1. Introduction

Q.4.4.1.1 State criteria that an algorithm satisfies.
Answer: The following criteria are satistied by an algorithm.
- ⋆ [Input] It has n inputs, $n \geq 0$.
- ⋆ [Output] It has p outputs, $p \geq 1$.
- ⋆ [Definiteness] All the instructions are clear and unambiguous.
- ⋆ [Finiteness] It contains a finite number of instructions.
- ⋆ [Effectivenes] Each instruction must be basic and feasible.

Q.4.4.1.2 How does a program relate to an algorithm? Do all the programs become an algorithm?
Answer: A program is an expression of an algorithm, written in a computer programming language. An operating system is a complex computer program that never stops, even when the computer remains idle. It fails to satisfy the finiteness property of an algorithm.

Q.4.4.1.3 Solve the following Fibonacci recurrence relation.
$F_n = F_{n-1} + F_{n-2},\ n = 3, 4, 5, \dots \qquad (1.1)$
where, $F_1 = F_2 = 1$
Answer: We shall solve the recurrence relation using ordinary generating function, given as follows:
Let $G(x) = u_1 + u_2x + u_3x^2 + \cdots + u_{n-1}x^{n-2} + u_nx^{n-1} + \dots$
where, $u_1 = u_2 = 1$.
Using (1.1), $\sum_{n=3}^{\infty} u_nx^n = \sum_{n=3}^{\infty} u_{n-1}x^n + \sum_{n=3}^{\infty} u_{n-2}x^n$
Then, $x\sum_{n=3}^{\infty} u_nx^{n-1} = x^2\sum_{n=3}^{\infty} u_{n-1}x^{n-2} + x^3\sum_{n=3}^{\infty} u_{n-2}x^{n-3}$
or, $x[G(x) - u_1 - u_2x] = x^2[G(x) - u_1] + x^3G(x)$
or, $G(x) - u_1 - u_2x = x[G(x) - u_1] + x^2G(x)$, for $x \neq 0$
or, $G(x)[1 - x - x^2] = u_1 + u_2x - u_1x = u_1 = 1$, since $u_1 = u_2 = 1$
or, $G(x) = (1 - x - x^2)^{-1} = \frac{1}{1-x-x^2}$
or, $G(x) = \frac{1}{(x-\alpha)(x-\beta)}$, where α, β are the roots of equation $1-x-x^2 = 0$
or, $x^2 + x - 1 = 0$
i.e. $\alpha = \frac{-1+\sqrt{5}}{2}, \beta = \frac{-1-\sqrt{5}}{2}$
or, $G(x) = -\frac{1}{\alpha-\beta}[\frac{1}{x-\alpha} - \frac{1}{x-\beta}]$
or, $G(x) = -\frac{1}{\sqrt{5}}[\frac{1}{x-\alpha} - \frac{1}{x-\beta}]$
or, $G(x) = +\frac{1}{\sqrt{5}}[\frac{1}{\alpha}(1 - \frac{x}{\alpha})^{-1} - \frac{1}{\beta}(1 - \frac{x}{\beta})^{-1}] \qquad (1.2)$
(1.2) is an identity. We equate co-efficient of x^{n-1} on both sides.

Here, $u_n =$ co-efficient of x^{n-1} in the left side of (1.2)
Then, $u_n = \frac{1}{\sqrt{5}}[\frac{1}{\alpha}.\frac{1}{\alpha^{n-1}} - \frac{1}{\beta}.\frac{1}{\beta^{n-1}}] = \frac{1}{\sqrt{5}}[\frac{1}{\alpha^n} - \frac{1}{\beta^n}]$
So, $u_n = \frac{1}{\sqrt{5}}[\frac{\beta^n - \alpha^n}{(\alpha\beta)^n}] = \frac{\alpha^n - \beta^n}{\sqrt{5}}$,
where $\alpha\beta = \frac{1}{4}((-1)^2 - (\sqrt{5})^2) = -1$

$$\text{Thus, } u_n = \frac{\alpha^n - \beta^n}{\sqrt{5}} = \frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}} \tag{1.3}$$

Formula (1.3) is called Binet's formula in honour of the mathematician who first proved it.

Q.4.4.1.4 Explain the following notions: algorithm validation, program verification.
Answer: Algorithm validation is a process of testing the algorithm. It requires checking correct answer for all possible inputs.
A complete proof of correctness of a program is called program verification. It involves expressing a program using a set of assertions. These assertions are normally expressed in predicate calculus.

Q.4.4.1.5 Give a non-inductive proof of the identity.
$\sum_{i=0}^{n}\left[i.\binom{n}{i}\right] = n.2^{n-1}$
Answer: Consider all the strings of n bits. Let us count the number of 1s in all these strings.
There are $\binom{n}{i}$ strings of having i 1s. Then total number of 1s is $\sum_{i=0}^{n}\left[i.\binom{n}{i}\right]$.
There are 2^n strings with each of them having length n. Total number of bits is $n.2^n$. Half of the bits are 1s. Then total number of 1s $= \frac{1}{2}.n.2^n = n.2^{n-1}$.

Q.4.4.1.6 Consider the following *for* loop structure.

```
for x = t1 to t2 step p do {
  s1;
  ...
  sn;
}
```

Here, $t1$ and $t2$ denote the initial and final values of variable x. At every step, the value of x is incremented by p. $s1, \ldots, sn$ are some statements with in the body of *for* loop. Express the given *for* loop using *while* loop.
Answer: A equivalent code for the given *for* loop is expressed using a *while* loop.

```
x = t1;
```

```
final = t2;
while ((x - final) <= 0) do
  s1;
  ...
  sn;
  x = x + p;
}
```

Q.4.4.1.7 Write an algorithm to sort a binary array in linear time.
Answer: Let A be an array with n bimary elements. Then each element is either 0 or 1. If we count the number of 0s in a binary array, and put them at the beginning of the array and then put 1s in the remaining cells, the sorting is done. We assume that array index starts from 1. Sorting algorithm $SortBinary()$ is given below.

```
procedure SortBinary (A, n)
   zeros = 0;
   for i = 1 to n do
      if (A(i) = 0) then zeros = zeros + 1; end if
   end for
   j = 1;
   while (zeros > 0) do
      A(j) = 0; j = j + 1; zeros = zeros - 1;
   end while
   while ( j <= n) do
      A(j) = 1; j = j + 1;
   end while
end procedure
```

Q.4.4.1.8 Prove the identity on Fibonacci numbers:
$F_n^2 - F_{n+1}F_{n-1} = (-1)^{n-1}, \forall n \geq 2$.
Answer: We shall prove the result by induction on n.
For $n = 2$, $F_2^2 - F_3F_1 = F_2^2 - (F_1 + F_2)F_1$
$= 1^2 - (1+1)1 = -1 = (-1)^{2-1}$, where $F_2 = F_1 = 1$
Then the result is true for $n = 2$.
Assume that the result is true for $n \leq k - 1$. We shal show that the result is true for $n = k$.
$F_k^2 - F_{k+1}F_{k-1} = F_k^2 - (F_k + F_{k-1})F_{k-1}$
$= (F_k - F_{k-1})F_k - F_{k-1}^2 = F_{k-2}F_k - F_{k-1}^2$ [Fibonacci recurrence relation]
$= -(F_{k-1}^2 - F_{k-2}F_k) = (-1)(-1)^{(k-1)-1}$ [Induction hypothesis]
$= (-1)^{k-1}$

2. Complexity

Q.4.4.2.1 Explain the notions of O, Ω and Θ.
Answer: Let $\mathbb{N}$ and $\mathbb{R}^+$ be the sets of natural numbers and positive real numbers respectively. Suppose that $f,\ g : \mathbb{N} \to \mathbb{N}$.
$\triangleright$ $f(n) = O(g(n))$ if there exist positive constants $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0$, $0 \leq f(n) \leq c.g(n)$.
$\triangleright$ $f(n) = \Omega(g(n))$ if there exist positive constants $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0$, $f(n) \geq c.g(n) \geq 0$.
$\triangleright$ $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Q.4.4.2.2 Show that $\lceil log\ n \rceil = O(n)$.
Answer: First, we shall show that $\lceil log\ n \rceil \leq n$, for $n \geq 1$. We shall apply induction principle to show the inequality.
For $n = 1$, $\lceil log\ n \rceil = 0$. So, the result is true for $n = 1$.
For $n > 1$, we assume that $\lceil log\ (n-1) \rceil \leq n - 1$ (induction hypothesis)
Now, $\lceil log\ n \rceil \leq \lceil log\ (n-1) \rceil + 1$
or, $\lceil log\ n \rceil \leq (n-1) + 1$, by induction hypothesis
or, $\lceil log\ n \rceil \leq n$
It implies that $\lceil log\ n \rceil = O(n)$.
Here, we can take $c = 1, n_0 = 1$. (See definition of big-oh in Q.4.4.2.1)

Q.4.4.2.3 Prove that $3n\lfloor log\ n \rfloor = O(n^2)$.
Answer: First, we shall show that $3n\lfloor log\ n \rfloor \leq 3n^2$, for $n \geq 1$. We shall apply induction principle to show the inequality.
For $n = 1$, $3n\lfloor log\ n \rfloor = 0$, and $3n^2 = 3$. So, the result is true for $n = 1$.
For $n > 1$, we assume that $3n\lfloor log\ (n-1) \rfloor \leq 3(n-1)^2$ (induction hypothesis)
Now, $3n\lfloor log\ n \rfloor \leq 3n(\lfloor log\ (n-1) \rfloor + 1)$
or, $3n\lfloor log\ n \rfloor \leq 3(n-1)(\lfloor log\ (n-1) \rfloor + 1) + 3(\lfloor log\ (n-1) \rfloor + 1)$
or, $3n\lfloor log\ n \rfloor \leq 3(n-1)\lfloor log\ (n-1) \rfloor + 3(n-1) + 3(\lfloor log\ (n-1) \rfloor + 1)$
or, $3n\lfloor log\ n \rfloor \leq 3(n-1)^2 + 3(n-1) + 3(\lfloor log\ (n-1) \rfloor + 1)$
(By induction hypothesis)
or, $3n\lfloor log\ n \rfloor \leq 3(n-1)^2 + 3(n-1) + 3n$ (see solution of Q.4.4.2.2)
or, $3n\lfloor log\ n \rfloor \leq 3n^2 - 6n + 3 + 3n - 3 + 3n$
or, $3n\lfloor log\ n \rfloor \leq 3n^2$
This implies that $3n\lfloor log\ n \rfloor = O(n^2)$.
Here, we can take $c = 3, n_0 = 1$. (See definition of big-oh in Q.4.4.2.1)

Q.4.4.2.4 Prove by induction: $\binom{n}{n/2} = \Omega\left(2^n/n\right)$, for all even n

Answer: For $n = 2$, $\binom{n}{n/2} = \binom{2}{1} = 2$.

$2^n/n = 2^2/2 = 2$

Thus, $\binom{n}{n/2} \geq 2^n/n$, for $n = 2$.

Assume that the result is true for $n = 2k$. Then $\binom{2k}{k} = \Omega\left(2^{2k}/2k\right)$.

$$\text{or, } \binom{2k}{k} \geq 2^{2k-1}/k \qquad (2.1)$$

Now, we consider for $n = 2k + 2$.

$\binom{2k+2}{k+1} = \frac{(2k+2)!}{(k+1)!(k+1)!} = \frac{(2k+2)(2k+1)}{(k+1)(k+1)} \binom{2k}{k}$

$\geq \frac{(2k+2)(2k+1)}{(k+1)(k+1)} \cdot \frac{2^{2k-1}}{k}$, using (2.1)

$= \frac{2k+1}{k} \cdot \frac{2^{2k-1}}{k+1} = \frac{2k+1}{2k} \cdot \frac{2^{2k+2}}{2(k+1)}$

$= \left(1 + \frac{1}{2k}\right) \cdot \frac{2^{2k+2}}{2k+2} \geq \frac{2^{2k+2}}{2k+2}$

Induction step follows.

Thus, the result is true. Here, we take $c = 1$, and $n_0 = 2$.

Note: This result can also be verified by Stirling's approximation given as follows: $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Q.4.4.2.5 Find the complexity of procedure $ABC()$.

```
1  procedure ABC (int n)
2    count = 0;
3    for i = n div 2 to n step 1 do
4      for j = 1 to n step 2*j do
5        for k = 1 to n step 2*k do
6          count = count + 1;
7        end for
8      end for
9    end for
10 end procedure
```

Answer: We shall count the frequencies of different loops. Other statements do not affect the asymptotic time complexity of the procedure. Frequencies of lines 3, 4, and 5 are in $O(n), O(log\ n)$ and $O(log\ n)$ respectively. Thus, the complexity of procedure $ABC()$ is $O(n(log\ n)^2)$.

Q.4.4.2.6 Show that $7 \times 2^n + n^2 = \Theta(2^n)$.

Answer: Refer to notion of *Theta* given in Q.4.4.2.1. It can be defined in another way:

The function $h(n) = \Theta(g(n))$ if and only if there exist positive constant

a, b and n_o such that $a \times g(n) \leq h(n) \leq b \times g(n)$, for all $n \geq n_o$.
Now, $7\times\ 2^n + n^2 \leq 10 \times 2^n, n \geq 1$ [See Q.4.4.2.7]
Also, $1 \times 2^n \leq 7 \times 2^n + n^2, n \geq 1$
Thus, $1 \times 2^n \leq 7 \times 2^n + n^2 \leq 10 \times 2^n$
Here, $a = 1, b = 10, n_o = 1$.

Q.4.4.2.7 Show that $n^2 \leq 3 \times 2^n$, for $n \geq 1$.
Answer: We shall prove the result using the method of induction on n.
For $n = 1, n^2 = 1$, and $3 \times 2^n = 6$.
The result is true for $n = 1$.
We assume that the result is true for $n = k$.
Then $k^2 \leq 3 \times 2^k$ (induction hypothesis)
We need to prove that the results holds for $n = k + 1$.
$(k + 1)^2 = k^2 + 2k + 1 \leq 3 \times 2^k + 2k + 1$ (by induction hypothesis)
$< 3 \times 2^k + 2(k + 1)$
$< 3 \times 2^k + 3(k + 1)$
$< 3 \times 2^k + 3 \times 2^k$, for $k \geq 2$
$= 3 \times 2^{k+1}$
Thus, the induction step follows.

Q.4.4.2.8 Show that $n^{1.001} + nlogn = \Theta(n^{1.001})$
Answer: $c_1 n^{1.001} \leq n^{1.001} + nlogn$, where $c_1 = 1$
Let c_2 be 2^{1000}.
Then $c_2 \times n^{1.001} = 2^{1000} \times (2^{1000})^{1.001}$, where $n = 2^{1000}$
$= 2^{1000} \times 2^{1001}$
Now, $n^{1.001} + nlogn{=}(2^{1000})^{1.001} + 2^{1000} log_2 2^{1000}$
$= 2^{1001} + 1000 \times 2^{1000} {=} 2^{1000}(2 + 1000) = 1002 \times 2^{1000}$
Now, $c_2 \times n^{1.001} = 2^{1000} \times 2^{1001} \geq 1002 \times 2^{1000} = n^{1.001} + nlogn$
Thus, $c_1 \times n^{1.001} \leq n^{1.001} + nlogn \leq c_2 \times n^{1.001}$, for $n_0 = 2^{1000}$, $c_1 = 1$, $c_2 = 2^{1000}$
Thus, $n^{1.001} + nlogn = \Theta\ (n^{1.001})$ [See Q.4.4.2.6]

Q.4.4.2.9 Discuss the notions of o and ω with the help of examples.
Answer: $f(n) = o(g(n))$ if and only if $0 \leq f(n) < cg(n))$
for *all* constants $c > 0, n \geq n_0$
For example, $4n = o(n^2)$, but $4n^2 \neq o(n^2)$
In other words, $f(n) = o(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) = \omega(g(n))$ if and only if $0 \leq cg(n) < f(n)$
for *any* positive constant $c > 0, n \geq n_0$
For example, $4n^2 = \omega(n)$, but $4n^2 \neq \omega(n^2)$

3. Data Structures

Q.4.4.3.1 Suppose an array $A(1 \ldots 4, 1 \ldots 3, 1 \ldots 3)$ is stored at the address $(789F)_H$. Let us assume that it is stored in row major form. What is the address of the (3, 2, 1)-th element?
Answer: The element of A will be stored in the following order.
$A(1,1,1)A(1,1,2)A(1,1,3)A(1,2,1)A(1,2,2)A(1,2,3)A(1,3,1)A(1,3,2)A(1,3,3$
$A(2,1,1)A(2,1,2)A(2,1,3)A(2,2,1)A(2,2,2)A(2,2,3)A(2,3,1)A(2,3,2)A(2,3,3$
$A(3,1,1)A(3,1,2)A(3,1,3)\mathbf{A(3,\ 2,\ 1)}A(3,2,2)A(3,2,3)A(3,3,1)A(3,3,2)A(3,3$
$A(4,1,1)A(4,1,2)A(4,1,3)A(4,2,1)A(4,2,2)A(4,2,3)A(4,3,1)A(4,3,2)A(4,3,3$
Assume that integer takes two bytes.
Address of (3, 2, 1)-th element $= (789F)_H + [\{(3-1) \times 3 \times 3 + (2-1) \times 3 + 1 - 1\} \times 2]_{10}$
$= (789F)_H + (42)_{10} = (789F)_H + (2A)_H = (78C9)_H$
Note: Suffixes H represents hexadecimal number, and 10 represents decimal number.

Q.4.4.3.2 Let A and B be two lower triangular matrices, each of order $n \times n$. Devise a scheme to represent both the triangles in an array $C(1:n, 1:n+1)$.
Answer: Total number of elements in each matrix on or below diagonal $= 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$.
The number of non-zero elements in both the matrices
$= \frac{n(n+1)}{2} \times 2 = n(n+1)$.
Thus, we need a matrix of order $n \times (n+1)$ to store both the matrices together. Here C is used to store all the elements of A and B.
Let $A = [a_{ij}], B = [b_{ij}], C = [c_{ij}]$.

$$c_{ij} \leftarrow a_{ij} \begin{array}{l} \text{for } i = 1,\ 2,\ \ldots,\ n \\ \text{for } j = 1,\ 2,\ \ldots,\ n \end{array} \tag{3.1}$$

$$c_{i(n+1)} \leftarrow b_{ii} \ \text{ for } i = 1,\ 2,\ \ldots,\ n \tag{3.2}$$

$$c_{(n-j)(n+2-i)} \leftarrow b_{ij} \begin{array}{l} \text{for } i = 2,\ 3,\ \ldots,\ n \\ \text{for } j = 1,\ 2,\ \ldots,\ (i-1) \end{array} \tag{3.3}$$

Mappings for the elements in (3.3) are given below.
Row 2 of $B \xrightarrow{\textit{mapped to}}$ Column n of C
Row 3 of $B \xrightarrow{\qquad\qquad}$ Column $(n-1)$ of C

Row 4 of $B \longrightarrow$ Column $(n-2)$ of C
...
In general, Row i of $B \longrightarrow$ Column $(n+2-i)$ of C
Column 1 of $B \longrightarrow$ Row $(n-1)$ of C
Column 2 of $B \longrightarrow$ Row $(n-2)$ of C
Column 3 of $B \longrightarrow$ Row $(n-3)$ of C
...
In general, Column j of $B \longrightarrow$ Row $(n-j)$ of C

Q.4.4.3.3 Write an algorithm to find the number of occurrences of string $S1$ in $S2$.
Answer: We use variable *count* to keep the frequency of string $S1$ in string $S2$. We assume that a string is ended with a null character ($'\backslash 0'$) as implemented in C language. Function *Occurrences*() counts the number of times string $S1$ occurs in string $S2$.

```
function Occurrences (S1, S2)
  count = 0; i = 0; j = 0;
  while (S2(j) != '\0') do
  L: k = j;
     while (S1(i) = S2(j)) and (S1(i) != '\0') do
       i = i+1; j = j+1;
       if (S2(i) = '\0') and (S1(i) != '\0') then
         goto E;
       end if
     end while
     if (S1(i) = '\0') then
       count = count+1;
     else
       i = 0; j = k+1;
       goto L;
     end if
  end while
  E: return (count);
end function
```

When there is a match of the first character of $S1$ with a character of $S2$, the *while* loop keeps matching the charaters in $S1$ and $S2$. If we reach the last character of $S1$, i.e., null character ($'\backslash 0'$), then we have got an instance of $S1$ in $S2$, and *count* is incremented by 1. Otherwise, the indices of $S1$ and $S2$, i.e., i and j respectively, are updated.

Q.4.4.3.4 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, and it points to a similar structure of type *node* as followed in C language. Algorithm *Reverse*() is given below to reverse a linked list pointed by *head*.

```
procedure 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 procedure
```

Statements under *repeat-until* loop 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 in the circular linked list.

Q.4.4.3.5 Write a procedure to return the n-th data from the end of a linked list.
Answer: We assume the following node structure of a linked list.

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