Sorting Algorithm - Contd. from Day3
====================================

Quick Sort
==========

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.

A large array is partitioned into two arrays (similar to Merge sort). Partitioning of the array is NOT based on the number of elements but rather based on a value which is called as pivot.

Based on this pivot value, one array contains elements which are smaller than pivot value and another contains elements with greater than pivot value. Thus, the division of array into two may not result into 2 equal size arrays.

There are many different versions of quickSort that pick pivot in different ways.

    Picks first element as pivot,
    Picks last element as pivot (we follow this),
    Picks a random element as pivot,
    Picks median as pivot.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of O(n^2), where n is the number of items.

Quick Sort algorithm

```
low  --> Starting index,
high  --> Ending index
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Partition Algorithm
----

* This function takes last element as a pivot.
* Places the pivot element at its correct position in sorted array.
* Places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot

```
partition (arr[], low, high)
{
    pivot = arr[high];
```

```
    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high - 1; j++)
    {
        // If current element is smaller than or equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)  // Index where the pivot is placed in its proper slot
}

Illustration of partition method::

arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6

low = 0, high =  6, pivot = arr[high] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

j = 1 : Since arr[j] > pivot, do nothing // No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swapped 80 and 30

j = 3 : Since arr[j] > pivot, do nothing // No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is greater than high-1.

Finally we place pivot at correct position by swapping arr[i+1] and arr[high]
(or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than 70 are before it and
all elements greater than 70 are after it.

=====================================================================

Selection Sort
==============

Similar to Bubble sort wherein on each iteration the next largest elements move
to its correct position, in selection sort we first find an element which needs
to be moved first without moving any other elements.
```

Therefore, it is more efficient than Bubble Sort.

Algorithm for selection sort
==========

array a, size n is given.
min is an index variable pointing to the array location which has the minimum value.

```
for i=0; i< n-1; i++ {
   min=i
   for j = i+1; j < n, j++ {
     if (a[j] < a(min)) then min = j;

   swap(a[min], a[i])

   }
}
```

E.g.,
a[] = {10, 9, 12, 3, 20, 2}
n = 6

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
10	9	12	3	20	2

i=0
min=0

min=5
swap(a[5],a[0])

2	9	12	3	20	10

i=1
min=2 3 9

i=2
min=3 9 12

i=3
min=4 10 12

i=4
min=5 12 20

i=5 Loop breaks. array is sorted.

Insertion sort
==========

Playing cards game example as analogy to this algorithm.

Bubble sort and selection sort, quick sort makes use of swap.
Insertion sort does not swap elements.

Algorithm
==========

```
array a, size n is given.

for (i=1; i < n; i++) {
  temp = a[i];

  for (j=i; j > 0 && a[j-1] > temp; j--)
    a[j] = a[j-1];

  a[j] = temp;
}

E.g.,
      a[] = {10, 9, 12, 3, 20, 2}



                  a[0]  a[1]  a[2]  a[3]  a[4]  a[5]
                  10    9     12    3     20    2

i=1
temp=9
j=i=1
a[0] is 10 & > 9
a[1]=a[0]
                  10
j=0
a[0]=9
            9

i=2
temp=12
j=i=2
a[j-1]=a[1] which is 10
is not greater than temp which is 12
loop breaks
a[2]=12

                  a[0]  a[1]  a[2]  a[3]  a[4]  a[5]
                  9     10    12    3     20    2
i=3
temp=a[3] which is 3
j=i=3
a[j-1] i.e. a[2] i.e. 12 > temp
a[3] is set to a[2]                12
j--. now j=2
a[j-1], i.e. a[1], i.e.,10
is greater than temp
a[j] i.e. a[2] is set to a[1]
                        10
```

==========

An Algorithm is a sequence of steps to solve a problem. Design and Analysis of Algorithm is very important to solve different types of problems in the branch of computer science and information technology.

Basic knowledge of programming and mathematics is essential for designing algorithms.

Algorithm

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks.

An algorithm needs to be an efficient method that can be expressed within finite amount of resources (time and space).

An algorithm needs to be simple.

Algorithm is independent of programming language.

Algorithm Design

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.

To solve a problem, different approaches can lead to different algorithms.

Some of the algorithms may be efficient with respect to time consumption, whereas other approaches may be more memory efficient.

Keep in mind that both time consumption and memory usage cannot be optimized simultaneously.

Bigger picture - Computational Problem solving

The following steps are involved in solving computational problems.

1) Problem definition
2) Development of a model
3) Specification of an Algorithm
4) Designing an Algorithm
5) Checking the correctness of an Algorithm
6) Analysis of an Algorithm
7) Implementation of an Algorithm
8) Program testing
9) Documentation

Characteristics of Algorithms

The main characteristics of algorithms are as follows −

Algorithms must have a unique name

Algorithms should have explicitly defined set of inputs and outputs

Algorithms are well-ordered with unambiguous operations

Algorithms halt(complete in terms of desired steps) in a finite amount of time.

Pseudo-code

Pseudo-code gives a high-level description of an algorithm without the ambiguity associated with plain text but also without the need to know the syntax of a

particular programming language.

The running time can be estimated in a more general manner by using Pseudo code to represent the algorithm as a set of fundamental operations which can then be counted.

Difference between Algorithm and Pseudo-code

An algorithm is a formal definition with some specific characteristics that describes a process, which could be executed by a Turing-complete computer machine to perform a specific task.
Generally, the word "algorithm" can be used to describe any high level task in computer science.

Pseudo-code is an informal and (often rudimentary) human readable description of an algorithm leaving many granular details of it.

Writing a pseudo-code has no restriction of styles and its only objective is to describe the high level steps of algorithm in a much realistic manner in natural language.

For example, following is an algorithm for Insertion Sort.

```
Algorithm: Insertion-Sort
Input: A list L of integers of length n
Output: A sorted list L1 containing those integers present in L
Step 1: Keep a sorted list L1 which starts off empty
Step 2: Perform Step 3 for each element in the original list L
Step 3: Insert it into the correct position in the sorted list L1.
Step 4: Return the sorted list
Step 5: Stop
```

Here is a pseudo-code which describes how the high level abstract process mentioned above in the algorithm Insertion-Sort could be described in a more realistic way.

```
for i <- 1 to length(A)
   x <- A[i]
   j <- i
   while j > 0 and A[j-1] > x
      A[j] <- A[j-1]
      j <- j - 1
   A[j] <- x
```

Algorithm Analysis - What is it?

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem.

Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as
      time complexity, or

volume of memory, known as space complexity.

The main concern of analysis of algorithms is the required time or performance

Generally, following types of analysis are performed −

Worst-case − The maximum number of steps taken on any instance of size s.

Best-case − The minimum number of steps taken on any instance of size s.

Average case − An average number of steps taken on any instance of size s. (sampling)

Amortized − A sequence of operations applied to the input of size s averaged over time.
Explain amortization.

The Need for Analysis - Why Analyze?

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa.

In this context, if we compare bubble sort and merge sort. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

Different Methods

Asymptotic Analysis

The asymptotic behavior of a function f(n) refers to the growth of f(n) as n gets large.

We typically ignore small values of n, since we are usually interested in estimating how slow the program will be on large inputs.

A good rule of thumb is that the slower the asymptotic growth rate, the better the algorithm.

For example, a linear algorithm f(n) = d*n+ k is always asymptotically better than a quadratic one, f(n) = c*n*n +q.

Solving Recurrence Equations

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrences are generally used in divide-and-

conquer paradigm.

Recurrence relation can be solved using following methods.

Substitution Method − In this method, we guess a bound and using mathematical induction we prove that our assumption was correct.

Recursion Tree Method − In this method, a recurrence tree is formed where each node represents the cost.

Master’s Theorem − This is another important technique to find the complexity of a recurrence relation.

Lab Assignment::
==========

Write a program to implement quick sort, selection sort and insertion sort. Create an array of size 10,000 integers and measure performance of all the three sorts in terms of elapsed time on the same set(s) data.