starts by dividing the complex algorithm into one or more modules.

★ These modules can further be decomposed into one or more sub-modules, and this process of decomposition is iterated until the desired level of module complexity is achieved.

★ Top-down design method is a form of stepwise refinement where we begin with the topmost module and incrementally add modules that it calls.

★ Therefore, in a top-down approach, we start from an abstract design and then at each step, this design is refined into more concrete levels until a level is reached that requires no further refinement.

Bottom-Up Approach

★ A bottom-up approach is just the reverse of top-down approach.

★ In the bottom-up design, we start with designing the most basic or concrete modules and then proceed towards designing higher level modules.

★ The higher level modules are implemented by using the operations performed by lower level modules.

★ Thus, in this approach sub-modules are grouped together to form a higher level module. All the higher level modules are clubbed together to form even higher level modules.

★ This process is repeated until the design of the complete algorithm is obtained.

Top-Down VS Bottom-Up Approaches

★ While top-down approach follows a stepwise refinement by decomposing the algorithm into manageable modules, the bottom-up approach on the other hand defines a module and then groups together several modules to form a new higher level module.

★ Top-down approach is highly appreciated for ease in documenting the modules, generation of test cases, implementation of code, and debugging. However, it is also criticized because the sub-modules are analysed in isolation without concentrating on their communication with other modules or on reusability of components and little attention is paid to data, thereby ignoring the concept of information hiding.

Top-Down VS Bottom-Up Approaches

★ Although the bottom-up approach allows information hiding as it first identifies what has to be encapsulated within a module and then provides an abstract interface to define the module's boundaries as seen from the clients.

★ But all this is difficult to be done in a strict bottom-up strategy.

★ Some top-down activities need to be performed for this.

★ All in all, design of complex algorithms must not be constrained to proceed according to a fixed pattern but should be a blend of top-down and bottom-up approaches.

Control Structure used in Algorithms

- ★ An algorithm has a finite number of steps.
- ★ Some steps may involve decision-making and repetition.
- ★ Broadly speaking, an algorithm may employ one of the following control structures:
  - ★ (a) sequence,
  - ★ (b) decision, and
  - ★ (c) repetition.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: SET SUM = A+B
Step 4: PRINT SUM
Step 5: END
```

Figure 2.10 Algorithm to add two numbers

Sequence

★ By sequence, we mean that each step of an algorithm is executed in a specified order. Let us write an algorithm to add two numbers. This algorithm performs the steps in a purely sequential order, as shown in Fig. 2.10.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: SET SUM = A+B
Step 4: PRINT SUM
Step 5: END
```

Figure 2.10 Algorithm to add two numbers

Decision

★ Decision statements are used when the execution of a process depends on the outcome of some condition.

if `x = y`, then print `EQUAL`.

So the general form of `IF` construct can be given as:

```
IF condition Then process

IF condition
        Then process1
ELSE  process2
```

Decision

★ Decision statements are used when the execution of a process depends on the outcome of some condition.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: IF A = B
            PRINT "EQUAL"
        ELSE
            PRINT "NOT EQUAL"
        [END OF IF]
Step 4: END
```

Figure 2.11 Algorithm to test for equality of two numbers

Repitition

★ Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as while, do–while, and for loops.

★ These loops execute one or more steps until some condition is true. Figure 2.12 shows an algorithm that prints the first 10 natural numbers.

```
Step 1: [INITIALIZE] SET I = 1, N = 10
Step 2: Repeat Steps 3 and 4 while I<=N
Step 3: PRINT I
Step 4: SET I = I+1
        [END OF LOOP]
Step 5: END
```

Figure 2.12 Algorithm to print the first 10 natural of

Examples:- Control Structure

1. Write an algorithm for swapping two values.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: SET TEMP = A
Step 4: SET A = B
Step 5: SET B = TEMP
Step 6: PRINT A, B
Step 7: END
```

Examples:- Control Structure

2. Write an algorithm to find the larger of two numbers.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: IF A>B
            PRINT A
        ELSE
            IF A<B
                PRINT B
        ELSE
                PRINT "The numbers are equal"
                [END OF IF]
        [END OF IF]
Step 4: END
```

Examples:- Control Structure

3. Write an algorithm to find whether a number is even or odd.

```
Step 1: Input number as A
Step 2: IF A%2 =0
                PRINT "EVEN"
        ELSE
                PRINT "ODD"
        [END OF IF]
Step 3: END
```

Examples:- Control Structure

4. Write an algorithm to print the grade obtained by a student using the following rules.

```
Step 1: Enter the Marks obtained as M
Step 2: IF M>75
             PRINT O
Step 3: IF M>=60 AND M<75
             PRINT A
Step 4: IF M>=50 AND M<60
             PRINT B
Step 5: IF M>=40 AND M<50
             PRINT C
        ELSE
             PRINT D
```

Marks	Grade
Above 75	O
60–75	A
50–59	B
40–49	C
Less then 40	D

Examples:- Control Structure

5. Write an algorithm to find the sum of first N natural numbers.

```
Step 1: Input N
Step 2: SET I = 1, SUM = 0
Step 3: Repeat Step 4 while I <= N
Step 4:       SET SUM = SUM + I
              SET I = I + 1
        [END OF LOOP]
Step 5: PRINT SUM
Step 6: END
```

Time and Space Complexity

- ★ Analyzing an algorithm means determining the amount of resources (such as time and memory) needed to execute it.
- ★ Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity.

Time and Space Complexity

- ★ The time complexity of an algorithm is basically the running time of a program as a function of the input size.
- ★ Similarly, the space complexity of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.
- ★ In other words, the number of machine instructions which a program executes is called its time complexity.
- ★ This number is primarily dependent on the size of the program's input and the algorithm used.

Time and Space Complexity

- Generally, the space needed by a program depends on the following two parts:
  - Fixed part: It varies from problem to problem.
    It includes the space needed for storing instructions, constants, variables, and structured variables (like arrays and structures).
  - Variable part: It varies from program to program.
    It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

Time and Space Complexity

- ★ However, running time requirements are more critical than memory requirements.
- ★ Therefore, in this section, we will concentrate on the running time efficiency of algorithms.

Time and Space Complexity

★ *Worst-case running time*

This denotes the behaviour of an algorithm with respect to the worst possible case of the input instance.

The worst-case running time of an algorithm is an upper bound on the running time for any input.

Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

Time and Space Complexity

★ ***Average-case running time***

The average-case running time of an algorithm is an estimate of the running time for an 'average' input.

It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution.

Average-case running time assumes that all inputs of a given size are equally likely.

Time and Space Complexity

★ ***Best-case running time***

The term 'best-case performance' is used to analyse an algorithm under optimal conditions.

For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list.

However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance. It is always recommended to improve the average performance and the worst-case performance of an algorithm.

Time and Space Trade-off

★ The best algorithm to solve a particular problem at hand is no doubt the one that requires less memory space and takes less time to complete its execution.

★ But practically, designing such an ideal algorithm is not a trivial task.

★ There can be more than one algorithm to solve a particular problem.

★ One may require less memory space, while the other may require less CPU time to execute.

Time and Space Trade-off

★ Thus, it is not uncommon to sacrifice one thing for the other.

★ Hence, there exists a time–space trade-off among algorithms.

★ So, if space is a big constraint, then one might choose a program that takes less space at the cost of more CPU time.

★ On the contrary, if time is a major constraint, then one might choose a program that takes minimum time to execute at the cost of more space.

Expressing Time-Space Complexity

- ★ The time and space complexity can be expressed using a function f(n) where n is the input size for a given instance of the problem being solved.
- ★ Expressing the complexity is required when
  - ★ We want to predict the rate of growth of complexity as the input size of the problem increases.
  - ★ There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.
- ★ The most widely used notation to express this function f(n) is the Big O notation. It provides the upper bound for the complexity.

Algorithm Efficiency

- ★ If a function is linear (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains.
- ★ However, if an algorithm contains loops, then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm.

Linear Loop

★ To calculate the efficiency of an algorithm that has a single loop, we need to first determine the number of times the statements in the loop will be executed.

★ This is because the number of iterations is directly proportional to the loop factor. Greater the loop factor, more is the number of iterations. For example, consider the loop given below:

```
for(i=0;i<100;i++)
          statement block;
```

Linear Loop

★ However calculating efficiency is not as simple as is shown in the above example. Consider the loop given below:

```
for(i=0;i<100;i+=2)
        statement block;
```

★ Here, the number of iterations is half the number of the loop factor. So, here the efficiency can be given as

$$f(n) = n/2$$

Logarithmic Loop

★ In logarithmic loops, the loop-controlling variable is either multiplied or divided during each iteration of the loop. For example, look at the loops given below:

```
for(i=1;i<1000;i*=2)
        statement block;
```

```
for(i=1000;i>=1;i/=2)
        statement block;
```

★ Consider the first for loop in which the loop-controlling variable i is multiplied by 2. The loop will be executed only 10 times and not 1000 times because in each iteration the value of i doubles. That is, when n = 1000, the number of iterations can be given by log 1000 which is approximately equal to 10.

```
f(n) = log n
```

Nested Loop

- ★ Loops that contain loops are known as nested loops.
- ★ In order to analyse nested loops, we need to determine the number of iterations each loop completes.
- ★ The total is then obtained as the product of the number of iterations in the inner loop and the number of iterations in the outer loop.
- ★ In this case, we analyse the efficiency of the algorithm based on whether it is a linear logarithmic, quadratic, or dependent quadratic nested loop.

Linear Logarithmic Loop

★ Consider the following code in which the loop-controlling variable of the inner loop is multiplied after each iteration.

★ The number of iterations in the inner loop is log 10. This inner loop is controlled by an outer loop which iterates 10 times.

★ Therefore, according to the formula, the number of iterations for this code can be given as 10 log 10.

```
for(i=0;i<10;i++)
     for(j=1; j<10;j*=2)
          statement block;
```

In more general terms, the efficiency of such loops can be given as `f(n) = n log n`.

Quadratic Loop

★ In a quadratic loop, the number of iterations in the inner loop is equal to the number of iterations in the outer loop.

★ Consider the following code in which the outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times. Therefore, the efficiency here is 100.

```
for(i=0;i<10;i++)
        for(j=0; j<10;j++)
                statement block;
```

The generalized formula for quadratic loop can be given as $f(n) = n^2$.

Dependent Quadratic Loop

★ In a dependent quadratic loop, the number of iterations in the inner loop is dependent on the outer loop. Consider the code given below:

```
for(i=0;i<10;i++)
        for(j=0; j<=i;j++)
                statement block;
```

★ In this code, the inner loop will execute just once in the first iteration, twice in the second iteration, thrice in the third iteration, so on and so forth. In this way, the number of iterations can be calculated as

$$1 + 2 + 3 + \dots + 9 + 10 = 55$$

★ Therefore, the efficiency of such a code can be given as

$$f(n) = n\ (n + 1)/2$$

Big-O Notation

- ★ The Big O notation, where O stands for 'order of', is concerned with what happens for very large values of n.
- ★ For example, if a sorting algorithm performs n^2 operations to sort just n elements, then that algorithm would be described as an $O(n^2)$ algorithm.
- ★ When expressing complexity using the Big O notation, constant multipliers are ignored. So, an O(4n) algorithm is equivalent to O(n), which is how it should be written.
- ★ If f(n) and g(n) are the functions defined on a positive integer number n, then

$$f(n) = O(g(n))$$

Big-O Notation

★ If f(n) and g(n) are the functions defined on a positive integer number n, then

$$f(n) = O(g(n))$$

★ That is, f of n is Big–O of g of n if and only if positive constants c and n exist, such that $f(n) \leq cg(n)$. It means that for large amounts of data, f(n) will grow no more than a constant factor than g(n).

Hence, g provides an upper bound.

Big-O Notation

- ★ Note that here c is a constant which depends on the following factors:
  - ★ the programming language used,
  - ★ the quality of the compiler or interpreter,
  - ★ the CPU speed,
  - ★ the size of the main memory and the access time to it,
  - ★ the knowledge of the programmer, and
  - ★ the algorithm itself, which may require simple but also time-consuming machine instructions.

Big-O Notation

We have seen that the Big O notation provides a strict upper bound for $f(n)$. This means that the function $f(n)$ can do better but not worse than the specified value. Big O notation is simply written as $f(n) \in O(g(n))$ or as $f(n) = O(g(n))$.

Here, n is the problem size and $O(g(n)) = \{h(n): \exists$ positive constants c, n_0 such that $0 \le h(n) \le cg(n), \forall n \ge n_0\}$. Hence, we can say that $O(g(n))$ comprises a set of all the functions $h(n)$ that are less than or equal to $cg(n)$ for all values of $n \ge n_0$.

If $f(n) \le cg(n)$, $c > 0$, $\forall n \ge n_0$, then $f(n) = O(g(n))$ and $g(n)$ is an asymptotically tight upper bound for $f(n)$.

Examples of functions in $O(n^3)$ include: $n^{2.9}$, n^3, $n^3 + n$, $540n^3 + 10$.

Big-O Notation

Examples of functions not in $o(n^3)$ include: $n^{3.2}$, n^2, $n^2 + n$, $540n + 10$, $2n$

To summarize,

- Best case O describes an upper bound for all combinations of input. It is possibly lower than the worst case. For example, when sorting an array the best case is when the array is already correctly sorted.
- Worst case O describes a lower bound for worst case input combinations. It is possibly greater than the best case. For example, when sorting an array the worst case is when the array is sorted in reverse order.

Big-O Notation

Table 2.1 Examples of f(n) and g(n)

g(n)	f(n) = O(g(n))
10	O(1)
$2n^3 + 1$	$O(n^3)$
$3n^2 + 5$	$O(n^2)$
$2n^3 + 3n^2 + 5n - 10$	$O(n^3)$

- If we simply write O, it means same as worst case O.

Now let us look at some examples of g(n) and f(n). Table 2.1 shows the relationship between g(n) and f(n). Note that the constant values will be ignored because the main purpose of the Big O notation is to analyse the algorithm in a general fashion, so the anomalies that appear for small input sizes are simply ignored.

Categories of Big-O Notation

According to the Big O notation, we have five different categories of algorithms:

- Constant time algorithm: running time complexity given as `O(1)`
- Linear time algorithm: running time complexity given as `O(n)`
- Logarithmic time algorithm: running time complexity given as `O(log n)`
- Polynomial time algorithm: running time complexity given as $O(n^k)$ where $k > 1$
- Exponential time algorithm: running time complexity given as $O(2^n)$

Table 2.2 Number of operations for different functions of n

n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4096

Big-O Notation

Example 2.1 Show that $4n^2 = o(n^3)$.

Solution By definition, we have

$$0 \le h(n) \le cg(n)$$

Substituting $4n^2$ as $h(n)$ and n^3 as $g(n)$, we get

$$0 \le 4n^2 \le cn^3$$

Dividing by n^3

$$0/n^3 \le 4n^2/n^3 \le cn^3/n^3$$
$$0 \le 4/n \le c$$

Now to determine the value of c, we see that $4/n$ is maximum when $n=1$. Therefore, $c=4$.

To determine the value of n_0,

$$0 \le 4/n_0 \le 4$$
$$0 \le 4/4 \le n_0$$

Big-O Notation

Example 2.2 Show that $400n^3 + 20n^2 = O(n^3)$.

Solution By definition, we have

$$0 \le h(n) \le cg(n)$$

Substituting $400n^3 + 20n^2$ as $h(n)$ and n^3 as $g(n)$, we get

$$0 \le 400n^3 + 20n^2 \le cn^3$$

Dividing by n^3

$$0/n^3 \le 400n^3/n^3 + 20n^2/n^3 \le cn^3/n^3$$
$$0 \le 400 + 20/n \le c$$

Note that $20/n \to 0$ as $n \to \infty$, and $20/n$ is maximum when $n = 1$. Therefore,

$$0 \le 400 + 20/1 \le c$$

This means, $c = 420$

To determine the value of n_0,

$$0 \le 400 + 20/n_0 \le 420$$
$$-400 \le 400 + 20/n_0 - 400 \le 420 - 400$$
$$-400 \le 20/n_0 \le 20$$
$$-20 \le 1/n_0 \le 1$$

$-20\ n_0 \le 1 \le n_0$. This implies $n_0 = 1$.

Hence, $0 \le 400n^3 + 20n^2 \le 420n^3\ \forall\ n \ge n_0=1$.

Big-O Notation

Example 2.3 Show that $n = O(n \log n)$.

Solution By definition, we have

$$0 \le h(n) \le cg(n)$$

Substituting n as $h(n)$ and $n\log n$ as $g(n)$, we get

$$0 \le n \le c\, n \log n$$

Dividing by $n \log n$, we get

$$0/n \log n \le n/n \log n \le c\, n \log n / n \log n$$
$$0 \le 1/\log n \le c$$

We know that $1/\log n \to 0$ as $n \to \infty$

To determine the value of c, it is clearly evident that $1/\log n$ is greatest when $n=2$. Therefore,

$0 \le 1/\log 2 \le c = 1$. Hence $c = 1$.

To determine the value of n_0, we can write

$$0 \le 1/\log n_0 \le 1$$
$$0 \le 1 \le \log n_0$$

Now, $\log n_0 = 1$, when $n_0 = 2$.

Hence, $0 \le n \le cn \log n$ when $c = 1$ and $\forall\ n \ge n_0 = 2$.

Questions and Answers