Strictly based on the latest CISCE Curriculum

OSWAAL BOOKS

LEARNING MADE SIMPLE

For 2019 Exam

ISC SOLVED PAPER 2018

COMPUTER SCIENCE

OSWAAL BOOKS
1/11, Sahitya Kunj, M.G. Road,
Agra - 282002, UP (India)
Ph.: 0562 2857671, 2527781
email: contact@oswaalbooks.com
website: www.oswaalbooks.com

LATEST SYLLABUS
COMPUTER SCIENCE
CLASS 12

There will be two papers in the subject.

Paper I: Theory - 3 hours ... 70 marks

Paper II: Practical - 3 hours ... 30 marks

PAPER I – THEORY – 70 MARKS

Paper 1 shall be of 3 hours duration and be divided into two parts.

Part I (20 marks) : *This part will consist of compulsory short answer questions, testing knowledge, application and skills relating to the entire syllabus.*

Part II (50 marks) : *This part will be divided into three Sections, A, B and C. Candidates will be required to answer **two** questions out of **three** from Section A (each carrying 10 marks) and two questions out of three from Sections B (each carrying 10 marks) and **two** questions out of **three** from Section C (each carrying 5 marks). Therefore, a total of **six** questions are to be answered in Part II*

SECTION A

1. Boolean Algebra

(a) Propositional logic, well formed formulae, truth values and interpretation of well formed formulae (wff), truth tables, satisfiable, unsatisfiable and valid formulae. Equivalence laws and their use in simplifying wffs.

*Propositional variables; the common logical connectives (~ (not)(negation), Λ (and)(conjunction), V (or)(disjunction), ⇒ (implication), ⇔ (biconditional); definition of a well-formed formula (wff); representation of simple word problems as wff (this can be used for motivation); the values **true** and **false**; interpretation of a wff; truth tables; satisfiable, unsatisfiable and valid formulae.*

*Equivalence laws: commutativity of Λ, V; associativity of Λ, V; distributivity; de Morgan's laws; law of implication (p ⇒ q ≡ ~p V q); law of biconditional ((p ⇔ q) ≡ (p V q) Λ (q ⇒ p)); identity (p ≡ p); law of negation (~ (~p) ≡ p); law of excluded middle (p V~p ≡ **true**); law of contradiction (pΛ~p ≡ **false**); tautology and contingency simplification rules for Λ, V. Converse, inverse and contra positive. Chain rule(Modus ponen).*

p V p ≡ p	*p Λ p ≡ p*
*p V **true** ≡ **true***	*p Λ **true** ≡ p*
*p V **false** ≡ p*	*p Λ **false** ≡ **false***
p V (p V q) ≡ p	*p Λ (p V q) ≡ p*

The equivalence rules can be used to simplify propositional wffs, for example :

1) (p ⇒ q) Λ (p ⇒ r) to p ⇒ (q Λ r)

*2) ((p ⇒ q) Λ p) ⇒ q to **true***

etc.

(b) Binary valued quantities; basic postulates of Boolean algebra; operations AND, OR and NOT; truth tables.

(c) Basic theorems of Boolean algebra (e.g. Duality, idempotence, commutativity, associativity, distributivity, operations with 0 and 1, complements, absorption, involution); De Morgan's theorem and its applications; reducing Boolean expressions to sum of products and product of sums forms; Karnaugh maps (up to four variables).

Verify the laws of boolean algebra using truth tables. Inputs, outputs for circuits like half and full adders, majority circuit etc., SOP and POS representation; Maxterms & Minterms, Canonical and Cardinal representation, reduction using Karnaugh maps and boolean algebra.

2. Computer Hardware

(a) Elementary logic gates (NOT, AND, OR, NAND, NOR, XOR, XNOR) and their use in circuits.

(b) Applications of Boolean algebra and logic gates to half adders, full adders, encoders, decoders, multiplexers, NAND, NOR as universal gates.

Show the correspondence between boolean functions and the corresponding switching circuits or gates. Show that NAND and NOR gates are universal by converting some circuits to purely NAND or NOR gates.

SECTION B

The programming element in the syllabus (Sections B and C) is aimed at algorithmic problem solving and not merely rote learning of Java syntax. The Java version used should be 1.5 or later. For programming, the students can use any text editor and the javac and java programs or any development environment: for example, BlueJ, Eclipse, NetBeans etc. BlueJ is strongly recommended for its simplicity, ease of use and because it is very well suited for an 'objects first' approach.

3. Implementation of algorithms to solve problems:

The students are required to do lab assignments in the computer lab concurrently with the lectures. Programming assignments should be done such that each major topic is covered in at least one assignment. Assignment problems should be designed so that they are non-trivial and make the student do algorithm design, address correctness issues, implement and execute the algorithm in Java and debug where necessary.

Self explanatory.

...contd.

4. Programming in Java (Review of Class XI Sections B and C)

Note that items 4 to 8 will get introduced almost simultaneously when classes and their definitions are introduced.

5. Objects

(a) Objects as data (attributes) + behaviour (methods or functions); object as an instance of a class. Constructors.

Difference between object and class should be made very clear. BlueJ (www.bluej.org) and Greenfoot (www.greenfoot.org) can be profitably used for this purpose. Constructor as a special kind of function; the new operator; multiple constructors with different argument structures; constructor returns a reference to the object.

(b) Analysis of some real world programming examples in terms of objects and classes.

Use simple examples like a calculator, date, number, etc. to illustrate how they can be treated as objects that behave in certain welldefined ways and how the interface provides a way to access behaviour. Illustrate behaviour changes by adding new functions, deleting old functions or modifying existing functions.

6. Primitive values, wrapper classes, types and casting

Primitive values and types: int, short, long, float, double, boolean, char. Corresponding wrapper classes for each primitive type. Class as type of the object. Class as mechanism for user defined types. Changing types through user defined casting and automatic type coercion for some primitive types.

Ideally, everything should be a class; primitive types are defined for efficiency reasons; each primitive type has a corresponding wrapper class. Classes as user defined types. In some cases types are changed by automatic coercion or casting – e.g. mixed type expressions. However, casting in general is not a good idea and should be avoided, if possible.

7. Variables, expressions

Variables as names for values; expressions (arithmetic and logical) and their evaluation (operators, associativity, precedence). Assignment operation; difference between left hand side and right hand side of assignment.

Variables denote values; variables are already defined as attributes in classes; variables have types that constrain the values it can denote. Difference between variables denoting primitive values and object values – variables denoting objects are references to those objects. The assignment operator = is special. The variable on the lhs of = denotes the memory location while the same variable on the rhs denotes the contents of the location e.g. i=i+2.

8. Statements, scope

Statements; conditional (if, if-then-else, switchbreak,?: ternary operator), looping (for, while-do, do-while, continue, break); grouping statements in blocks, scope and visibility of variables.

Describe the semantics of the conditional and looping statements in detail. Evaluation of the condition in conditional statements (esp. difference between || and | and && and &). Emphasize fall through in switch statement. Many small examples should be done to illustrate control structures. Printing different kinds of patterns for looping is instructive. When number of iterations are known in advance use the for loop otherwise the while-do or do-while loop. Express one loop construct using the others. For e.g.:

for (<init>; <test>; <inc>) <stmt>; is equivalent to:

Using while

<init>; while <test> {<stmt>; <inc> }

Using do-while

<init>; if !<test> do <stmt>; <inc> while <test>;

Nesting of blocks. Variables with block scope, function scope, class scope. Visibility rules when variables with the same name are defined in different scopes.

9. Functions

Functions/methods (as abstractions for complex user defined operations on objects), functions as mechanisms for side effects; formal arguments and actual arguments in functions; different behaviour of primitive and object arguments. Static functions and variables. The **this** variable. Examples of algorithmic problem solving using functions (various number theoretic problems, finding roots of algebraic equations).

*Functions are like complex operations where the object is implicitly the first argument. Variable **this** denotes the current object. Functions typically return values, they may also cause sideeffects (e.g. change attribute values of objects) – typically functions that are only supposed to cause side-effects return void (e.g. Set functions). Java passes argument by value. Illustrate the difference between primitive values and object values as arguments (changes made inside functions persist after the call for object values). Static definitions as class variables and class functions visible and shared by all instances. Need for static functions and variables. Introduce the main method – needed to begin execution.*

10. Arrays, strings

(a) Structured data types – arrays (single and multi-dimensional), strings. Example algorithms that use structured data types (e.g. searching, finding maximum/minimum, sorting techniques, solving systems of linear equations, substring, concatenation, length, access to char in string, etc.).

...contd.

Storing many data elements of the same type requires structured data types – like arrays. Access in arrays is constant time and does not depend on the number of elements. Sorting techniques (bubble, selection, insertion). Structured data types can be defined by classes – String. Introduce the Java library String class and the basic operations on strings (accessing individual characters, various substring operations, concatenation, replacement, index of operations). The Class StringBuffer should be introduced for those applications that involve heavy manipulation of strings.

(b) Basic concept of a virtual machine; Java virtual machine; compilation and execution of Java programs (the javac and java programs).

The JVM is a machine but built as a program and not through hardware. Therefore it is called a virtual machine. To run, JVM machine language programs require an interpreter (the java program). The advantage is that such JVM machine language programs (.class files) are portable and can run on any machine that has the java program.

(c) Compile time and run time errors; basic concept of an exception, the Exception class, catch and throw.

Differentiate between compile time and run time errors. Run time errors crash the program. Recovery is possible by the use of exceptions. Explain how an exception object is created and passed up until a matching catch is found. This behaviour is different from the one where a value is returned by a deeply nested function call. It is enough to discuss the Exception class. Sub-classes of Exception can be discussed after inheritance has been done in Class XII.

(d) Class as a contract; separating implementation from interface; encapsulation; private and public.

Class is the basic reusable unit. Its function prototypes (i.e. the interface) work as a visible contract with the outside world since others will use these functions in their programs. This leads to encapsulation (i.e. hiding implementation information) which in turn leads to the use of private and public for realizing encapsulation.

(e) Interfaces in Java; implementing interfaces through a class; interfaces for user defined implementation of behaviour.

Motivation for interface: often when creating reusable classes, some parts of the exact implementation can only be provided by the final end user. For example, in a class that sorts records of different types the exact comparison operation can only be provided by the end user. Since only he/she knows which field(s) will be used for doing the comparison and whether sorting should be in ascending or descending order be given by the user of the class.

Emphasize the difference between the Java language construct interface and the word interface often used to describe the set of function prototypes of a class.

(f) Basic input/output using Scanner and Printer classes from JDK; files and their representation using the File class, file input/output; input/output exceptions. Tokens in an input stream, concept of whitespace, extracting tokens from an input stream (String Tokenizer class).

The Scanner class can be used for input of various types of data (e.g. int, float, char etc.) from the standard input stream or a file input stream. The File class is used model file objects in the underlying system in an OS independent manner. Similarly, the Printer class handles output. Only basic input and output using these classes should be covered.

Discuss the concept of a token (a delimited continuous stream of characters that is meaningful in the application program – e.g. words in a sentence where the delimiter is the blank character). This naturally leads to the idea of delimiters and in particular whitespace and user defined characters as delimiters. As an example show how the String Tokenizer class allows one to extract a sequence of tokens from a string with user defined delimiters.

(g) Concept of recursion, simple recursive functions (e.g. factorial, GCD, binary search, conversion of representations of numbers between different bases). Recursive sorting techniques.

Many problems can be solved very elegantly by observing that the solution can be composed of solutions to 'smaller' versions of the same problem with the base version having a known simple solution. Recursion can be initially motivated by using recursive equations to define certain functions. These definitions are fairly obvious and are easy to understand. The definitions can be directly converted to a program. Emphasize that any recursion must have a base case. Otherwise, the computation can go into an infinite loop. Illustrate this by removing the base case and running the program. Examples:

(i) Definition of factorial :

```
factorial(0) = 1 //base case
factorial(n) = n * factorial(n-1)
```

(ii) Definition of GCD :

```
gcd(m, n) =
if (m==n) then n //base case
else if (m>n) then gcd(m-n, n)
else gcd(m, n-m)
```

(iii) Definition of Fibonacci numbers :

fib(0) = 1 //base case

fib(1) = 1 //base case

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

The tower of Hanoi is a very good example of how recursion gives a very simple and elegant solution where as non-recursive solutions are quite complex. Discuss the use of a stack to keep track of function calls. A stack can also be used to solve the tower of Hanoi problem non-recursively. Merge sort and Quick sort on arrays.

SECTION C

Inheritance, polymorphism, data structures, computational complexity

11. Inheritance and polymorphism

Inheritance; base and derived classes; member access in derived classes; redefinition of variables and functions in subclasses; abstract classes; class Object; protected visibility. Subclass polymorphism and dynamic binding.

Emphasize the following :

- ➢ *inheritance as a mechanism to reuse a class by extending it.*
- ➢ *inheritance should not normally be used just to reuse some functions defined in a class but only when there is a genuine specialization (or subclass) relationship between objects of the base class and that of the derived class.*
- ➢ *Allows one to implement operations at the highest relevant level of abstraction.*
- ➢ *Freezes the interface in the form of abstract classes with abstract functions that can be extended by the concrete implementing classes. For example, an abstract class Shape can have an abstract function draw that is implemented differently in the sub-classes like Circle, Quadrilateral etc.*
- ➢ *how the exact function call at run time depends on the type of the object referenced by the variable. This gives sub-class polymorphism. For example in the code fragment :*

  Shape s1=new Circle(), s2=new Quadrilateral();

  s1.draw(); //the draw is the draw in Circle

  s2.draw(); //the draw is the draw in Quadrilateral the two draw function invocations on s1, s2 invoke different draw functions depending on the type of objects referenced by s1 and s2 respectively.

12. Data structures

(a) Basic data structures (stack, queue, dequeue); implementation directly through classes; definition through an interface and multiple implementations by implementing the interface. Basic algorithms and programs using the above data structures.

A data structure is a data collection with well defined operations and behaviour or properties. The behaviour or properties can usually be expressed formally using equations or some kind of logical formulae. Consider for e.g. a stack with operations defined as follows :

void push(Object o)

Object pop()

boolean isEmpty()

Object top()

Then, for example the LIFO property can be expressed by (assume s is a stack) :

if s.push(o); o1=pop() then o ≡ o1

What the rule says is: if o is pushed on the stack s and then it is popped and o1 is the object obtained then o, o1 are identical.

Another useful property is :

if s.isEmpty() == true then s.pop() = ERROR

It says that popping an empty stack gives ERROR.

Similarly, several other properties can also be specified. It is important to emphasize the behavioural rules or properties of a data structure since any implementation must guarantee that the rules hold.

Some simple algorithms that use the data structures:

(i) For stack: parentheses matching, tower of Hanoi, nested function calls; solving a maze.

(ii) For queue: scheduling processes, printers, jobs in a machine shop.

(b) Recursive data structures: single linked list (Algorithm and programming), binary trees, tree traversals (Conceptual)

Data structures should be defined as abstract data types with a well defined interface (it is instructive to define them using the Java interface construct) – see the comments in (a) above. Emphasize that algorithms for recursive data structures are themselves recursive and that algorithms are usually the simplest and most elegant. The following should be covered for each data structure :

Linked List (single) : *insertion, deletion, reversal, extracting an element or a sublist, checking emptiness.*

Binary trees : *apart from the definition the following concepts should be covered : external and internal nodes, height, level, size, degree, completeness, balancing, Traversals (pre, post and in-order).*

13. Complexity and big O notation

Concrete computational complexity; concept of input size; estimating complexity in terms of functions; importance of dominant term; best, average and worst case. Big O notation for computational complexity; analysis of complexity of example algorithms using the big O notation (e.g. Various searching and sorting algorithms, algorithm for solution of linear equations etc.).

Points to be given particular emphasis:

(i) Algorithms are usually compared along two dimensions – amount of space (that is memory) used and the time taken. Of the two the time taken is usually considered the more important. The motivation to study time complexity is to compare different algorithms and use the one that is the most efficient in a particular situation.

(ii) Actual run time on a particular computer is not a good basis for comparison since it depends heavily on the speed of the computer, the total amount of RAM in the computer, the OS running on the system and the quality of the compiler used. So we need a more abstract way to compare the time complexity of algorithms.

(iii) This is done by trying to approximate the number of operations done by each algorithm as a function of the size of the input. In most programs the loops are important in deciding the complexity. For example in bubble sort there are two nested loops and in the worst case the time taken will be proportional to n(n-1) where n is the number of elements to be sorted. Similarly, in linear search in the worst case the target has to be compared with all the elements so time taken will be proportional to n where n is the number of elements in the search set.

(iv) In most algorithms the actual complexity for a particular input can vary. For example in search the number of comparisons can vary from 1 to n. This means we need to study the best, worst and average cases. Comparisons are usually made taking the worst case. Average cases are harder to estimate since it depends on how the data is distributed. For example in search, if the elements are uniformly distributed it will take on the average n/2 comparisons when the average is taken over a statistically significant number of instances.

*(v) Comparisons are normally made for large values of the input size. This means that the dominant term in the function is the important term. For example if we are looking at bubble sort and see that time taken can be estimated as: a*n2 +b*n + c where n is the number of elements to be sorted and a, b, c are constants then for large n the dominant term is clearly n2 and we can, in effect, ignore the other two terms.*

*All the above motivates the big O notation. Let f(n), g(n) be positive functions, then f(n) is said to be O(g(n)) if there exists constants c, n0 such that f(x)≤ c*g(n) whenever n>n0. What this means is that g(n) asymptotically dominates f(n). Expressing time complexity using the big O notation gives us an abstract basis for comparison and frees us from bothering about constants. So the estimated time complexity a*n2+b*n+c is O(n2).*

Analyse the big O complexity of the algorithms pertaining to the data structures in 11 (a) and (b) above.

❑❑

ISC Solved Paper, 2018
Class-XII
Computer Science
Paper-I (Theory)

(Maximum Marks : 70)

(Time allowed : Three hours)

*(Candidates are allowed additional 15 minutes for **only** reading the paper.*

They must NOT start writing during this time.)

---

*Answer all questions in Part I (compulsory) and six questions from Part-II, choosing two questions from **Section-A**, two from **Section-B** and two from **Section-C**.*

All working, including rough work, should be done on the same sheet as the rest of the answer.

The intended marks for questions or parts of question are given in brackets [ ].

PART- I (20 Marks)

*Answer **all** questions*

While answering question in this Part, indicate briefly your working and reasoning, wherever required.

1. **(a)** State the *Commutative law* and prove it with the help of a the truth table. 1

(b) Convert the following expression into its canonical POS form : 1

$$F(X, Y, Z) = (X + Y')\cdot(Y' + Z)$$

(c) Find the dual of : 1

$$(A' + B)\cdot(1 + B') = A' + B$$

(d) Verify the following proposition with the help of a truth table : 1

$$(P \wedge Q) \vee (P \wedge \sim Q) = P$$

(e) If F(A, B, C) = A' (BC' + B'C), then find F' 1

2. **(a)** What are *Wrapper classes* ? Given *any two* examples. 2

(b) A matrix A[m][m] is stored in the memory with each element requiring 4 bytes of storage. If the base address at A[1][1] is 1,500 and the address of A[4][5] is 1,608, determine the order of the matrix when it is stored in **Column Major Wise.** 2

(c) Convert the following *infix notation* to *postfix* form : 2

A + (B – C * (D/E) * F)

(d) Define *Big 'O' notation.* State the *two* factors which determine the complexity of an algorithm. 2

(e) What is *exceptional handling* ? Also, state the purpose of *finally* block in a try catch statement. 2

3. The following is a function of some class which checks if a positive integer is a Palindrome number by returning true or false. *(A number is said to be palindrome if the reverse of the number is equal to the original number.)* The function does not use modulus (%) operator to extract digit. There are some places in the code marked by ?1?, ?2?, ?3?, ?4?, ?5? which may be replaced by a statement /expression so that the function works properly.

```
boolean PalindromeNum(int N)
{
int rev = ?1?
int num = N;
while (num > 0)
{
```

```
            int f = num/10;
            int s = ?2?;
            int digit = num – ?3?
            rev = ?4? + digit;
            num /= ?5?
        }
        if (rev = = N)
            return true;
        else
            return false;
    }
```

(i) What is the statement or expression at ?1? **1**

(ii) What is the statement or expression at ?2? **1**

(iii) What is the statement or expression at ?3? **1**

(iv) What is the statement or expression at ?4? **1**

(v) What is the statement or expression at ?5? **1**

PART- II

(50 Marks)

*Answer **six** questions in this part, choosing **two** questions from Section A, **two** from Section B and **two** from Section C.*

SECTION - A

*Answer **any two** question.*

4. **(a)** Given the Boolean function **F(A, B, C, D,) = $\sum$ (0, 2, 4, 8, 9, 10, 12, 13).**

(i) Reduce the above expression by using 4-variable Karnaugh map, showing the various groups (*i.e.*, octal, quads and pairs). **4**

(ii) Draw the logic gate diagram for the reduced expression. Assume that the variables and their complements are available as inputs. **1**

(b) Given the Boolean function **F(A, B, C, D) = π (3, 4, 5, 6, 7, 10, 11, 14, 15).**

(i) Reduce the above expression by using 4-variable Karnaugh map, showing the various groups (*i.e.* octal, quads and pairs). **4**

(ii) Draw the logic gate diagram for the reduced expression. Assume that the variables and their complements are available as inputs. **1**

5. **(a)** A training institute intends to give scholarships to its students as per the criteria given below : **5**

- The student has excellent academic record but is financially weak.

OR

- The student does not have an excellent academic record and belongs to a backward class.

OR

- The student does not have an excellent academic record and is physically impaired.

The inputs are :

INPUTS	
A	Has excellent academic record
F	Financially sound
C	Belongs to a backward class
I	Is physically impaired

(In all the above cases 1 indicates yes and 0 indicates no).

Output : X [1 indicates yes, 0 indicates no for all cases]

Draw the truth table for the inputs and outputs given above and write the SOP expression for X (A,F, C,I).

(b) Using the truth table, state whether the following proposition is a *tautology, contingency* or *a contradiction* : **3**

$$\sim (A \wedge B) \vee (\sim A \Rightarrow B)$$

(c) Simplify the following expression, using Boolean laws : **2**

$$A \bullet (A' + B) \bullet C \bullet (A + B)$$

6. (a) What is a *Encoder* ? Draw the Encoder circuit to convert A-F hexadecimal numbers to binary. State an application of a Multiplexer. **5**

(b) Differentiate between *Half Adder* and *Full Adder.* Draw the logic circuit diagram for a Full Adder. **3**

(c) Using only NAND gates, draw the logic circuit diagram for A' + B. **2**

SECTION - B

*Answer **any two** questions.*

Each program should be written in such a way that it clearly depicts the logic of the problem.

This can be achieved by using mnemonic names and comments in the program.

*(Flowcharts and Algorithms are **not** required.)*

The programs must be written in Java.

7. Design a class **Perfect** to check if a given number is a perfect number or not. [A number is said to be perfect if sum of the factors of the number excluding itself is equal to the original number] **10**

Example : 6 = 1 + 2 + 3 (where 1, 2 and 3 are factors of 6, excluding itself)

Some of the members of the class are given below :

Class name	:	**Perfect**
Data members/instance variables :		
num	:	to store the number
Methods/Member functions :		
Perfect (int nn)	:	parameterized constructor to initialize the data member num=nn
int sum_of_factors (int i)	:	returns the sum of the factors of the number(num), excluding itself, using **recursive technique**
void check ( )	:	checks whether the given number is perfect by invoking the function *sum_of_factors ( )* and displays the result with an appropriate message

Specify the class **Perfect** giving details of the **constructor ( ), int sum_of_factors(int)** and **void check ( ).** Define a **main ( )** function to create an object and call the functions accordingly to enable the task.

8. Two matrices are said to be equal if they have the same dimension and their corresponding elements are equal. **10**

For example the two matrices A and B given below are equal :

Matrix A

1	2	3
2	4	5
3	5	6

Matrix B

1	2	3
2	4	5
3	5	6

Design a class **EqMat** to check if two matrices are equal or not. Assume that the two matrices have the same dimension.

Some of the members of the class are given below :

Class name	:	**EqMat**
Data members/instance variables :		
a[ ] [ ]	:	to store integer elements
m	:	to store the number of rows
n	:	to store the number of columns
Member functions / methods :		
EqMat(int mm, int nn)	:	parameterised constructor to initialise the data members m = mm and n = nn
void readarray ( )	:	to enter elements in the array
int check (EqMat P, EqMat Q)	:	checks if the parameterized objects P and Q are equal and returns 1 if true, otherwise returns 0
void print ( )	:	displays the array elements

Define the class **EqMat** giving details of the **constructor ( ), void readarray ( ), int check(EqMat, EqMat)** and **void print ( ).** Define the **main ( )** function to create objects and call the functions accordingly to enable the task.

9. A class **Capital** has been defined to check whether a sentence has words beginning with a capital letter or not. Some of the members of the class are given below : 10

Class name	:	**Capital**
Data member/instance variables :		
sent	:	to store a sentence
freq	:	stores the frequency of words beginning with a capital letter
Member functions / methods :		
Capital( )	:	default constructor
void input( )	:	to accept the sentence
boolean isCap(String w)	:	checks and returns true if word begins with a capital letter, otherwise returns false
void display( )	:	displays the sentence along with the frequency of the words beginning with a capital letter

Specify the class **Capital,** giving the details of the **constructor( ), void input( ), boolean isCap(String)** and **void display( )**. Define the **main( )** function to create an object and call the functions accordingly to enable the task.

SECTION - C

*Answer **any two** questions.*

Each program should be written in such a way that it clearly depicts the logic of the problem stepwise.

This can be achieved by using comments in the program and mnemonic names or pseudo codes for algorithms. The programs must be written in Java and the algorithms must be written in general / standard form, wherever required / specified.

*(Flowcharts are **not** required.)*

10. A super class **Number** is defined to calculate the factorial of a number. Define a sub class **Series** to find the sum of the series S = 1! + 2! + 3! + 4! + ... + *n* ! 5

The details of the members of both the classes are given below :

Class name	:	**Number**
Data member/instance variable :		
	:	to store an integer number
Member functions / methods :		
Number(int nn)	:	parameterized constructor to initialize the data member n = nn
int factorial(int a)	:	returns the factorial of a number (factorial of n = 1 × 2 × 3 × ..... × n)
void display ( )	:	displays the data members
Class name :		**Series**
Data member/instance variable :		
sum	:	to store the sum of the series
Member functions/methods :		
Series(...)	:	Parameterized constructor to initialize the data members of both the classes
void calsum( )	:	calculates the sum of the given series
void display( )	:	displays the data members of both the classes

*Assume that the super class **Number** has been defined.* Using the **concept of inheritance,** specify the class **Series** giving the details of the **constructor(...), void calsum( )** and **void display( ).**

The super class, main function and algorithm need NOT be written.

11. **Register** is an entity which can hold a maximum of 100 names. the register enables the user to add and remove names from the top most end only 5

Define a class **Register** with the following details :

Class name	:	**Register**
Data Members/instance variables :		
stud[ ]	:	array to store the names of the students
cap	:	stores the maximum capacity of the array
top	:	to point the index of the top end

Member functions :

Register (int max)	:	constructor to initialize the data member cap = max, top = – 1 and create the string array
void push(String n)	:	to add names in the register at the top location if possible otherwise display the message "OVERFLOW"
String pop( )	:	removes and returns the names from the top most location of the register if any, else returns "$$"
void display( )	:	displays all the names in the register

(a) Specify the class **Register** giving details of the functions **void push(String)** and **String pop( ).** Assume that the other functions have been defined.

The main function and algorithm need NOT be written.

(b) Name the entity used in the above data structure arrangement .

12. **(a)** A linked list is formed from the objects of the class **Node.** The class structure of the Node is given below : **2**

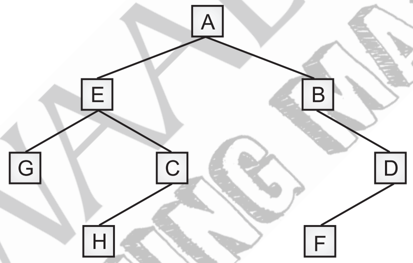
```
class Node
{
        int n;
        Node    link;
}
```

Write an *Algorithm* **OR** a *Method* to search for a number from an existing linked list.

The method declaration is as follows :

void FindNode(Node str, int b)

(b) Answer the following questions from the diagram of a Binary Tree given below :



(i) Write the inorder traversal of the above tree structure. **1**

(ii) State the height of the tree, if the root is at level 0 (zero) **1**

(iii) List the leaf nodes of the tree. **1**

■■

ANSWERS

PART-I

(20 Marks)

1. **(a)** **Commutative law :**

A + B = B + A Additive and Multiplicative A·B = B·A

A	B	A+B	A·B
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Table 1

B	A	B+A	B·A
0	0	0	0
1	0	1	0
0	1	1	0
1	1	1	1

Table 2

From table 1 and table 2

A + B = B + A and A·B = B·A

(b)

$$F(X, Y, Z) = (X + Y')\cdot(Y' + Z)$$
$$= (X + Y' + ZZ')\ (XX' + Y' + Z)$$
$$= (X + Y' + Z)\ (X + Y' + Z')\ (X + Y' + Z)\ (X' + Y' + Z)$$

(c) Dual of $(A' + B)\cdot(1 + B') = A' + B$

is $(A' \cdot B) + (0 \cdot B') = A' \cdot B$

(d)

P	Q	~Q	P ∧ Q	P ∧ ~ Q	(P ∧ Q) ∨ (P ∧ ~ Q)
0	0	1	0	0	0
0	1	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1

From the table $(P \wedge Q) \vee (P \wedge \sim Q) = P$

(e)

$$F(A, B, C) = A'\ (BC' + B'C)$$
$$F' = (A'\ (BC' + B'C))'$$
$$= A + (BC' + B'C)'$$
$$= A + (BC')'\ (B'C)'$$
$$= A + (B' + C)\ (B + C')$$

2. (a) A wrapper class wraps around a data type and gives it an object appearance. Wherever, the data type is required as an object, this object can be used. Wrapper classes include type methods to unwrap the object and give back the data type.

e.g., Integer, Boolean

(b) Formula for finding out address of A[J] [K] element is column-major order of Matrix (MXN) order.

$$LOC\ (A[J]\ [K]) = Base\ (A) + w\ (M(K-1) + (J-1))$$

Given : $M = N = m,\ w = 4$ bytes

Bass address = 1500

address of A[4] [5] = 1608

$$LOC\ (A\ [4]\ [5]) = 1500 + 4(m\ (5-1) + (4-1))$$
$$1608 = 1500 + 4(4\ m + 3)$$
$$1608 - 1500 = 16\ m + 12$$
$$16\ m = 108 - 12 = 96$$
$$m = 6$$

Therefore,

Order of the matrix is (6×6)

(c)

Scanned	Stack	Postfix Expression
	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
–	(+(–	AB
C	(+(–	ABC
*	(+(–*	ABC
(	(+(–*(	ABC
D	(+(–*(	ABCD
/	(+(–*(/	ABCD
E	(+(–*(/	ABCDE
)	(+(–*	ABCDE/

*	(+(–**	ABCDE/
F	(+(–**	ABCDE/F
)	(+	ABCDE/F**–
)		ABCDE/F**–+

Postfix form of A + (B – C * (D/E) * F) = ABCDE/F**–+

Infix → A + (B – C * (D/E) * F)

= A + (B – C * **DE/** * F)

= A + (B – **CDE/** * * F)

= A + (B – **CDE/** * **F** *)

= A + (B C D E/ * F * –)

= ABCDE/* F * – +

(d) **Big 'O' Notation :** Big O notation is used to describe the performance or complexity of an algorithm.

For a given function $g(n)$, it is denoted by $O(g(n))$ the set of functions

$$\mathbf{O}(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

Two factors which determine the complexity of an algorithm :

(1) **Time Complexity :** The amount of computer time, algorithm needs to run to completion.

(2) **Space Complexity** : The amount of memory, algorithm needs to run to completion.

(e) **Exceptional Handling :** A Java exception is an object that describes an exceptional (that is error) condition that has occurred in a piece of code. When an exceptional condition arise, an object representing that exception is created and thrown in the method that caused the error. The method may choose to handle the exception itself, or pass it on.

Java exception handling is managed via five keywords. try, catch, throws and finally.

Any code that absolutely must be executed after a try block completes is put under finally block.

3. **(i)** ? 1 ? ⇒ 0

(ii) ? 2 ? ⇒ rev * 10

(iii) ? 3 ? ⇒ f * 10

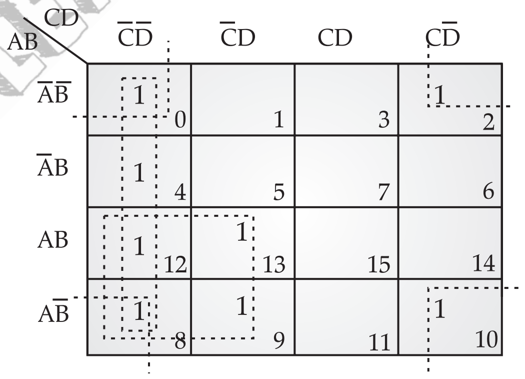
(iv) ? 4 ? ⇒ s

(v) ? 5? ⇒ 10;

PART- II

SECTION - A

4. **(a)** (i) F(A, B, C, D) = Σ(0, 2, 4, 8, 9, 10, 12, 13)



I quad $(m_0 + m_2 + m_8 + m_{10}) = \bar{B}\,\bar{D}$

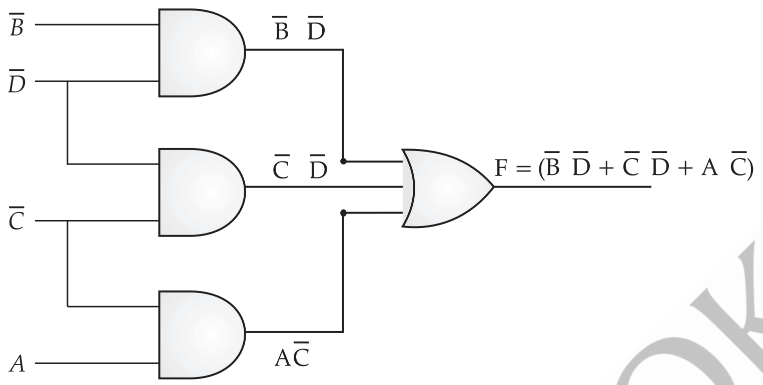
II quad $(m_0 + m_4 + m_{12} + m_8) = \bar{C}\,\bar{D}$

III quad $(m_{12} + m_{13} + m_8 + m_9) = A\bar{C}$

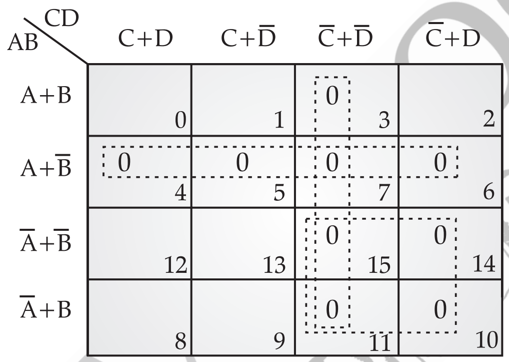
so the reduced expression is,

$$F(A, B, C, D) = \bar{B}\,\bar{D} + \bar{C}\,\bar{D} + A\bar{C}$$

(ii)



(b) (i) $F(A, B, C, D) = \pi\,(3, 4, 5, 6, 7, 10, 11, 14, 15)$



AB \ CD	C+D	C+$\bar{D}$	$\bar{C}$+$\bar{D}$	$\bar{C}$+D
A+B	0	1	0 3	2
A+$\bar{B}$	0 4	0 5	0 7	0 6
$\bar{A}$+$\bar{B}$	12	13	0 15	0 14
$\bar{A}$+B	8	9	0 11	0 10

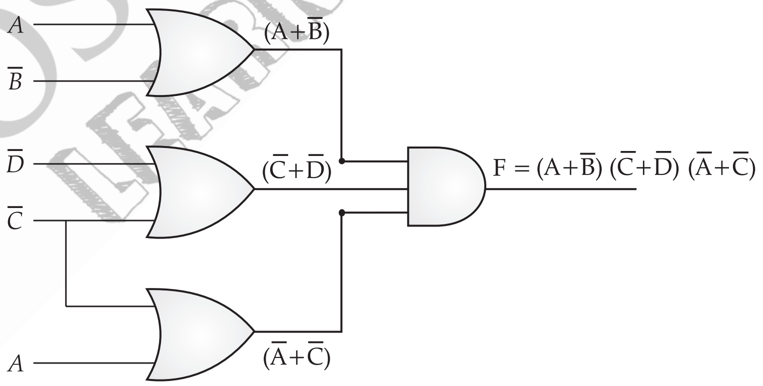
I quad ($M_4 M_5 M_7 M_6$) = $A+\bar{B}$

II quad ($M_3 M_7 M_{15} M_{11}$) = $(\bar{C}+\bar{D})$

III quad ($M_{15} M_{14} M_{11} M_{10}$) = $(\bar{A}+\bar{C})$

$F(A, B, C, D) = A+\bar{B}\ (\bar{C}+\bar{D})\ (\bar{A}+\bar{C})$

(ii)



5. **(a)** From the question, we have four inputs A, F, C, I and on output X. The truth table for given variables is shown.

A	F	C	I	X
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1

1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

So, the SOP expression is,

A' F' C' I + A' F' C I' + A' F' CI + A' FC' I + A' FC I' + A' FCI + AFC' I' + AFC' I + AFC I' + AFCI

(b)

A	B	$A \wedge B$	$\sim (A \wedge B)$	$\sim A$	$\sim A \Rightarrow B$	$\sim(A \wedge B) \vee (\sim A \Rightarrow B)$
0	0	0	1	1	0	1
0	1	0	1	1	1	1
1	0	0	1	0	1	1
1	1	1	0	0	1	1

$\sim (A \wedge B) \vee (\sim A \Rightarrow B)$ have all 1's, therefore Proposition is a tautology.

(c)

$A \cdot (A' + B) \cdot C \cdot (A + B) = (A \cdot A' + A \cdot B) \cdot (C \cdot A + C \cdot B)$ (Distributive law)

$= AB \cdot (CA + CB)$ (Complement law)

$= AB \cdot CA + AB \cdot CB$ (Distributive law)

$= ABC + ABC$ (Idempotent law)

$= ABC$ (Idempotent law)

6. (a) Encoder : An encoder is a digital function that produces a reverse operation from that a decoder. An encoder has 2^n (or less) input limes and n output lines. The output lines generate the binary code for the 2^n input variables.

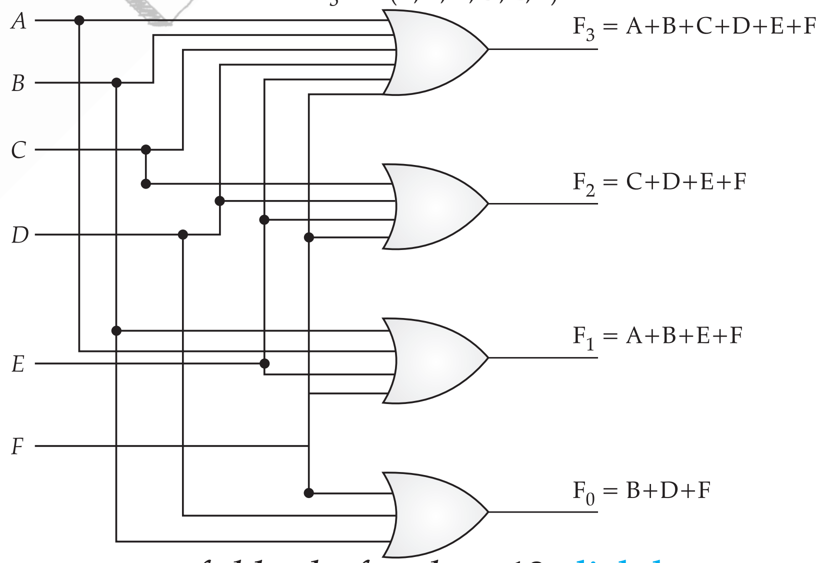
Hexadecimal No	F_3	F_2	F_1	F_0
A	1	0	1	0
B	1	0	1	1
C	1	1	0	0
D	1	1	0	1
E	1	1	1	0
F	1	1	1	1

$F_0 = \Sigma(B, D, F)$

$F_1 = \Sigma(A, B, E, F)$

$F_2 = \Sigma(C, D, E, F)$

$F_3 = \Sigma(A, B, C, D, E, F)$

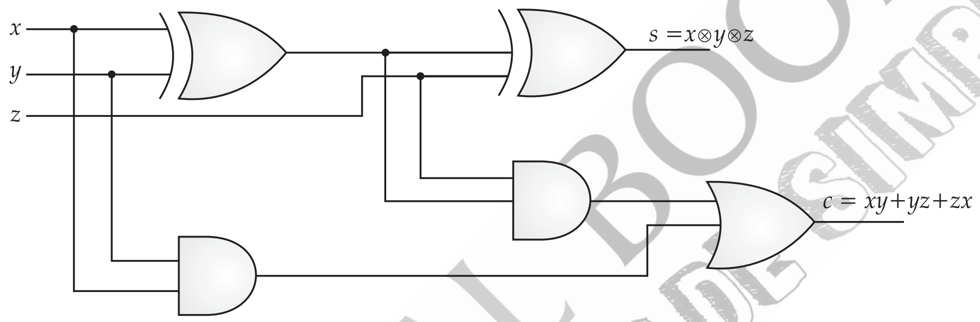


In telephone network, multiple audio signals are integrated on a single line for transmission with the help of multiplexers.

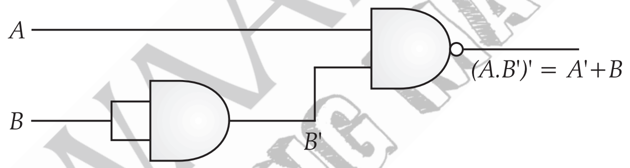
(b)

S.No.	Half Adder	Full Adder
1.	A combinational circuits that performs the addition of two bits is called a half adder	A combinational circuit that performs the addition of three bits (two bits are significant and one is a previous carry) is called a full adder
2.	The half adder circuits needs two binary inputs and two binary outputs.	It consists of three binary inputs and two binary outputs.
3.	Half adder does not have carry in input.	Full adder has carry in input.

Full adder circuit diagrams :



(c)



SECTION–B

```
7. import  java. io. *;
   class Perfect
       {
            int num;
            Perfect (int nn)
            {
                num = nn;
            }
            int sum_of_factors (int i)
            {
                if (num = = i)
                     return (sum_of_factors (i/2));
                else if (i = = 1)
                     return (1);
                else if (num % i = = 0)
                     return (i + sum_of_factors (i – 1));
                else return (sum_of_factors (i – 1));
            }
            void check ( )
            {
                if (num = = sum_of_factors (num))
                     System. out. println (" Perfect Number");
                else System. out. println ("Not Perfect Number");
            }
       public    static void main (String args [ ] )
       {
            system. out. println ("Enter one number");
```

```
    BufferedReader br = new BufferedReader  (new InputStreamReader (System.in));
            int n = Integer parseInt (br. readLine( ));
            Perfect ob = new Perfect (n);
            ob. check ( );
        }
    }
```

8.
```
    import java. io. *;
    class EqMat
    {
        int  a [ ] [ ];
        int  m, n;
        EqMat (int mm, int nn)
        {
            m = mm;
            n = nn;
            a = new int[m][n];
        }
    void readarray (  )
    {
        int  i, j;
        BufferedReader br = new BufferedReader (new InputStreamReader (System. in));
        System. out. println ("Enter" + m + n + "values");
        for (i = 0; i < m; i + +)
        {
            for (j = 0; j < n; j ++)
                a [i] [j] = Integer. parseInt (br. readLine ());
            }
        }
        int check (EqMat P, EqMat Q)
        {   int i, j;
            for (i = 0; i < m; i ++)
            {
                for (j = 0; j < n; j++)
                    if (P. a [i] [j] ! =  Q. a [i] [j])
                        return (0);
            }
            return (1);
        }
    void print ( )
    {   int i, j;
        for (i = 0; i < m; i ++)
        {
            for (j = 0; j < n; j ++)
                System. out. print (a [i] [j]);
            system. out. println ( ) ;
        }
    }
    public static void main (String args [ ])
    {
        BufferedReader br = new BufferedReader (new InputStreamReader (System. in));
        system. out. println ("Enter no. of rows of  matrix");
        int row = Integer. parseInt  (br. readLine ( ));
        system . out. println ("Enter no. of Columns of matrix ");
        int col = Integer. parseInt (br. readLine ( ));
        EqMat Ob1 = new EqMat (row, col);
        Ob1. readarray ( );
        Ob1. print ( );
        EqMat Ob2 = new EqMat (row, col);
        Ob2. readarray ( );
        Ob2. print ( );
        EqMat Ob3 = new EqMat (row, col);
        if (Ob3. check (Ob1, Ob2) = = 1)
            system. out. println (" Matrices are equal");
        else
```

```
          system. out. println (" Matrices are not equal");
      }
  }
```

9.
```
  import java. io. * ;
  class Capital
  {
      String sent;
      int freq;
      Capital ( )
      {    sent = " ";  freq = 0;
      }
      void input (  )
  {
      BufferedReader br = new BufferedReader
      (new InputStreamReader (System. in));
      system. out. println ("Enter one sentence");
      sent =  br. readLine (  );
      String [  ] word = sent. split  (" \\ s");
      for (int i = 0; i < word. length; i ++) {
          if (isCap (word [i]) = = true)
                            freq ++;
      }
  }
      boolean isCap (String w)
      {
  if (Character. isUpperCase (w. charAt (0)) = = true)
          return true;
          else  return false;
      }
      void display (  )
      }
          System. out. println (sent);
          System. out. println (" the frequency of the words beginning with a capital
      letter is "+ freq);
      }
      public static void main ( )
      {
          Capital Ob = new Capital  (  );
          Ob. input ( );
          Ob. display ( );
      }
  }
```

SECTION–C

10.
```
  class Series extends Number
  {
          int  sum;
          Series  (int  nn)
          {
              super (nn);
              sum = 0;
          }
          void calsum ( )
          {    int  i;
              for (i = 1; i < n; i ++)
              {
                      sum = sum + factorial (i);
              }
          }
          void display ( )
          {
                  system. out.  println ("n = "+ n);
                  system. out. println (" sum of series is " + sum);
             }
    }
```

11. (a)

```
class Register
{
          String  stud [ ];
          int cap, top;
          Register (int  max)
          {   cap = max;
              top = – 1;
              stud = new int  [cap];
          }
          void push (String n)
          {
              if (top = = cap – 1)
                   System. out. println ("OVER FLOW");
              else
              {
                   top ++;
                   stud [ top] = n;
              }
          }
          String  pop ( )
          {
              if (top = = – 1)
                   return ("$ $");
              else
              {
                   String w = stud [top];
                   top – – ;
                   return (w);
              }
          }
     }
```

11. (b) Stack

12. (a)

```
class Node
{
     int n;
     Node link;
void FindNode (Node str, int b)
{
     Node t;
     t = str;
     while  (t ! = null)
     {
         if (t. n = = b)
         {
              system. out. println  ("Number is present in linked list");
              system. exit(0);
         }
         t = t. link;
     }
     system. out. println ("Number is  not present in linked list");
     }
}
```

(b) (i) In order traversal $\rightarrow$ left root right
G E H C A B F D

(ii) height of the tree = 3

(iii) leaf nodes = G, H, F

❒❒