1. Introduction

Q.4.2.1.1 What is an algorithm? Discuss the characteristics of an algorithm.
Answer: An algorithm is a set of instructions to perform a task. An algorithm which can be executed by a computer is called a computer algorithm.
All algorithms need to perform a task. So, all algorihms have some output. But an algorithm may not necessarily have an input. For example, let us consider an algorithm to generate 10 prime numbers. Here we do not have any input to be given as the given task is very specific. Let us modify the definition of our problem. Suppose we would like to generate n prime numbers, where n is an arbitrary integer. n must be given some value at the run time. So, the algorithm of this task has an input. In most of the cases, an algorithm performs a finite number of steps. Each step of an algorithm must be unambiguous, and a basic one to carry out the given task.

Q.1.4.1.2 Elaborate the idea of abstract data type (ADT)? Give examples.
Answer: An abstract data type is a collection of data (D), set of functions (F), and a set of rules (R) followed by the functions in F.
Consider abstract data type SET, where SET is a collection of well defined elements.
Example 1: Some of the operations that might be defined on abstract data type SET are given below:
1. $DefSet(A)$: This procedure makes the null set A.
2. $Union(A, B, C)$: This procedure makes a new set C as the union of sets A and B.
3. $Member(a, A)$: This function returns true if $a \in A$, else it returns false.
We can define a number of other operations on SET. Each set of operations performed on SET defines a distinct ADT. Let us define the triplet ($the\ set\ of\ elements$, $set\ of\ functions\ as\ defined\ above$, $the\ set\ rules\ that\ the\ functions\ follow$) as an abstract data type, called SET1. The set of operations are performed on the data of the instance variables of SET1 data type. Each programming language facilitates to implement different data types using its in-built data types. Note that we can implement

SET1 using different in-built data types in different ways.
Example 2: Another set of operations performed on data type SET are given below.
1. $DefSet(A)$: This procedure makes the null set A.
2. $Union(A, B, C)$: This procedure makes a new set C as the union of set A and set B.
3. $Intersection(A, B, C)$: This procedure makes a new set C as the intersection of set A and set B.
4. $Size(A)$: This function finds the number of elements of the set A.

Let us call the corresponding abstract data type as SET2. Note that we have defined another set of operations on SET in Example 2. Some operations of ADTs SET1 and SET2 are common.

Q.1.4.1.3 What are the advantages of algorithms over flowcharts?
Answer: The advantages of algorithms include the following.
i. Algorithms can be expressed in a shorter form.
ii. Some problems involve parallel tasks. We can express such parallel tasks using some special keywords in an algorithm. But a flowchart is sequential in nature.

Q.4.2.1.4 Design a recursive algorithm to multiply two non-negative integers.
Answer: Function *multiply* () is expressed recursively as follows:

```
function multiply $(x, y)$
    // It multiplies two non-negative integers $x, y$
    if $(x = 0)$ or $(y = 0)$ then return (0); end if
    if $(x = 1)$ then return $(y)$; end if
    if $(y = 1)$ then return $(x)$; end if
    return $(y\ +\ multiply\ (x - 1, y))$;
end function
```

Function *multiply*() keeps on adding y for x times.

Q.4.2.1.5 Define greatest common divisor *gcd*() function using recursion.
Answer: The greatest common divisor (*gcd*) of integers x and y is defined recursively as follows: $gcd(x, y) = \begin{cases} gcd(y, x), & \text{if } y > x \\ x, & \text{if } y = 0 \\ gcd(y, mod(x, y)), & \text{otherwise} \end{cases}$

Q.4.2.1.6 Present an algorithm to check the primality of an integer.

Answer: Function *isPrime* () returns true, when the argument is prime, else it returns false.

```
function $isPrime$ ($x$)
  for $i \leftarrow 2$ to $\sqrt{x}$ do
       if $x\ mod\ i = 0$ then return (false); end if
  end for
  return (true);
end function
```

It can be shown that an integer $x(> 2)$ is prime if it is not divisible by the following integers: $2, 3, 4, \ldots, int(\sqrt{x})$.

Q.4.2.1.7 Obtain prime factors of an integer using an algorithm.
Answer: All the factors of x, except 1 and x, lie between 2 and $x\ div\ 2$.

```
function $primeFactors$ ($x$)
 for $i \leftarrow 2$ to ($x\ div\ 2$) do
      if ($x\ div\ i = 0$) then
         if $isPrime$ ($i$) then print ($i$); end if
      end if
 end for
end function
```

The function *primeFactors*() calls *isPrime*() of Q.4.2.1.6 repeatedly to find all the prime factors.

Q.4.2.1.8 Discuss the idea of dangling pointer in a programming language that supports pointer.
Answer: Many programming languages such as C and C++, have features of handling pointer variables. A pointer variable is used to store a memory location, that may be a start of a list, or array, or starting address of a useful information. When the purpose of the memory, whose start address is held in a pointer variable p is served, one may wish to free the memory for other purposes as required by the operating system. After the memory becomes free, by issuing an instruction such as $free(p)$ in C, the content of p becomes useless or illegal. Then pointer p is called a dangling pointer. It is better to set $p =$ NULL, after executing $free(p)$. It implies that p does not point to any memory location.

2. Arrays, Iteration and Invariants

Q.4.2.2.1 Consider a string represented by C language. Find a procedure (function) to copy a string into an array.
Answer: A string in C language is ended with a null character ('\0'). Function *stringToArray*() copies content of string *s* into an array *a*.

```
void stringToArray (char s[ ], char a[ ]) {
      int i = 0;
      while (s[i] != '\0') {
              a[i] = s[i]; i++;
      }
      return;
}
```

The last character of string *s*, i.e. the null character, is not copied into array *a*.

Q.4.2.2.2 Write a function to copy an array into a string using C language.
Answer: Consider an array *a* containing *n* characters. The function *arrayToString*() copies the content of *a* into a string *s*. To make *s* as a valid string, the last character of *s* is kept as null ('\0').

```
void arrayToString (char a [ ], char s [ ], int n) {
      int i;
      for (i = 0; i < n; i++) {
           s [i] = a [i];
      }
      s [i] = '\0';
      return;
}
```

Q.4.2.2.3 Design a recursive function in C language to reverse a string read by the user.
Answer: A recursive function *reverse*() is designed below.

```
void reverse() {
   char c;
   if ((c = getchar()) != '\n') reverse();
```

```
  putchar(c);
  return;
}
```

Using *getchar*() function, a character is read, and assigned to *c*. Then the character is checked with return character ('\n'). A recursive call is made, when the return character is not entered. The *putchar*() function is not executed untill the user enters a string ended by return character ('\n'). When the return character is entered, the last character is displayed first by the *putchar*() function. The compiler uses system stack to hold all the pending *putchar*() functions for the subsequent executions. Data structure stack being a LIFO data structure, the function *reverse*() reverses a string read by the user.

Q.4.2.2.4 Find a procedure to find maximum of n numbers ($n \geq 1$).
Answer: Pseudo code of the algorithm is designed as follows.

```
1  read ($n$);
2  read ($a$);
3  max = $a$; $i = 1$;
4  for $i = 2$ to $n$ do
5    read ($a$);
6    if ($max < a$) then $max = a$; end if
7  end for
8  print ($max$);
```

At every iteration, the current maximum is compared with current value read, and the current maximum *max* is updated, when the current value is larger. Note that we need not store all the elements entered by the user.

Q.4.2.2.5 Write an algorithm to generate all permutations of 0, 1, 2 using (i) *for* loop, (ii) *while* loop, (iii) *goto* statement.
Answer: (i) Permutations of 0, 1, 2 using *for* loop are retured by procedure *permute*1 ().

```
procedure permute1 ()
for $i = 0$ to 2 do
 for $j = 0$ to 2 do
  for $k = 0$ to 2 do
   if ($i \neq j$) and ($j \neq k$) then print ($i, j, k$);
  end for
 end for
end for
end procedure
```

Three nested *for* loops on variables i, j and k will generate all permutations of (i, j, k).

(ii) *permute*2 () finds all permutations of 0, 1 and 2 using *while* statement.

```
procedure permute2 ()
i = 0;
while $i \leq 2$ do
  $j = 0$;
  while $j \leq 2$ do
   $k = 0$;
   while $k \leq 2$ do
    if $(i \neq j)$ and $(j \neq k)$ then print $(i, j, k)$;
    $k = k + 1$;
   end while
   $j = j + 1$;
  end while
  $i = i + 1$;
 end while
end procedure
```

Three nested *while* loops on variables i, j and k will generate all permutations of (i, j, k). This algorithm is the same as that of (i). The only difference is that a *for* loop is implemented using a *while* loop.

(iii) *permute*3 () prints all permutations of 0, 1 and 2 using *goto* statement.

```
procedure permute3 ()
$i \leftarrow 0$;
loopi: if $(i < 3)$ then
   $j \leftarrow 0$;
   loopj: if $(j < 3)$ then
     $k \leftarrow 0$;
     loopk: if $(k < 3)$ then
        if $(i \neq j)$ and $(j \neq k)$ then print $(i, j, k)$; end if
        $k \leftarrow k + 1$;
        goto loopk;
```

3. Lists, Recursion, Stacks and Queues

Q.4.2.3.1 State some disadvantages of linear list.
Answer: A few disadvantages of linear linked list are given below.
i. It is not possible to reach a preceding node from a given node.
ii. The head pointer, the address of the start node, needs to be reserved always.

Q.4.2.3.2 We are given an array of numbers sorted in ascending order. The array has been rotated from element with the least index to element with highest index for a number of times. Then the array becomes a collection of two sorted lists. Design an algorithm to find pivot element.
Answer: We first try to understand the rotation process. Let the sorted elements in array A be 1, 4, 7, 9, 12. We assume that index starts from 0. After 2^{nd} rotation, the array becomes 7, 9, 12, 1, 4. The pivot element becomes 12, i.e., $A(2)$.

```
function $findPivot$ ($A$, $low$, $high$)
  if ($high<low$) return -1; end if
  if ($high = low$) return $low$; end if
  $mid = (low + high)/2$;
  if ($mid < high$) and ($A(mid) > A(mid+1)$) return mid; end if
  if ($mid > low$) and ($A(mid) < A(mid-1)$) return ($mid-1$); end if
  if ($A(low) \geq A(mid)$) return $findPivot(A, low, mid-1)$; end if
  return $findPivot(A, mid+1, high)$;
end function
```

Q.4.2.3.3 Design an algorithm to search an element in a pivoted sorted array.
Answer: Refer to Q.1.4.3.2 regarding pivoted sorted array. Now we present an algorithm to search an element in a pivoted sorted array of size n.

```
function $searchPivotedArray$ ($A, n, key$)
  pivot = $findPivot(A, 0, n-1)$;
  // If the array is not rotated then apply binary search on the whole array
  if (pivot = -1) return $binarySearch$ ($A, 0, n-1, key$); end if
  if ($A(pivot) = key$) return pivot; end if
  if ($A(0) \leq key$) return $binarySearch$ ($A, 0, pivot - 1, key$); end if
  return $binarySearch$ ($A, pivot + 1, n - 1, key$);
end function
```

Q.4.2.3.4 Can you implement a queue using stack(s)?
Answer: Yes, it is possible to implement a queue using two stacks. Let $S1$ and $S2$ be two stacks. We discuss here two major operations *enQueue* and *deQueue* for adding and deleting an item for a queue respectively. In this technique, for *enQueue* operation, the new element is entered at the top of stack $S1$. In *deQueue* operation, if stack $S2$ is empty then all the elements are moved to stack $S2$, and then the top of $S2$ is returned. We discuss here both $enQueue()$ and $deQueue()$ operations.

```
$enQueue(q, x)$
  i. push $x$ into stack $S1$;
```

```
$deQueue(q)$
  i. pop an element from stack $S2$, and return it;
  ii. if $S2$ is empty then
        while $S1$ is not empty do
          pop $x$ from $S1$;
          push $x$ into $S2$;
        end while
      end if
  iii. if ($S1$ is empty) and ($S2$ is empty) then display error; end if
```

$deQueue(q)$ takes $O(n)$ time, and $enQueue(q, x)$ takes $O(1)$ time.

Q.4.2.3.5 Is it possible to implement a stack using queue(s)?
Answer: A stack can be implemented using two queues. Let S be a stack to be implemented using queues $Q1$ and $Q2$. Main operations of a stack are $push()$ and $pop()$ as discussed below.

```
$push(S, x)$
  i. enqueue $x$ to $Q2$;
  ii. for every $y \in Q1$ do
        dequeue $y$ from $Q1$;
        enqueue $y$ to $Q2$;
      end for
      swap the names of $Q1$ and $Q2$;
```

```
pop(S)
  i. dequeue an item from Q1, and return it;
```

push(S, x) takes $O(n)$ time, and *pop*(S) takes $O(1)$ time.

Q.4.2.3.6 Present an algorithm to reverse a circular linked list.
Answer: We assume the following node structure of linked list.

```
structure node
   int data;
   structure node* link;
end structure
```

Note that the second field is a pointer type, that poits to a similar structure of type *node*. Algorithm *reverse*() is given below to reverse a linked list pointed by *head*.

```
function reverse (head)
   if (head = NULL) return NULL; end if
   // reverse technique is same as reversing a singly linked list
   prev = NULL;
   current = head;
   repeat
      next = current→link;
      current→link = prev;
      prev = current;
      current = next;
   until (current = head);
   // adjusting the links so as to make the last node point to the first node
   head →link = prev;
   head = prev;
   return head;
end function
```

Statements under *repeat-until* block is repeated unless the *current* points to the node where *head* points to. The time complexity of *reverse*() algorithm is $O(n)$, where n is the number of nodes.