UNIT 1

20CO41

Space Complexity

Variable	byte
int	4
float	4
char	1
(return)	.
(depends)	

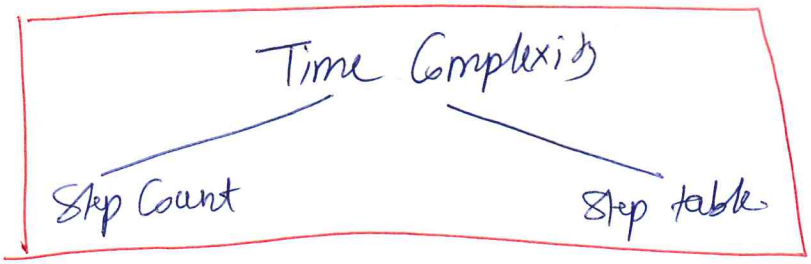
$S(P) = C + S_p$

- C → fixed space
- S_p → variable space.

Time Complexity

$T(P) = C + T_p(n)$

- C → Compile time
- $T_p(n)$ → Run time.



Note : Step Count

1. Algorithm heading → 0
2. Brackets/Braces { } → 0
3. Assignment Statement → 1
4. for any looping statement → No. of times the loop is repeating.

Master Theorem

$$T_n = \Theta(n^d) \quad \text{if } a < b^d$$

$$= \Theta(n^d \log_b n) \quad \text{if } a = b^d$$

$$= \Theta(n^{\log_b a}) \quad \text{if } a > b^d$$

$$T_n = aT\left(\frac{n}{b}\right) + f(n) \qquad d = \text{degree of } f(n)$$

Growth of ORDER

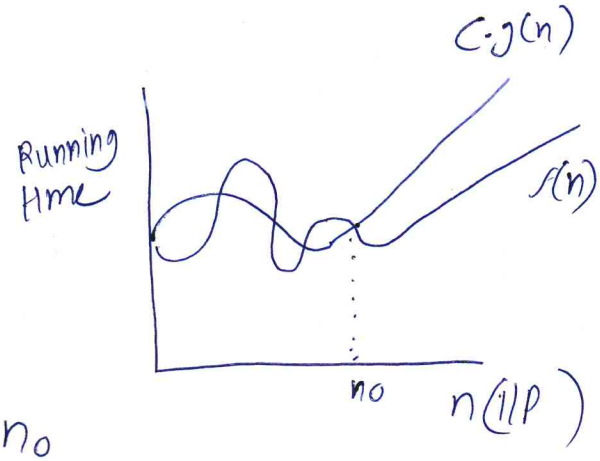
$$O(1) < O(\log N) < O(N) < O(N \log N) < O(N^2) < O(N^3) < O(2^N) < O(N!)$$

Asymptotic Notations

- **Big-Oh Notation (O)**
  - Upper bound
  - Worst Case

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

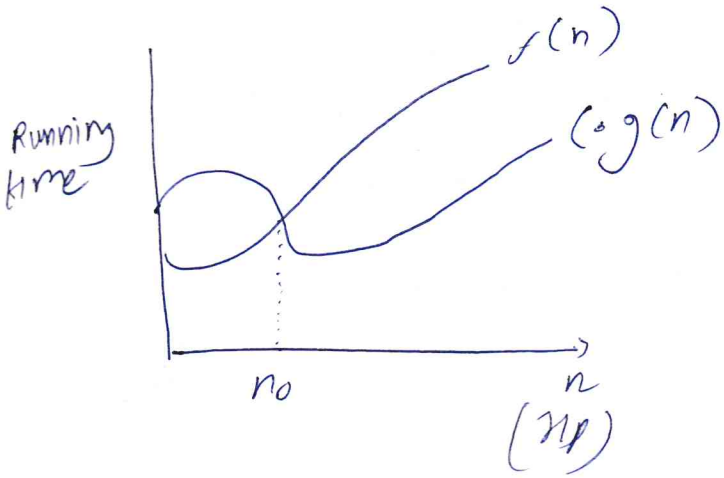
$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$



- **Big Omega (Ω)**
  - Best Case

$f(n) \geq c \cdot g(n)$

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

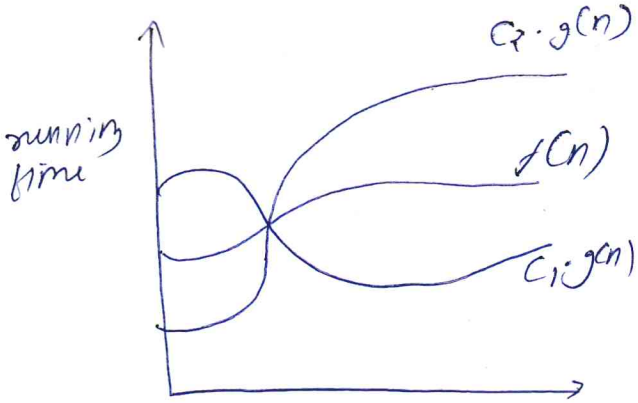


- **Theta Notation (Θ)**
  - average case

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

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Ω $\quad$ Θ



Little - Oh - notation (o)

$f(n) < c \cdot g(n)$ $\quad n_0 > 0 \quad c > 0$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

* g should satisfy for all values of c

$f(n) = o(g(n))$

Little Omega (ω)

$f(n) > c \cdot g(n)$

$f(n) = \omega \cdot g(n)$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Divide and Conquer

- Top down approach

```
Algorithm D&C(P)
{ if small(p) then
    return S(p);
  else
  {divide p into smaller subproblems P1 P2, P3 ... Pk  k >= 1
   apply D&C to each of the subproblems;
   return Combine(D&C(P1), D&C(P2), ....., D&C(Pk));
  }}
```

Small(P): is a boolean function that determines whether the input size is small enough that answer can be computed without split.

Combine(): is a function that determines the solution P using the solutions to K sub problems.

$$T(n) = \begin{cases} g(n) & ; n = 1 \text{ or } n \text{ is small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) + f(n) & ; \text{otherwise} \end{cases}$$

T(n): is a time for D and C on any input size n

g(n): is a time to compute any answer directly for small elements.

f(n): time for dividing P and combining solution to subproblem

Binary Search

- Elements in the array sorted order.
- mid = $\frac{low + high}{2}$

```
{ low = 0
  high = n-1
  mid = (low+high)/2
  if (high > low)
  mid = low +((high - low)/2)
  if A[mid] = value
    return mid
  Else if A[mid] > value
    return Binary_search(A, value, low, mid-1)
Else
  return Binary_search(A, value, mid+1, high)
Else
  return -1
}
```

Time Complexity = $\log_2 n$

$T\left(\frac{n}{2}\right) + 1$

Merge Sort

```
Algorithm MergeSort (A, low, high)
{ if (low < high)
 { mid = (low + high) / 2
  MergeSort (A, low, mid);
  MergeSort (A, mid+1, high);
  Merge (A, low, mid, high);
 }}

Algorithm Merge (A, low, mid, high)
{ i = low
  j = mid + 1
  k = low
 while ((i <= mid) and (j <= high))
  if a[i] < a[j]
  C[k] = A[i]
   i = i + 1;
   k = k + 1;
 Else
 C[k] = A[j]
   j = j + 1;
   k = j + 1;
```

```
while (i<= mid)
 c[k] = A[i]
 i = i++;
 k = k++;
while(j <= high)
{c[k] = A[j]
 j = j+1;
 k = k+1;
 }
}
for( i = low ; i<high ; i++)
  {a[i] = c[i];
}
}
```

Time Complexity : $O(n \log_2 n)$

$2T\left(\frac{n}{2}\right)+n$

Min Max

```
Algorithm Min_max(i,j,max,min)
{ if(i==j)                                   //n=1
   max = min = a[i];
 else if (i=j-1)
{ if a[i] <a[j])                             //n=2
 { max = a[j];
   min = a[i];
 } else
{ max = a[i];
   min = a[j];
}
 else
{ mid = (i+j)/2                              //n>2
 Max-min Min_max(i,mid,max,min);
         Min_max(mid+1,j,max1,min1);
        if(max<max1) then
          max=max1;
       if (min <min1) then
         min = min1;
    }
```

Time Complexity : $O(n)$

$$T(n) = \begin{cases} 0 & ; n = 1 \\ 1 & ; n = 2 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2 & ; n > 2 \end{cases}$$

$$2T\left(\frac{n}{2}\right) + 2$$

left side
max
min

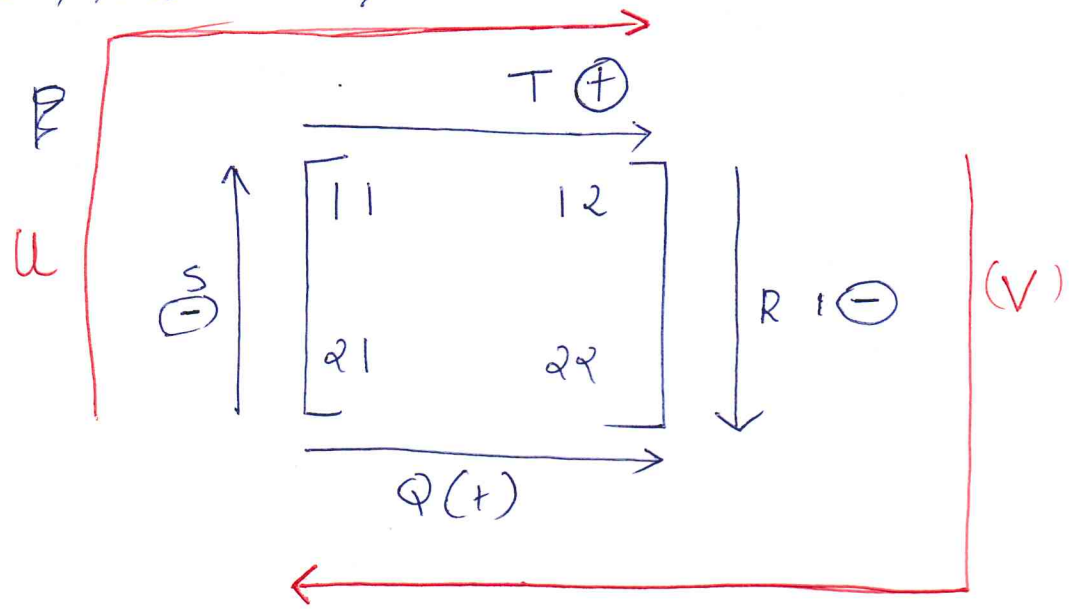
Right side
max1
min1

Insertion Sort

```
Algorithm InsertionSort(A, n)
{ for i = 1 to n-1 do
  { temp = A[i]
    j = j - 1
  while [A[j] > temp and j >= 0)
  { A[j+1] = A[j]
    j--;
  }
  A[j+1] = temp;
  }
```

Time Complexity = $O(n^2)$

STRASSEN'S MATRIX MULTIPLICATION



$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = B_{11}(A_{21} + A_{22})$$

$$R = A_{11}(B_{12} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12})B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

Quick Sort

TC = $O(n \log_2 n)$

$T(n) = \begin{cases} 0 & ; n = 1 \\ T(0) + T(n-1) + n & \end{cases}$

$T(0)$ → Equals zero as no left part

$T(n-1)$ → Sort $(n-1)$ Elements

n → Partitioned n elements

```
Algorithm QuickSort(A, lb, ub)
{
  if (lb < ub)
  {
    loc = par(partition(A, lb, ub))
      //Divide the array
    QuickSort(A, lb, loc-1)      //Sort the left part
    QuickSort(A, loc+1, ub)      //Sort the right part
  }
}

Algorithm Partition(A, lb, ub)
{
  pivot = A[lb];
  start = lb;
  End   = ub;
  while (start < End)
  { while (A[start] <= pivot and start < End)
    start++;
    while (A[End] > Pivot)
      End--;
    if (start < End)
    Swap(A[start], A[End]);
  }
  Swap(A[lb], A[End])
  return End;          //final position of Pivot Element.
}
```