Q.1) What is an algorithm? Explain in detail about various characteristics of an algorithm.

Ans.1)

Algorithm :- An algorithm is any well-defined computational procedure that takes some value or set of value as input and produces some value of set of values as output.

An algorithm is a sequence of computational steps that program or transform input into output.

Problem → Algorithm → program

Algorithm ↓ programming language

The study of algorithm is called algorithmic ~~choice~~

★ **Properties of algorithm**

i) **Input output :** Each algorithms is supplied with zero or more external quantities and also produces at least one quantities as a output.

ii) **Definitness :** Each instruction of an algorithm must be clear and unambitious

Page No.
Date

iii) Finitness : Algorithm ~~to~~ must terminate after finite number of steps.

iv) performance : The performance of algorithm is measured in terms of space and time complexity.

→ Space complexity : The total amount of memoral required to ~~pro~~ perform a task.

→ Time complexity : The amount of time required to execute a perticular task.

★ Arithmetic series :

Let S_n be the sum of first n terms of arithmatic series
$a ; a+d , a+2d , a+3d , \ldots , a+(n-1)d$
then,

$$S_n = an + \frac{n(n-1)d}{2}$$

$S_n = a + a+d + \ldots + a+(n-2)d + a+(n-1)d$ — ①
$S_n = a+(n-1)d + a+(n-2)d + \ldots + a+d+a$ — ②

Adding ① & ②

$2S_n = a+a+(n-1)d + a+d + a+(n-2)d + \cdots +$
$a+(n-2)d + a+d + a+(n-1)d + a$

Q.2) Explain in detail about various asymptomatic notations for analyzing algorithm.

Ans.2)

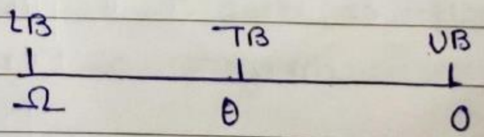
Asymptotic Notations.

- Asymptotic notations are used to find out three phase complexity of a function.
- The asymptotic running time of an algo. is defined in terms of function.
- This functions are set of natural no. and real no.
- A way of comparing functions that ignores constant factors and small input size is known as asymptotic notation.
- These are three types of asymptotic notation.

1) Theta Notation (Θ)
2) Omega Notation (Ω)
3) Big Oh Notation (O)

- The complexity of any algo. is based upon the following ~~size~~ three classes.

i) Best case.
ii) Worst case.
iii) Average case.



1) Theta Notation (Θ)

→ • This notation is also called as tight bound.

• The function $f(n) = \Theta(g(n))$ if there exists positive constants c_1, c_2 and n_0 such that

• The function $f(n)$ is sandwitched betn $c_1 g(n)$ and $c_2 g(n)$ since $\Theta(g(n))$ is a set and $f(n) \in \Theta(g(n))$ we can write:

$\boxed{f(n) = \Theta(g(n))}$ $\forall$ $n > n_0$

2) Omega Notation (Ω)

→ • This notation is called as lower bound

• The function $f(n) = \Omega(g(n))$ is called as lower bound of 'g' if there exists a positive constant c and n_0 such that

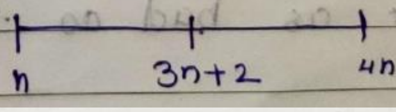
$\boxed{f(n) \geq g(n) * c}$ for $n \geq n_0$

• To calculate lower bound the value closet to f(n) should be selected. This is possible by ignoring the time for singular instruction involved in f(n)

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

Here, $f(n) = 3n+2$

$g(n) = 3n$



3) Big-oh Notation

→ The function $f(n) = O(g(n))$ is called as big oh g if there exists a positive constance 'c' and n_0 such that

~~fn~~ → $\boxed{fn <= g(n) * c}$

Q.3) Explain in detail Strassen's Matrix Multiplication.

Ans.3)

Strassen's matrix multiplication algorithm is a divide-and-conquer algorithm for multiplying two matrices. It was first published by Volker Strassen in 1969. Strassen's algorithm is based on the observation that matrix multiplication can be broken down into smaller subproblems. The algorithm recursively solves these subproblems and then combines the solutions to obtain the final result.

The basic idea of Strassen's algorithm is to divide the two matrices to be multiplied into four smaller matrices of half the size. These smaller matrices are then multiplied using a combination of seven basic operations: addition, subtraction, multiplication, and division. The results of these multiplications are then combined to form the product of the original two matrices.

Strassen's algorithm has a time complexity of O(n log 7), which is asymptotically faster than the naive matrix multiplication algorithm, which has a time complexity of O(n^3). However, Strassen's algorithm is more complex to implement and may not be faster in practice for small matrices.

Here is a more detailed explanation of Strassen's algorithm:

1. **The two matrices to be multiplied are divided into four smaller matrices of half the size.**
2. **The four smaller matrices are multiplied using the seven basic operations.**
3. **The results of the multiplications are combined to form the product of the original two matrices.**

The following table shows the time complexity of Strassen's algorithm for different matrix sizes:

Code snippet

Matrix size	Time complexity
2x2	O(1)
3x3	O(2)
4x4	O(7)
5x5	O(49)
6x6	O(343)

As you can see, the time complexity of Strassen's algorithm grows exponentially with the size of the matrices. This means that Strassen's algorithm is only practical for multiplying very small matrices.

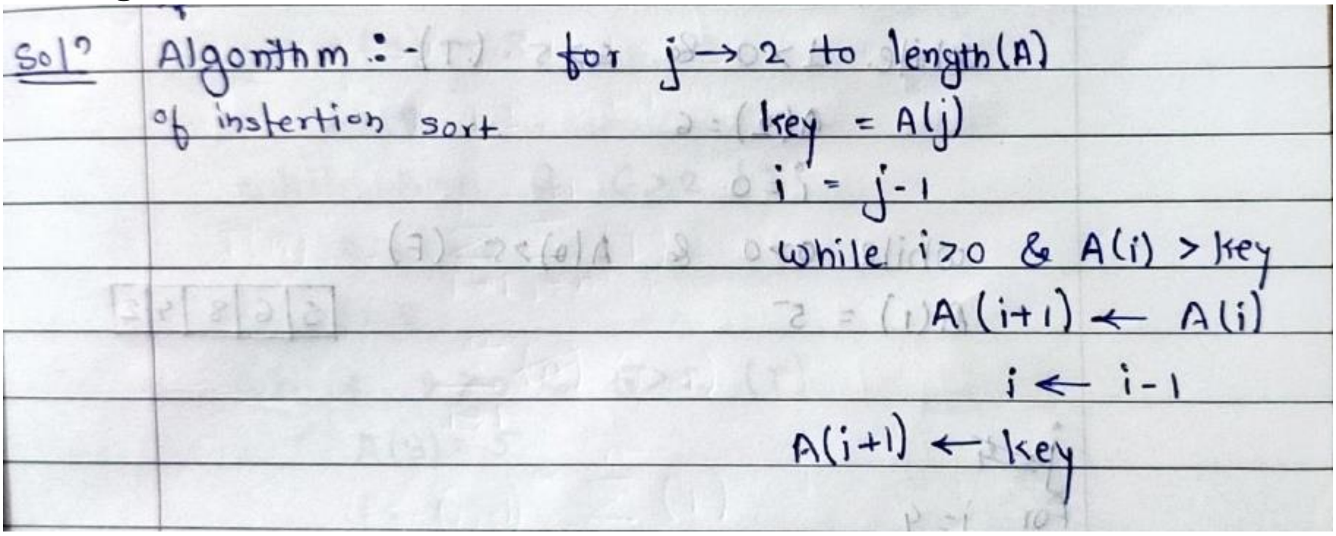
In practice, the naive matrix multiplication algorithm is often faster than Strassen's algorithm for multiplying large matrices. This is because the naive algorithm is simpler to implement and can be optimized for specific hardware platforms.

However, Strassen's algorithm can be useful for multiplying matrices of intermediate size. In these cases, the speedup provided by Strassen's algorithm can outweigh the increased complexity of implementation.

Q.4) Write the algorithm for Insertion Sort.

Ans.4)

The Algorithm for Insertion Sort:


Sol^n Algorithm :- of insertion sort

for $j \rightarrow 2$ to length(A)
key = A(j)
i = j-1
while i>0 & A(i) > key
A(i+1) ← A(i)
i ← i-1
A(i+1) ← key

Here is a breakdown of the algorithm:

1. The algorithm starts at the second element of the array.
2. The algorithm takes the current element and compares it to the elements before it.
3. If the current element is smaller than any of the elements before it, the algorithm swaps the current element with the element that is smaller than it.
4. The algorithm then repeats steps 2 and 3 for the next element in the array.
5. The algorithm continues this process until it has reached the end of the array.

Insertion sort is a simple sorting algorithm that is easy to understand and implement. However, it is not very efficient for large arrays. For large arrays, other sorting algorithms, such as quicksort or merge sort, are more efficient.

Here are some of the advantages of insertion sort:

- It is a simple algorithm to understand and implement.
- It is a stable algorithm, which means that the relative order of elements with equal keys is preserved.
- It is in-place, which means that it does not require any additional memory.

Here are some of the disadvantages of insertion sort:

- It is not very efficient for large arrays.
- It is not a recursive algorithm, which means that it cannot be easily parallelized.
- It is not very efficient for arrays that are already sorted.

Q.5) Derive worst case and best case run time for complexity for Insertion Sort.

Ans.5)

Page No.

Date

18/09/22

★ Find best & worst case ~~of~~ complexity running time of insertion sort.

⇒

```
For j → 2 to length(A)
    key = A(j)
    i = j-1
    while i>0 and A(i) > key
        A(i+1) = A(i)
        i = i-1
        ~~A(i+1) = key~~
    A(i+1) = key
```

	cost	No. of times
for	c_1	n
key	c_2	$n-1$
i	c_3	$n-1$
while	c_4	$\sum_{j=2}^{n} t_j$
A(i+1)	c_5	$\sum_{j=2}^{n} t_j - 1$
i	c_6	$\sum_{j=2}^{n} t_j - 1$
A(i)	c_7	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} t_j - 1 + c_6 \sum_{j=2}^{n} t_j - 1 + c_7(n-1)$$

for best case :-

If it is already in ascending order.

$t_j = 1$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n}(1) \cancel{+ c_5 \sum_{j=2}^{n}(1)} + c_7(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

$$= c_1 n + c_2 n - c_2 + c_3 n - c_n + c_4 n - c_4 + c_7 n - c_7$$

$$= c_1 n + c_2 n + c_3 n + c_4 n + c_7 n - c_2 - c_3 - c_4 - c_7$$

~~c_6~~ c -

Page No.
Date

$$= (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_3 - c_4 - c_7)$$
$$\quad\quad (a) \quad\quad\quad\quad\quad\quad\quad\quad (b)$$

$T(n) = \Theta(n)$
$T(n) = \Omega(n)$

For worst case :

If it is ~~de~~ ~~not~~ in descending order.
$\therefore t_j = j$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n}(j) + c_5 \sum_{j=2}^{n}(j-1) + c_6 \sum_{j=2}^{n}(j-1) + c_7(n-1)$$

$$= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n+1)}{2} - 1\right)$$

$$- c_5(n-1) + \left(c_6 \frac{n(n+1)}{2} - 1\right) - c_6(n-1) + c_7(n-1)$$

$$= c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4\left(\frac{n^2+n}{2} - 1\right) + c_5\left(\frac{n^2+n}{2} - 1\right)$$

$$- \cancel{c_5 n} - c_5 n + c_5 + c_6\left(\frac{n^2+n}{2} - 1\right) - c_6 n + c_6$$

$$+ c_7 n - c_7$$

$$= c_1 n + c_2 n - c_2 + c_3 n - c_3 + \frac{c_4 n^2}{2} + \frac{c_4 n}{2} - c_4 + \frac{c_5 n^2}{2}$$

$$+ \frac{c_5 n}{2} - \cancel{c_5} - c_5 n + \cancel{c_5} + \frac{c_6 n^2}{2} + \frac{c_6 n}{2} - \cancel{c_6}$$

$$- c_6 n + \cancel{c_6} + c_7 n - c_7$$

$$= \frac{c_4 n^2}{2} + \frac{c_5 n^2}{2} + \frac{c_6 n^2}{2} + c_1 n + c_2 n + c_3 n$$

$$+ \frac{c_4 n}{2} + \frac{c_5 n}{2} - c_5 n + \frac{c_6 n}{2} - c_6 n + c_7 n$$

$$\cancel{+} - c_2 - c_3 - c_4 - c_7$$

$$= n^2\left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) + n\left(c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} - c_5\right.$$

$$\left. + \frac{c_6}{2} - c_6 + c_7\right) + \left(-c_2 - c_3 - c_4 - c_7\right)$$

$$= \quad an^2 + bn + c \qquad \text{(quadratic func}^c \text{ of } n)$$

$T(n) = \theta(n^2)$

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

Q.7) State Dijkstra's algorithm for solving Single Source multiple destination shortest path problems? Illustrate your answer by stepwise execution by assuming suitable example.

Ans.7)

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a weighted graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later. The algorithm exists in many variants.

Here is the pseudocode for Dijkstra's algorithm:

```
function dijkstra(graph, source):
    distances = {}
    for v in graph.nodes():
        distances[v] = float("inf")
    distances[source] = 0
    unvisited = set(graph.nodes())
    while unvisited:
        u = min(unvisited, key=distances.get)
        unvisited.remove(u)
        for v in graph.neighbors(u):
            new_distance = distances[u] + graph.edge(u, v).weight
            if new_distance < distances[v]:
                distances[v] = new_distance
    return distances
```

Here is an example of how to use Dijkstra's algorithm to find the shortest paths from node A to all other nodes in a graph:

```
graph = {
    "A": {"B": 5, "C": 10},
    "B": {"A": 5, "C": 2},
    "C": {"A": 10, "B": 2}
}
distances = dijkstra(graph, "A")
print(distances)
```

This will print the following output:

```
{'A': 0, 'B': 5, 'C': 10}
```

This means that the shortest path from node A to node A is 0, the shortest path from node A to node B is 5, and the shortest path from node A to node C is 10.

Dijkstra's algorithm is a greedy algorithm. This means that it always chooses the node that is closest to the source node. This can be inefficient for graphs with negative weights, as Dijkstra's algorithm may end up choosing a path that is not the shortest path.

Here is a table that summarizes the time complexity of Dijkstra's algorithm for different input conditions:

Input condition	Time complexity
Unweighted graph	O(
Weighted graph	O(