DATA STRUCTURE

- The word algorism, came from the 9th century Persian mathematician "Abu Abdullah Muhammad bin Musa **al-Khwarizmi**" , which means the method of doing arithmetic using Indo-Arabic decimal system. It is also the root of the word "algorithm" .

- An algorithm is a well defined computational method that takes some value(s) as input and produce some value(s) as output. In other words, an algorithm is a sequence of computational steps that transforms input(s) into output(s).

- An algorithm is correct if for every input, it halts with correct output. A correct algorithm solves the given problem, where as an incorrect algorithm might not halt at all on some input instance, or it might halt with other than designed answer.

- Each algorithm must have

  - Specification: Description of the computational procedure.
  - Pre-conditions: The condition(s) on input.
  - Body of the Algorithm: A sequence of clear and unambiguous instructions.
  - Post-conditions: The condition(s) on output.

- Algorithms are written in pseudo code that resembles programming languages like C and Pascal.
- Consider a simple algorithm for finding the factorial of ***n***.

```
 Algorithm Factorial (n)

Step 1:     FACT = 1
Step 2:     for i = 1 to n do
Step 3:     FACT = FACT * i
Step 4:     print FACT
```

Specification: Computes *n*!.

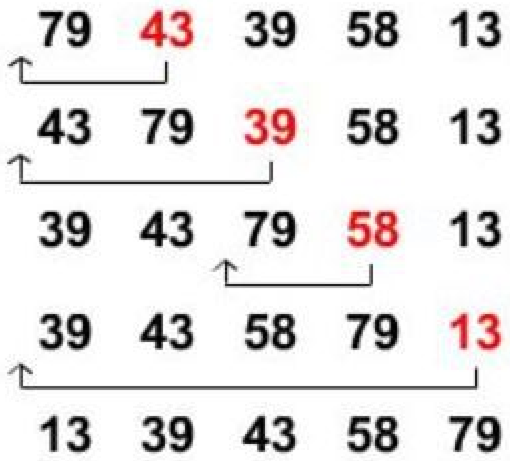
Pre-condition: *n* >= 0

Post-condition: FACT = *n*!

- For better understanding conditions can also be defined after any statement, to specify values in particular variables.
- Pre-condition and post-condition can also be defined for loop, to define conditions satisfied before starting and after completion of loop respectively.
- What is remain true before execution of the i^{th} iteration of a loop is called "loop invariant".
- These conditions are useful during debugging proces of algorithms implementation.
- Moreover, these conditions can also be used for giving correctness proof.

- Now, we take a more complex problem called sorting.
- Problem Definition: Sort given *n* numbers by non-descending order.
- There are many sorting algorithm. Insertion sort is a simple algorithm.
- Insertion Sort: We can assume up to first number is sorted. Then sort up to two numbers. Next, sort up to three numbers. This process continue till we sort all *n* numbers.
- Consider the following example of five integer:

  79 43 39 58 13 : Up to first number, 79, is sorted.
  43 79 39 58 13 : Sorted up to two numbers.
  39 43 79 58 13 : Sorted up to three numbers.
  39 43 58 79 13 : Sorted up to four numbers.
  13 39 43 58 79 : Sorted all numbers.
- That is, if first (i-1) numbers are sorted then insert ith number into its correct position. This can be done by shifting numbers right one number at a time till a position for ith number is found.
- That is, shift number at (i-1)th position to ith position, number in (i-2)th position to (i-1)th position, and so on, till we find a correct position for the number in ith position. This method is depicted in the figure on right side.



An Applet Demonstrating Insertion Sort

- The algorithmic description of insertion sort is given below.

```
        Algorithm Insertion_Sort (a[n])
Step 1: for i = 2 to n do
Step 2: current_num = a[i]
Step 3: j = i
Step 4: while (( j >1) and (a[j-1] > current_num)) do
Step 5:     a[j] = a[j-1]
Step 6:     j = j-1
Step 7: a[j] = current_num
```

- More about sorting algorithms are discussed in the modules 5 and 10.

- Execution time of an algorithm depends on numbers of instruction executed.
- Consider the following algorithm fragment:

```
for i = 1 to n do

  sum = sum + i ;
```

- The for loop executed $n+1$ times for i values 1,2,……. n, $n+1$. Each instruction in the body of the loop is executed once for each value of i = 1,2,……, n. So number of steps executed is $2n+1$.
- Consider another algorithm fragment:

```
for i = 1 to n do
  for j = 1 to n do
            k = k +1
```

- From prevoius example, number of instruction executed in the inner loop is $2n+1$, which is the body of outer loop.
- Total number of instruction executed is

$$= n + 1 + n(2n + 1)$$

$$= 2n^2 + 2n + 1$$

- To measure the time complexity in absolute time unit has the following problems

  1. The time required for an algorithm depends on number of instructions executed, which is a complex polynomial.
  2. The execution time of an instruction depends on computer's power. Since, different computers take different amount of time for the same instruction.
  3. Different types of instructions take different amount of time on same computer.

- Complexity analysis technique abstracts away these machine dependent factors . In this approach, we assume all instruction takes constant amount of time for execution.
- Asymptotic bounds as polynomials are used as a measure of the estimation of the number of instructions to be executed by the algorithm . Three main types of asymptotic order notations are used in practice:

1. Θ - ***notation*** : For a given function g(n), $\theta(g(n))$ is defined as

$$\theta(g(n)) = \left\{ \begin{array}{l} f(n): \text{ there exist } c_1 > 0, c_2 > 0 \text{ and } n_0 \in N \\ \qquad \text{such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \text{ for all } n \ge n_0 \end{array} \right\}$$

In other words, a function $f(n)$ is said to belong to $\theta(g(n))$, if there exists positive constants c_1 and c_2 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for sufficiently large value of n. For example, $n(n-1) \in O(n^2)$. This is because we can find constants $c_1 = \frac{1}{2}$ and $c_2 = 1$ such that $\frac{1}{2}n^2 \le \frac{n(n-1)}{2} \le n^2$, for all $n \ge 2$.

2. O - ***notation*** : For a given function $g(n), O(g(n))$ is defined as

$$O(g(n)) = \left\{ \begin{array}{l} f(n) : \textit{there exists } c > 0, \textit{ and } n_0 \in N \\ \qquad \textit{such that } 0 \leq f(n) \leq cg(n), \textit{ for all } n \geq n_0 \end{array} \right\}$$

For example, $2n^2 + 5n + 6 \in O(n^2)$ as $2n^2 + 5n + 6 \leq 6(n^2)$ for all $n \geq 2$

3. Ω - ***notation:*** This notation provides asymptotic lower bound. For a given $g(n)$, $\Omega(g(n))$ is defined as

$$\Omega(g(n)) = \left\{ \begin{array}{l} f(n) : \textit{there exists } c > 0, \textit{ and } n_0 \in N \\ \qquad \textit{such that } 0 \leq cg(n) \leq f(n), \textit{ for all } n \geq n_0 \end{array} \right\}$$

For example, $n^3 \in \Omega(n^2)$ because $n^3 \geq n^2$, for all $n \geq 0$

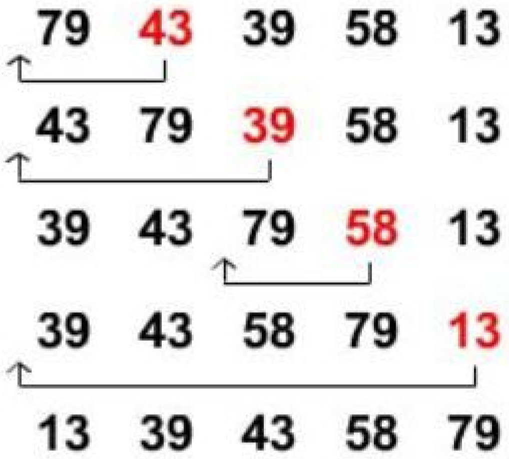
- Step 1: The for loop executed *n* times, for *i* values from 2, 3, ...*n*+1
- Step 2, 3, and 7: Each executed once for each iteration of the for loop. That is, each *n*-1 times.
- Step 4: The while loop is executed at most *i* times, for *i* = 2, 3, ..., *n*. That is, $n \times (n+1)/2-1$.
- step 5 and 6: Each instruction executed *i*-1 times, for each *i*. Hence, $(n-1) \times n / 2$ for each instruction.
- Total number of instruction executed in the worst case is n^2+4n-4. Hence the time complexity of the algorithm is in $\theta(n^2)$.

Similarly, we can analyze space required for the algorithm also. The Insertion sort algorithm has to store all n inputs and using three more variables. So, the space complexity of the algorithm is in $\theta(n)$.

- Another way of finding number of instruction executed is by recursive equation. Let *T*(*n*) be the time required to sort *n* numbers. *T*(*n*) can be expressed as a sum of *T*(*n*-1) and the time required to insert *n*th element in the sorted array of n-1 element.

- The time required to insert an element in sorted array of *n*-1 elements takes *cn* steps, where *c* is a positive constant. This is because to insert *n*th element, in worst case, we have to shift all *n*-1 elements one after other.

See the following figure.



- Hence, the recurrence relation for Insertion sort is

$$T(n) = T(n-1) + cn, \text{ if } n > 1$$

$$= 1 \text{ if } n = 1$$

- There are three main approach to solve recurrence relations. They are substitution, iterative and master theorem methods. In this notes we discuss iterative approach only. For other methods see Algorithms by Cormen et al.

- Iterative Method:

$$T(n) = T(n-1) + cn$$

$$= T(n-2) + c(n-1) + cn$$
$$= T(n-3) + c(n-2) + c(n-1) + cn$$
$$.$$
$$.$$

$= T(1) + 2c + 3c + \ldots\ldots\ldots\ldots + c(n-2) + c(n-1) + cn$

$= 1 + 2c + 3c + \ldots\ldots\ldots\ldots + c(n-2) + c(n-1) + cn$

$< c(1 + 2 + 3 + \ldots\ldots\ldots\ldots + n-2 + n-1 + n)$

$= c\ (n\ (n+1)/2)$

$= c\ n^2/2 + c\ n/2$

$= O(n^2)$

- Consider another recurrence:

$$T(n) \begin{cases} = O(1) & \text{if } n = 1 \\ = 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

- In the above recurrence relation O(1) means a constant. So we can replace with some constant c_1.
- Similarly,O(n) means a function of order n. So we can replace with C_2n.Hence, the recurrence can be rewritten as

$$T(n) \begin{cases} <= C_1 \text{ if } n=1 \\ <= 2T(n/2) + C_2n \text{ if } n > 1 \end{cases}$$

- Solution by Iterative method:

$$\begin{aligned} T(n) &<= 2T(n/2) + C_2n \\ &<= 2(\ 2T(n/4) + C_2n/2) + C_2n \\ &<= 2^2T(n/2^2) + C_2n + C_2n \\ &<= 2^2(2\ T(n/2^3) + C_2n/2^2) + C_2n + C_2n \\ &<= 2^3T(n/2^3) + C_2n + C_2n + C_2n \end{aligned}$$

.

.

.

$<= 2^i T(n/2^i) + C_2 n i$

- Assume $n = 2^i$ for some value of i.That is, i = log n

$$T(n) = nT(1) + C_2 n*\log(n)$$

$$= C_1 n + C_2 n*\log(n)$$

$$= O(n*\log(n))$$

- If $n \neq 2^i$ for some i, consider $i = \lceil(\log n)\rceil$

$$T(n) < 2nT(1) + c_2 n*\log(n)$$

$$= 2c_1 n + c_2 n*\log(n)$$

$$= O(n*\log(n))$$

1. Given an array of n integers, write an algorithm to find the smallest element. Find number of instruction executed by your algorithm. What are the time and space complexities?
2. Write a algorithm to find the median of n numbers. Find number of instruction executed by your algorithm. What are the time and space complexities?
3. Write a C program for the problem 1.
4. Write a C program for the problem 2.
5. Solve the following recurrence relations. Assume $T(1) = O(1)$

   1. $T(n) = T(n-2) + cn$
   2. $T(n) = 2T(n/2) + c\, n$^2
   3. $T(n) = 2T(n/2) + c\, n$^3
   4. $T(n) = 4\, T(n/4) + O(n)$
   5. $T(n) = 4\, T(n/2) + c\, n$
   6. $T(n) = T(n/2) + O(1)$
   7. $T(n) = 2T(n/2) + n \log n$
   8. $T(n) = T(n-1) + 1/n$
   9. $T(n) = T(n/2) + \log n$
   10. $T(n) = T(\sqrt{n}) + 1$