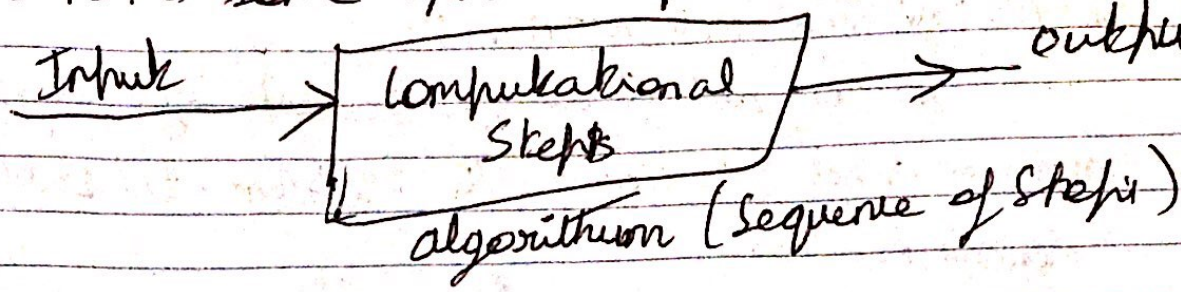
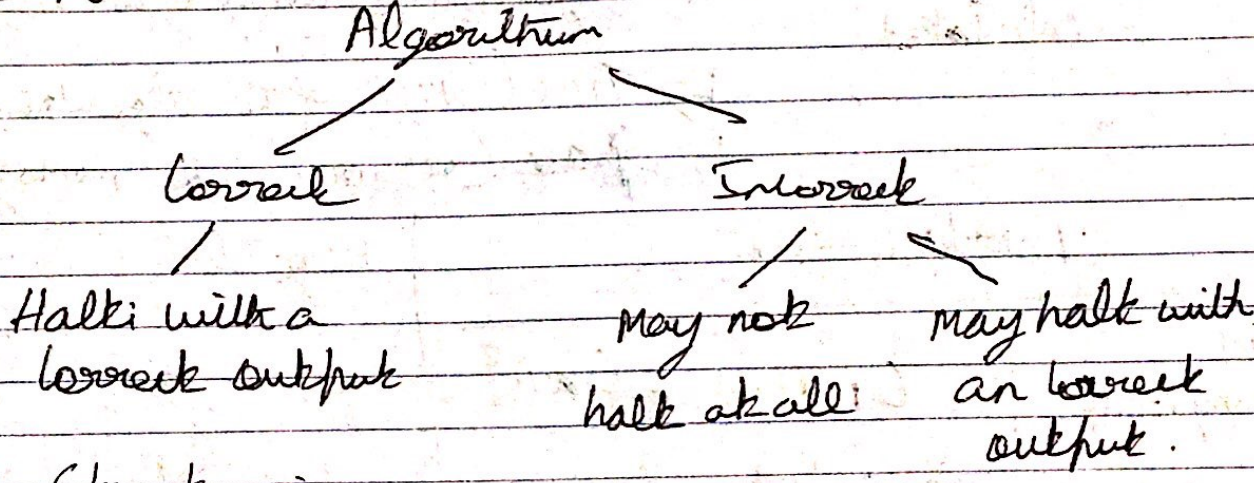
Chapter - I

ALGORITHUMS :

An algorithum is a computational procedure that takes some input and produces output



↳ Can be used as a tool to Solve Computational Problem



DATA Structure :

↳ It is a way to store the data in a organized way.

↳ The data is organized in such a way that retrieval and modifications are done in a much faster and efficient way.

↳ Based on the Problem a data structure Technique is selected.

In Most of the algorithums efficiency is measured in Speed.

no effective algorithum found → NP-complete Problems.

APPROXIMATION ALGORITHUMS:-

It does not Produce the best efficiency but ensures a good Solution for the Problem. Ex: Traveling Salesman Problem.

Efficiency of Algorithums

Based on two factors efficiency of an Algorithum are calculated.

↳ Time (Speed of the algorithum)
↳ Space (resources like memory).

	Insertion Sort	Merge Sort.
↳ Time complexity	$C_1 \cdot n \cdot n = C_1 n^2$	$C_2 n \cdot \log n$
↳ Speed for Small input Size.	It is faster for Smaller input Size	It is slower than Insertion Sort for Smaller Input Size.
↳ Speed for large Input Size.	It is Slow	It is fast because it takes advantage of logn
Example:	when $n = 1000$ $n^2 = 10,00,000$	when $n = 1000$ $n \log n$ $1000 \times \log_2 1000$ 1000×10 (approx) $= 10000$

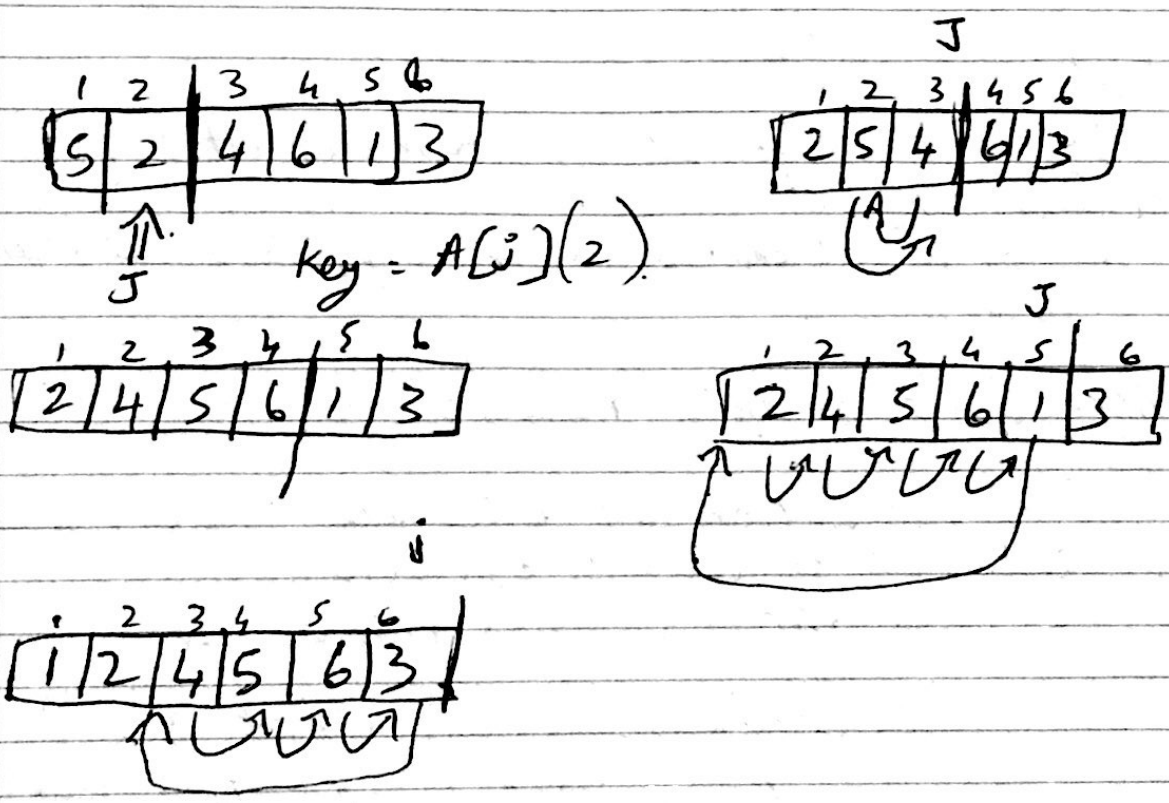
CHAPTER - 2

INSERTION SORT VS MERGE SORT.

INSERTION SORT :

Good algorithm for sorting a small number of elements.

```
                                          Cost    times
for J ← 2 to n                            C1      n
    do key ← A[j]                         C2      n-1
                                          C3      n-1
       i ← j-1                            C4      n-1
       while i>0 and A[i] > key           C5      n        } tj
           do A[i+1] ← A[i]               C6      Σ        } tj-1
              i ← i-1                     C7      j=2      } tj-1
       A[i+1] ← key                       C8      n-1
```



Invariant loop for Insertion Sort:

Initialization: Before the first Iteration, J = 2. The Sub array A[1... J-1] is the Single element A[1].

Maintenance: To be precise, we need to understand that the inner loop works by moving [J-1][J-2]... and so on, by one position to the Right until the Proper position is found. And then the key is placed into this Position

Termination: when J > n, it is terminated and this happens when J = n+1;

Analysis Algorithm:

↳ It means predicting the resources that the algorithm requires.

↳ Computation time is the only thing we want to measure

RAM - Random Access Machine:

↳ Instructions are executed one after the other with no concurrent operation.

↳ RAM MODEL Instructions are:

- arithmetic (+, -, /, *)
- data movement (load, store, copy)
- control (conditional and unconditional Branch. Subroutine call and return)

Analysis of Insertion Sort

Time taken α Input array.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Best case.

It occurs if the array is already sorted.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1).$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Expressed as $an + b$ for constants a & b depend on cost c_j.

Worst case

If the array is reversely sorted.

have to compare each of $A[j]$ to the sub array $A[1 \ldots j-1]$.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left[\frac{n(n+1)}{2} - 1\right] + c_6\left[\frac{n(n-1)}{2}\right] + c_7\left[\frac{n(n-1)}{2}\right] + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

which is expressed as an^2+bn+c

constants a, b & c

↓ quadratic functions.

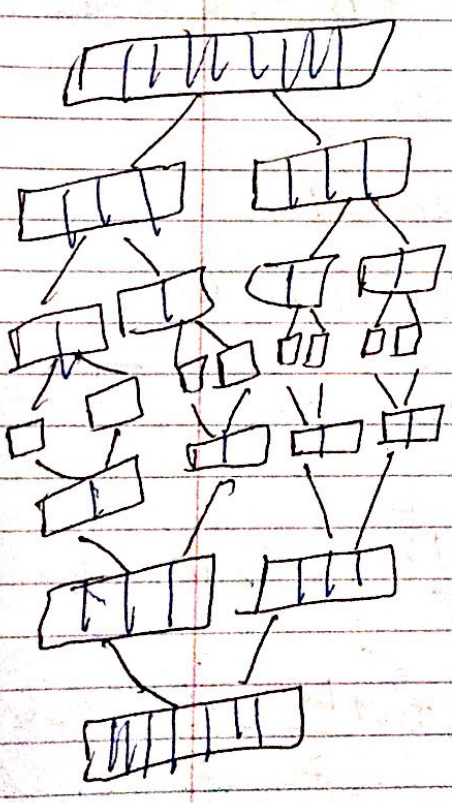
Order of growth or Rate of growth

Ignore the constant term they are less significant than rate of growth in determining computational efficiency for large input.

for Best case $\Theta(n)$
Average case $\Theta(n^2)$
Worst case $\Theta(n^2)$

Merge Sort.

↳ $\Theta(n \log n)$ in worst case.



↳ Recursive call
↳ Divide and conquer algorithm
↳ Not In-Place
↳ Stable

```
Merge (L, R, A)
{
 nL ← length(L)
 nR ← length(R)
 i ← j ← k ← 0
while (i < nL && j < nR)
 {if (L[i] <= R[j])
 { A[k] ← L[i]
   i ← i+1
 } else {
   A[k] ← R[j]
   j ← j+1
 }
 k ← k+1
}
```

```
while (i < nL) {
A[k] ← L[i]; i ← i+1; k ← k+1;
}
while (j < nR) {
A[k] ← R[j]; j ← j+1;
                k ← k+1;
}
```

```
Merge Sort(A) {
    n ← length(A)                    } C1
    if (n < 2) return
    mid ← n/2
    left ← array of size (mid)
    right ← array of size (n-mid)
    for i ← 0 to mid-1               } C2·n
        left[i] ← A[i]
    for i ← mid to n-1
        right[i-mid] ← A[i]
    T(n/2) ⇐ merge sort (left)
    T(n/2) ⇐ merge sort (Right)
    C3·n + C4 ⇐ Merge (left, Right, A)
}
```

$$T(n) = \begin{cases} C, & \text{if } n = 1 \\ 2T(n/2) + C'n, & \text{if } n > 1 \end{cases}$$

$$T(n) = 4T(n/4) + 2C'n = 8T(n/8) + 3C'n = 16T(n/16) + 4C'n$$

$$= 2^k T\left(n/2^k\right) + k \cdot C' \cdot n \qquad \left[\frac{n}{2^k} = 1 \;\middle|\; \log_2 n = k\right]$$

$$= 2^{\log_2 n} T(1) + \log_2 n \cdot C' \cdot n$$

$$= n\,C + C' \cdot n \log n$$

Chapter - 3

Asymptotic notation:

Notation used to describe the asymptotic running time of an algorithm.

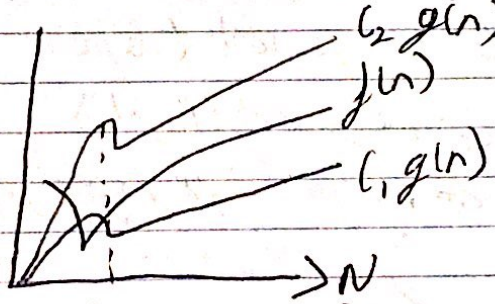
Defined in terms of functions (whose domains are the set of natural numbers $N = \{0, 1, 2, \ldots\}$)

eg Asymptotic notation $\Theta(n^2)$
(for the function $an^2 + bn + c$)

(1) Theta (Θ) notation

$\Theta(g(n))$ for a given function $g(n)$ we denote by $g(n)$ the set of functions.

$\Theta(g(n)) = \{ f(n) :$ there exist positive constants c_1, c_2 & n_0 such that
$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \quad \forall n \ge n_0 \}$



$f(n) = \Theta(g(n))$

- bounds a fn to within constant factors.
- for all the value of n to right of n_0, the value of $f(n)$ lies at above $c_1 g(n)$ or at below $c_2 g(n)$

$\forall n \ge n_0 \Rightarrow f(n) = g(n)$ (or) to within a constant factor.

② O notation - Asymptotic upperbound.

$O(g(n)) = \{ f(n)$ there exists +ve constants c & n_0 such that

$0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0$

$f(n) = O(g(n))$ means $f(n) = O(g(n))$

③ Ω notation Asymptotic lower bound.

$\Omega(g(n)) = f(n)$ · there exists constants c & n_0 such that $0 \leq g(n) \leq f(n) \quad \forall n \geq n_0$.