Iterative Algorithms:

① — $i = n$

$\log n + 1$ — while $(i < n)$ → run $x+1$ times

{ print "SE"

$\log n$ $\quad$ $\log n$ $i = i/2$ }

Let 'x' times body is running

Loop terminate at $i = 1$

iteration #	1	2	3	4	....	$x+1$
i	n	$n/2$	$n/4$	$n/8$		
	$\frac{n}{2^0}$	$\frac{n}{2^1}$	$\frac{n}{2^2}$	$\frac{n}{2^3}$		$\frac{n}{2^x}$

Loop terminate at $i = 1$

$$\frac{n}{2^x} = 1$$

$$n = 2^x$$

$$\log_2 n = \log_2 2^x$$

$$\log_2 n = x \log_2 2$$

$$\log_2 n = x \qquad \therefore \log_2 2 = 1$$

$$T(n) = 3\log n + 2$$
$$= O(\log n)$$

① ← $i = 2$

$\log\log n + 1$ ← while $(i < n)$

{ $i = i * i$ → ← $\log\log n$

}

Let body run 'x' times so

~~iteration# 1 2 3 4 x+1~~

~~i 4 16 256 65536~~

~~2^2 2^4 2^8 2^{16} $2^{(x+1)^2}$~~

iteration#	1	2	3	4		x+1
i	2	4	16	256		
	2^1	2^2	2^4	2^8	. . .	2^{2^x}
	2^{2^0}	2^{2^1}	2^{2^2}	2^{2^3}	. . .	

Loop fail when $i = n$

$$2^{2^x} = n$$

$$\log 2^{2^x} = \log n$$

$$2^x \log_2 2 = \log n$$

$$2^x = \log n$$

$$\log 2^x = \log \log n$$

$$x = \log\log n$$

$$T(n) = 2\log\log n + 2$$

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

```
for (i←1 to n)
{
    while (j←1 to n)
    {   print "SE"
        j←j*2
    }
}
```

$\log n \times n$

$O(n\log n)$

This idea works for independent loop i.e. inner while mei j i ki value ko use kre. So, then we goto

Dependent Loops:

```
for (i←1 to n)
{
    for (j=i to n)
    {
        print "SE"
    }
}
```

i	1	2	3	.....	n-1	n
printing time	n time	n-1	n-2		1	0

$$T(n) = n+(n-1)+(n-2)+\dots+3+2+1$$
$$= \frac{n(n+1)}{2}$$
$$= O(n^2)$$

CamScanner

```
for (i ← 1 to n)
{
  for (j = i to n)
  {  printing
     j = j * 2
  }
}
```

let body run x times

iter	1	2	3	
i	1	2	3	
j	1 → 2	2 → n	3 → n	
printing	$\log n$	$\log(n-1)$	$\log(n-2)$	$\cdots \log(2) + \log(1)$

$$T(n) = \log(n) + \log(n-1) + \log(n-2) + \cdots \log(1)$$

$$= \log(n \times (n-1) \times (n-2) \times \cdots \times 1)$$

$$= \log(n!)$$

$$\log(n!) = O(\log n^n)$$

$$= n \log n$$

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

Recursive Algorithms:

```
factorial (n)
{
  ① ← if (n==0)
        ① ← return 1.

        return n* factorial(n-1)
}
```

$T(n-1)$

① ①

Subsitution method

Recurrence:

S1 $T(n) = T(n-1) + 4$

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

$T(n-1) = T(n-2) + C$

So,

S2 $T(n) = [T(n-2) + C] + C$

$= T(n-2) + 2C$

S3 $= [T(n-3) + C] + 2C$

$= T(n-3) + 3C$

⋮

identify pattern & then

Sk (kth step)

$T(n) = T(n-k) + kC$

Recursion stops when $n-k=0 \Rightarrow k=n$

$T(n) = T(0) + nC$

$T(n) = 1 + Cn$

$T(n) = O(n)$

CamScanner

when $c + c$ then add
" $2c + c$ then dont add

```
T(n) — fibacci(n)
{
    if (n==0 || n==1)
        return n
    return fibacci(n-1) + fibacci(n-2)
}
```

$T(n-1)$ $T(n-2)$

$$T(n) = T(n-1) + T(n-2) + C$$
$$= T(n-1) + T(n-1) + C$$

S1 $T(n) = 2T(n-1) + C$

S2 $T(n) = 2[2T(n-2) + C] + C$
$= 2^2 T(n-2) + 2C + C$

S3 $= 2^2[2T(n-3) + C] + 2C + C$
$= 2^3 T(n-3) + 2^2 C + 2C + C$

$\vdots$

Sk $T(n) = 2^k T(n-k) + 2^{k-1} C + 2^{k-2} C + \dots + 4C + 2C + C$

Recursion stops when
$n - k = 0/1$
$n = k$

$$T(n) = 2^n T(0) + 2^{n-1} C + 2^{n-2} C + \dots + 2^2 C + 2C + C$$
$$= 2^n C + 2^{n-1} C + \dots + 2^2 C + 2C + C$$
$$= C(1 + 2 + 2^n + \dots + 2^n)$$
$$= C\left[(1)\frac{[2^{n+1} - 1]}{2-1}\right]$$
$$= 2^{n+1} - 1$$
$$= O(2^n)$$

~~H.W task~~

iterative
- Bubble Sort
- Selection "
- Insertion "
- Binary " (Recursive)

$5 = O(n^2)$
$n = O(n^2)$
$n^2 = O(n^2)$
$\log n = O(n^2)$

Asymptotic notations:

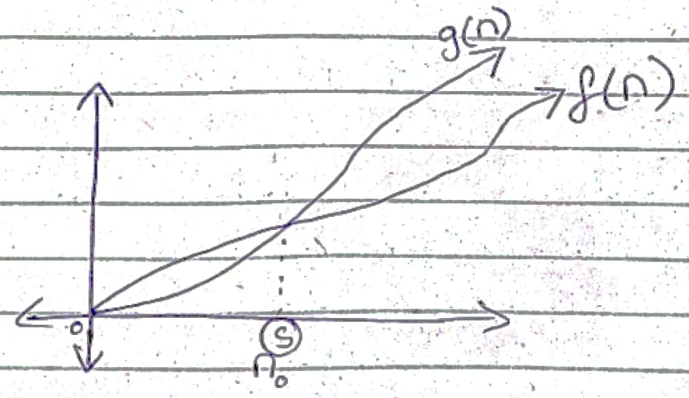
O - notation $\Rightarrow O(g(n)) = \{ f(n)$ there exist +ve constants c & n_0 s.t $0 \le f(n) \le c \cdot g(n)$ $(n \ge n_0) \}$

Ω - notation

Θ - notation

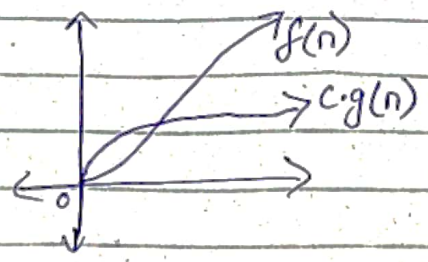
o - notation

ω - notation (Small omega)



For Ω, $\Omega(g(n)) = \{ f(n) \mid$ there exists c & n_0 such that

$$0 \le c \cdot g(n) \le f(n)$$



For Θ, $\Theta(g(n) = \{ f(n) |$ there exists constants c_1, c_2 & n_0 such that
$0 \le c_1 \cdot g(n) \le f(n) \le c_2 g(n) \}$

e.g. $n^2 < 2n^2 \le 3n^2$ (upper & lower same)

Asymptotic Intutions:

We'll perceive

- $O(1)$: So no redesign needed. It's constant time & we can't beat it.
- $O(\log n)$: half. Searching & inserting in binary search tree, Binary search.
- $O(n^2$ or $n^3)$: Polynomial Time (Acceptable but 'n' & 'exp' shouldn't be too large
- $O(n \log n)$: Best sorting algos (heap Sort, merge Sort etc)
- $O(2^n)$: Exponential time (like fibonnaci example). It is not acceptable (for large n)

~~Techniques:~~

~~Sorting~~ 1) **Divide & Conquer:** ↑ in parts divided problem & solve parts so global problem solved

Its a three step process:

1) Divide complex problem in small pieces
2) Conquer/Solve recursively
3) Combine: To Achieve global Solution

e.g.

Merge Sort:

Let an array be

8	7	9	4	5	2	3	6	1
0	1	2	3	4	5	6	7	8

1) Divide array of "n" size into two subparts (where size will be $\frac{n}{2}$ or $\frac{n}{2}-1$) so

8	7	9	4	5
0	1	2	3	4

2	3	6	1

2) Conquer: keep on dividing upto single element (single element trivially already sorted) & also sort & merge them together

8	7	9
0	1	2

4	5

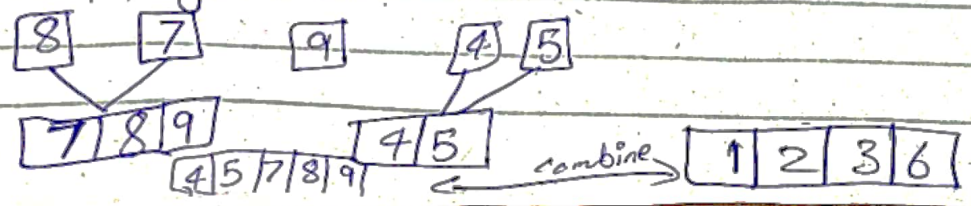
8	7

9

4	5

8	7

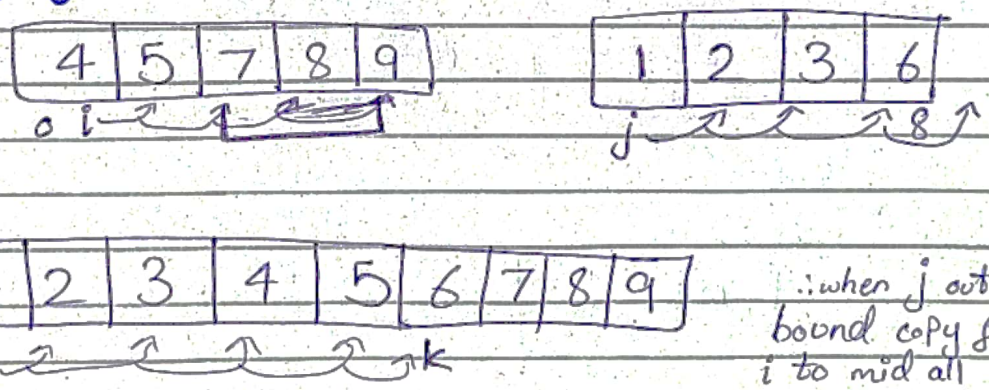
~~3) Combine/Merge:~~ i.e. now



It's algo is

```
T(n) MergeSort(Array(A), lowerbound(l), upperbound(h))
{
  1 ← if (l<h)
  {
    1 ← mid = (l+h)/2
  T(n/2) ← MergeSort(A, l, mid)
  T(n/2) ← MergeSort(A, mid+1, h)
  n ← merge(A, l, mid, h)
  }
}
```

Merging:



∴ when j out of bound copy from i to mid all

```
merge(A, l, mid, h)
{     new_Arr[h-l+1]          //declare new array
      i=l, j=mid+1, k=0
      while (i<=mid && j<=h)
  {   if (A[i] < A[j])
          newArr[k++] = A[i++]
      else
          new_Arr[k++] = A[j++]
```

```
while (i<=mid)        // i wali array abhi end nhi hui
    new_Arr[k++] = A[i++]

while (j<=h)
    new_Arr[k++] = A[j++]
  k=0
for (i<-l to h)
    A[i] = new_Arr[k++]

}
```

S1 $T(n) = 2T\left(\frac{n}{2}\right) + n$

S2

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$$

$$= 2^2 T\left(\frac{n}{4}\right) + n + n$$

$$= 2^2 T\left(\frac{n}{4}\right) + 2n$$

S3

$$T(n) = 2^2\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$$

$$= 2^3 T\left(\frac{n}{8}\right) + n + 2n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

Sk $T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$

CS CamScanner

stops when $\frac{n}{2^k} = 1$

So, $k = \log_2 n$

So, $T(n) = 2^{\log_2 n}(1) + n\log n$

$\therefore 2^{\log_2 n} = n^{\log_2 2}$

$T(n) = n + n\log n$

$= O(n\log n)$

CS CamScanner