How to Analyze time Complexity?

→ Running Time depends upon:

1) Single vs multi processor ×
2) Read/Write speed to memory ×
3) 32 bit or 64 bit ×
4) Inputs ✓

so we don't see them when we talk about time Complexity.

but we see

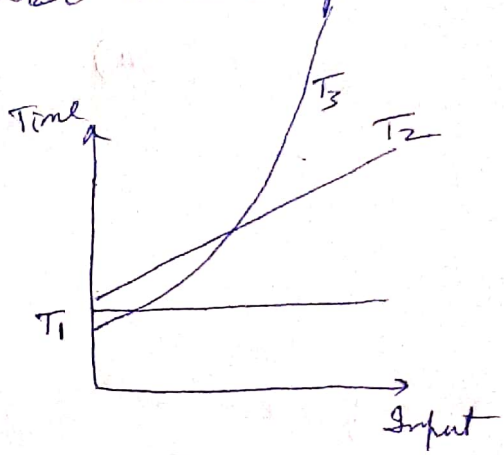
As we see the rate of growth of time w.r.t to our inputs

eg:

T_1 $T_{sum\ of\ two\ nos}$ = Constant time or $O(1)$

T_2 $T_{sum\ of\ nos\ in\ array}$ = $O(n)$

T_3 $T_{sum\ of\ nos\ in\ matrix}$ = $O(n^2)$ so



This big O is called Asymptotic notation
we also have some other Asymptotic notions like Θ, Ω etc.

#

Suppose we have two algorithms

Algo 1 : $T(n) = 5n^2 + 7$
Algo 2 $T(n) = 17n^2 + 6n + 8$

} Based on some Model Machine

Q) So why we only take n and not Constants in Asymptotic notations.

Ans) Generally we see for larger values of n say $n = \infty$

So that makes other constants and lesser power of n insignificant as compared to larger powers

eg: ① $17 \times n^2 + 6n + 8$ and ② $5n^2 + 7$

At $n = \infty$ this is very huge as compared to so we just consider n^2

The both equation will have some behaviour for bigger values of n.

Do we define the nature of rate of growth of time in terms of some Asymptotic notations.

1) O - "big-oh" notation :=>

Suppose we have a non-negative function $g(n)$ which takes a non-negative argument say n.

then

$O(g(n))$ is defined as the set of all the functions n s.t. $f(n)$

or $\{ f(n)$: there exist constants c and n_0 such that $f(n) \leq c \times g(n)$, for $n \geq n_0 \}$

eg:

$f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$

these two will always be lesser than

$5n^2$ so $\rightarrow$ $8n^2 \geq f(n)$ ☆

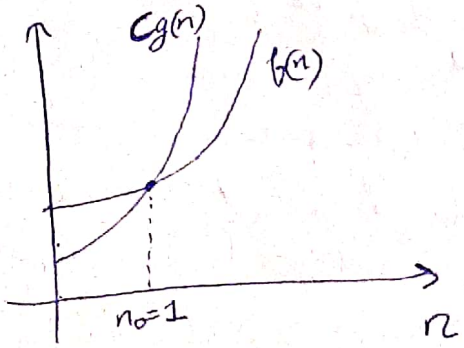
$c = 5+2+1 = 8$

$n_0 = 1$

then $O(n^2)$

which is $c \times g(n)$

so



Do this gives us the upper bound of rate of growth of a function i.e time can't grow faster than this

Note :-> We can choose c and n_0 but the conditions will still be valid

2) Ω - Omega notation:→

Suppose if we have +ve function $g(n)$ which takes positive argument n then

$\Omega(g(n)) = \{ f(n) :$ there exist constants c and n_0 such that

$cg(n) \leq f(n)$ for $n \geq n_0$ $\}$

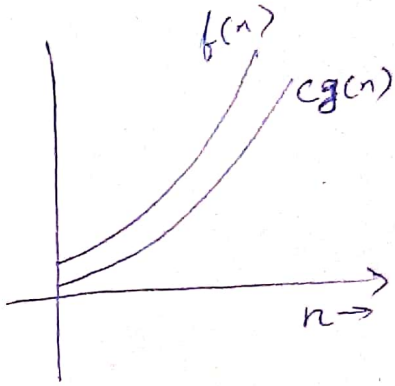
eg:

$f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$

$c = 5$ $\quad 5n^2 \leq f(n) \quad n \geq 0$

$n_0 = 0$



we may say $f(n)$ is $\Omega(n^2)$

So Omega notations gives us lower bound of a function i.e. atleast the time will grow with this rate

3) Θ — Theta notation.

It says if we have a +ve function $g(n)$ that takes +ve argument $f(n)$ then

$\Theta(g(n)) = \{ f(n) :$ there exist constants c_1 and c_2 and n_0 such that

$c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$ $\}$

eg: $f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$

$\Theta(n^2)$

$c_1 = 5$

$c_2 = 8$

$n_0 = 1$

If we choose them then conditions of Θ holds

Θ theta notation gives the best idea about the rate of growth of ~~function~~ time because, it gives us a tight bound unlike Big Oh and Omega which gives us upper bound and lower bound

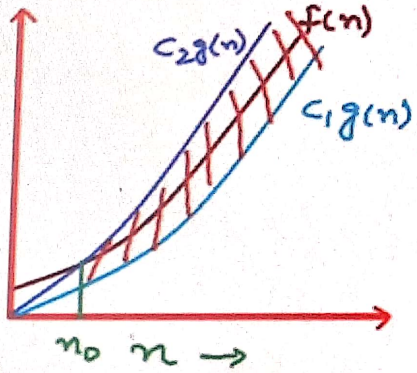
Time Complexity analysis - asymptotic notations

Θ - Theta notation - Tight bound

$\Theta(g(n)) = \{ f(n)$: there exist constants c_1, c_2 and n_0,

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

for $n \geq n_0$ $\}$



$f(n) = 5n^2 + 2n + 1 = \Theta(n^2)$

$g(n) = n^2$

$c_1 = 5, c_2 = 8, n_0 = 1$

Time Complexity Analyses - some general rules.

We analyze time complexity for :-

a) Very large input size

b) Worst case scenario.

Suppose this is the function :- $T(n) = n^3 + 3n^2 + 4n + 2$

↑ for very large value of n
i.e $(n \xrightarrow[\text{to}]{\text{tend}} \infty)$
this portion will become insignificant and

$T(n) \approx n^3$
or
$C \times n^3$ or $O(n^3)$

So
Rule 1) a) Drop all the lower order terms
b) Drop constant multiplier

eg: $T(n) = 17n^4 + 3n^3 + 4n + 8$

Rule a) $17n^4$
Rule b) n^4

So $O(n^4)$

eg: $16n + \log n$

Rule a) $16n$
Rule b) n

So $O(n)$ or $\Theta(n)$

Rule :- We can calculate the Running time of an algorithm by Suming the running time of all fragments

a) All the simple declaration run in constant time $O(1)$

eg:
```
int a;
a=5;      } O(1)
a++;
```

b) If we have a loop then time Complexity will be the number of times the loop runs multiplied by the running time of statements inside

eg:
```
for( i=0; i<n; i++)        // n+1
{
  //simple statement       // n x 1
}
```
So $O(n)$

c) If nested loops are there then also we multiply considering one at a time eg:
```
for( )
{
  for( )
  {
    statement  → n x n x 1 = n^2
  }
}
```
So $O(n^2)$

Suppose we have a program like this

```
int a;
a = 5;
a++;
```
} O(1)

```
for (i=0; i<n; i++)
{
  // simple statements
}
```
} O(n)

```
for(i=0; i<n; i++)
{
  for(j=0, j<n, j++)
  {
    // simple statements
  }
}
```
} O(n²)

$T(n) = O(1) + O(n) + O(n^2)$

$= O(n^2)$ because for very large values of n the bracketed part becomes insignificant.

Rule :→ Do the most will always be taken

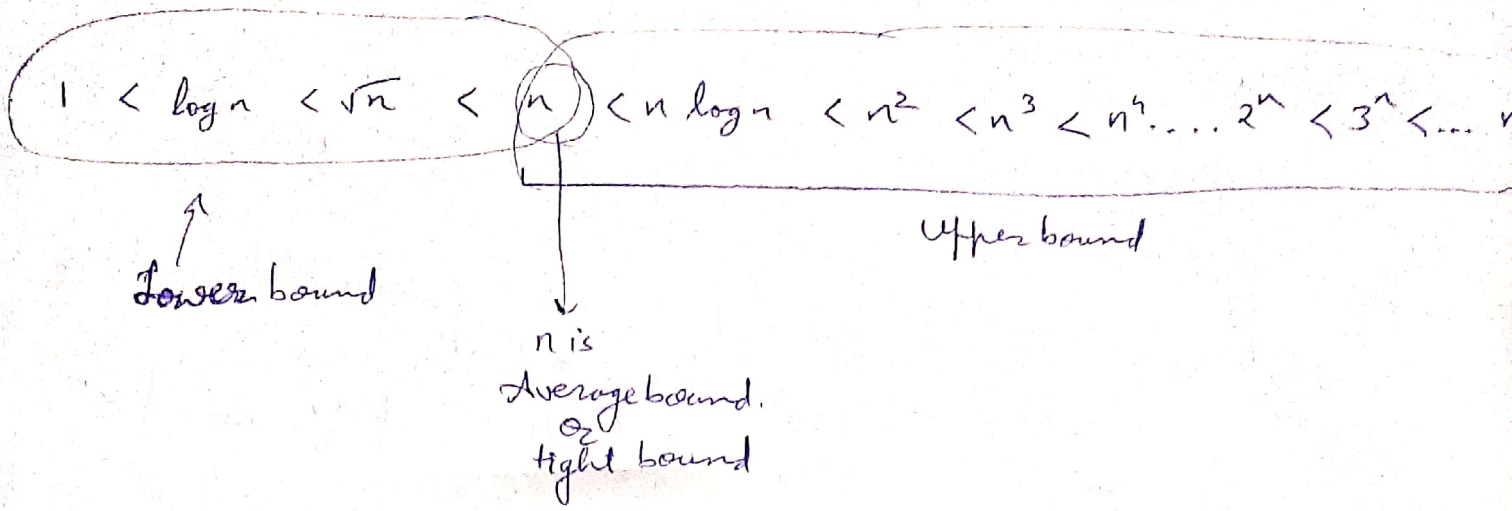
Suppose we have a program

```
if (some condition)
{
  for (i=0; i<n; i++)
  {
    // some statement
  }
}
else
{
  for (i=0; i<n; i++)
  {
    for (j=0, j<n, j++)
    {
      // some statement
    }
  }
}
```

for (i=0; i<n; i++) { // some statement } } O(n) ← Time Complexity will be this if the condition is true

for (i=0; i<n; i++) { for (j=0, j<n, j++) { // some statement } } } O(n²) ← Time complexity will be this else

Note :→ We always consider time complexity for worst case as discussed earlier



1) O - big - Oh

The function $f(n) = O(g(n))$ if and only if there exist a +ve constant Cand n_0 such that

$$f(n) \leq C * g(n) \quad \forall \quad n \geq n_0$$

eg: $f(n) = 2n+3$ $\qquad n_0 = 1$

$$2n+3 \leq C * g(n)$$

* Note:→ Always consider the nearest value

$1 * n$ ✗
$2 * n$ ✗
$3 * n$ ✗
$4 * n$ ✗
$5 * n$ ✓ condition satisfies after this as we consider
$6 * n$ ✓

n^2, $5n^2$, n^3, n^{10}
2^n, 3^n ....

So time complexity is: $f(n) = \Theta(n)$

2) Ω - Omega notation :→

The function $f(n) = \Omega(g(n))$ if and only if there exist a +ve constant C and n_0 such that

$$f(n) \geq C * g(n) \quad \text{where} \quad n \geq n_0 \quad \text{and} \quad n_0 = 1$$

eg: $f(n) = 2n+3$

$$2n+3 \geq C * g(n)$$

$1 \times n$ ✓
$1 \times \log n$ ✓ so least or nearest is $\log n$ so $\Omega(\log n)$
$1 \times \sqrt{n}$ ✓

3) Θ - Theta notation :→

$$C_1 * g(n) \leq f(n) \leq C_2 * g(n)$$

so $\Theta(n)$ not always depend on Upper and lower bound eg: n^2 n^3 etc

general rules

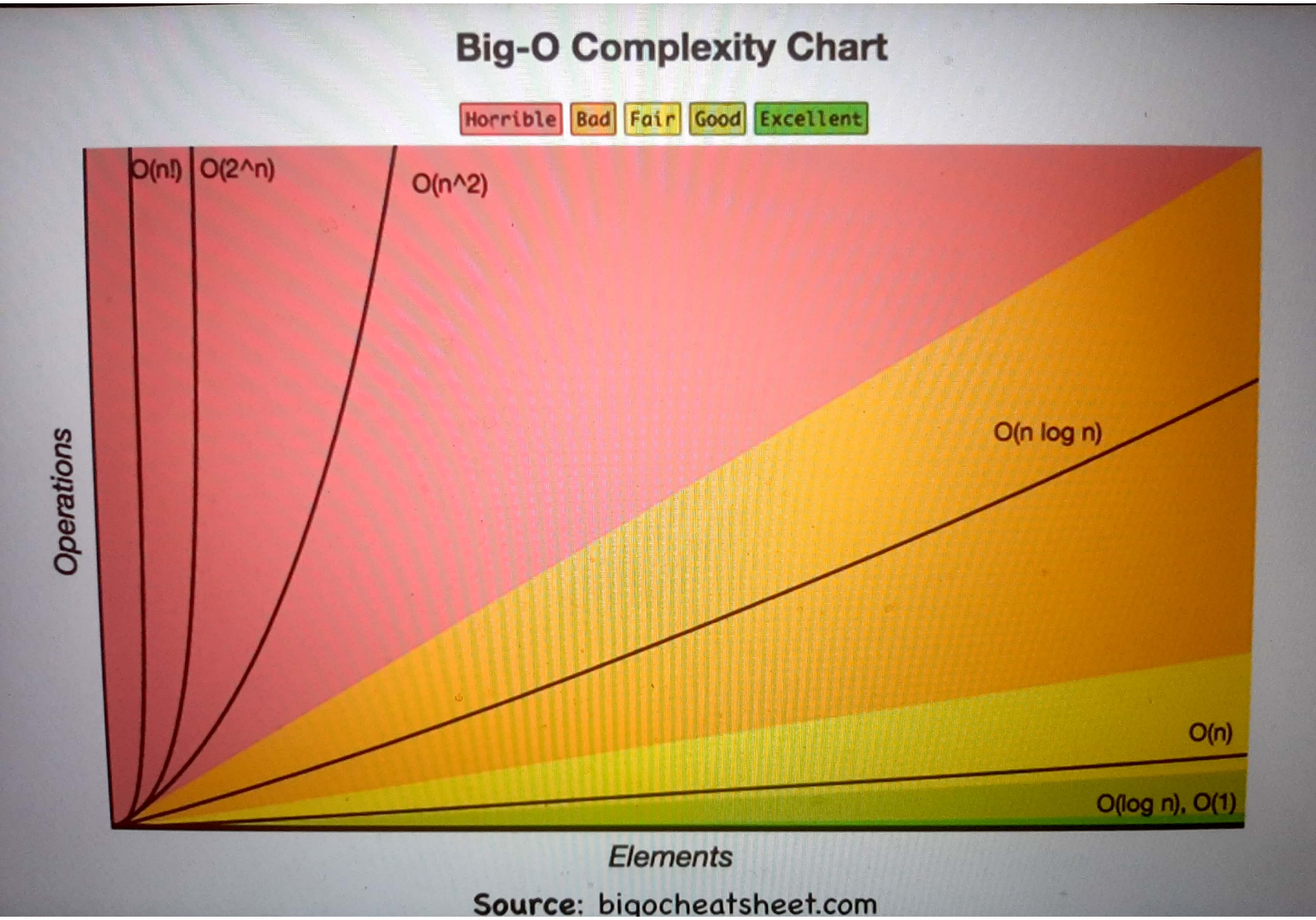
1. ignore constants

   $5n \rightarrow O(n)$

2. certain terms "dominate" others

   $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

   i.e., ignore low-order terms


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements
Source: bigocheatsheet.com

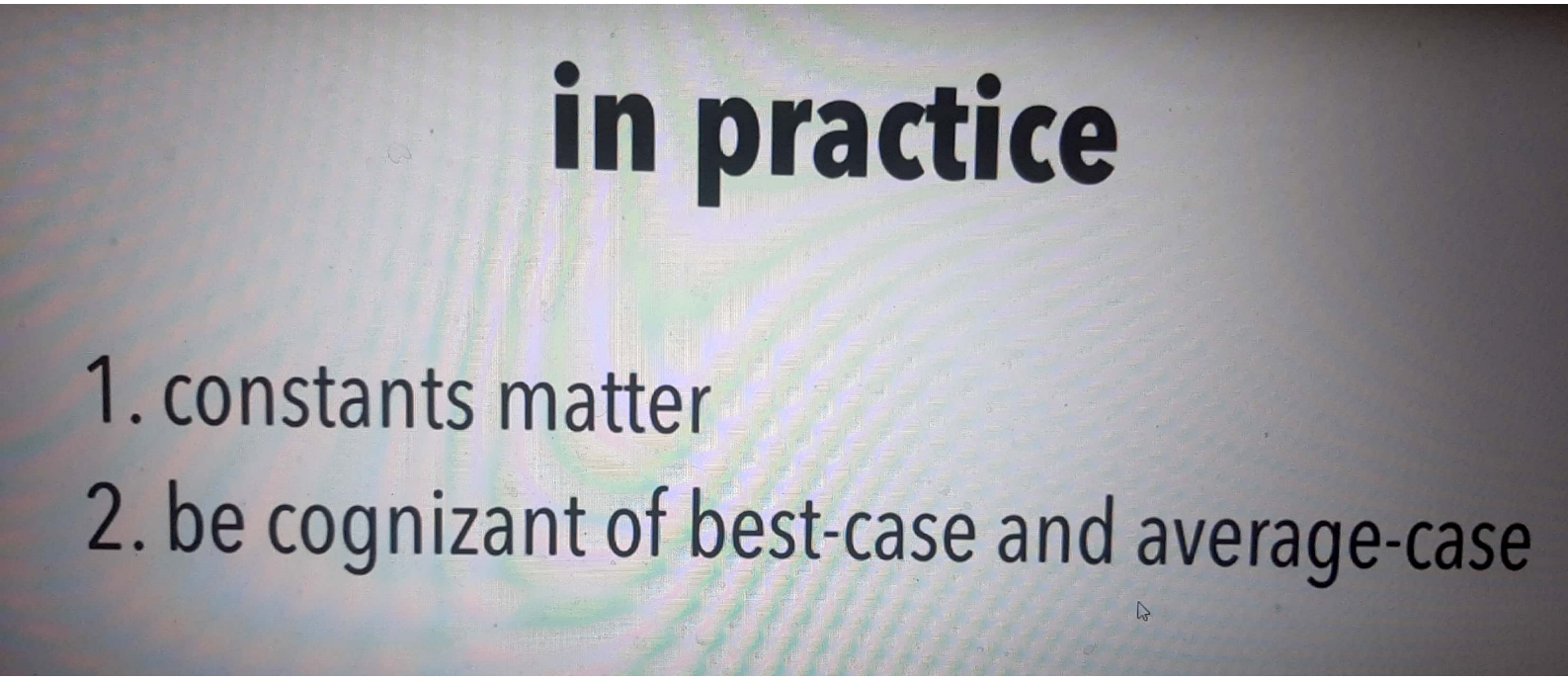
```
x = 5 + (15 * 20);
y = 15 - 2;
print x + y;
```

total time = O(1) + O(1) + O(1) = O(1)

3 * O(1)

```
y = 5 + (15 * 20);          O(1)
for x in range (0, n):   }  O(N)
    print x;
```

total time = O(1) + O(N) = O(N)


in practice
1. constants matter
2. be cognizant of best-case and average-case

#

Constant time :

```
x = 5 + (15 * 20);
```

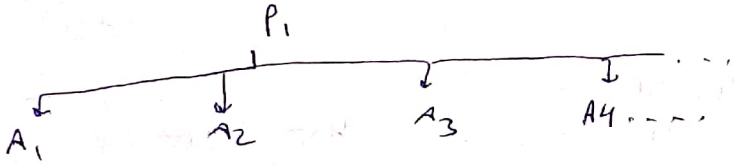
This basic computer step doesn't depend on anything in any way and we say it gets computed in constant time.

For Compt science we have some problems

↓ In order to solve them we write

Codes/programs in C, C++, python but before directly writing the program we must write our ~~dgaa~~ logic in an informal language i.e making a blueprint which is called as algorithm.

Like there is some Problem then it can have many solutions or algorithm.



So how to choose the best Sol^n. It is chosen based on time and Memory.

1) Time
2) Memory

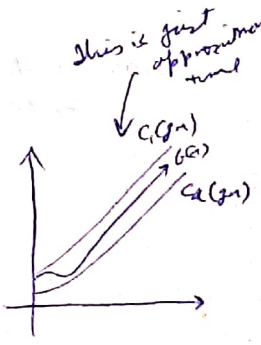
Best Algorithm is one which takes less time and memory.

so Design and Analyse

Practical significance of Asymptotic notations

1) O worst case
2) Ω best case
3) Θ Average

✱ We are always interested in the worst case of any algorithm.



Eg :

5	6	7	8	9	12	14	15

Let the size of this array be n.

We will do linear search

So in best case if x=5 we can find it one 1st location only. So $\Omega(1)$

and worst case is $O(n)$

When best case and worst case are same then we use the Average case

There are three types of Algorithms.

1) Iterative Algorithms ⟶ eg: for(i=1), while, do-while etc.

2) Recursive Algorithm. ⟶ eg: A(n) { if ( ) i.e. If a function calling itself
A(n/2)
}

Note :- Any Iterative Algorithm can be written in Recursive Alg. and vice-versa
So both are equivalent in power

Analysis wise they both are different. i.e Iterative and Recursive.

Note :⇒ If Algorithm doesnt contains either iteration or recursion then it means that there is no dependency on input size which means that running time for such algorithm will be a constant value.

eg:
```
A()
{
  int i;
  for(i=1 to n)
      Pf("Todeb")  → n
  }
```
So O(n)

```
A()
{ i=1
  for(i=1; i^2<=n; i++)
     Pf("Todeb")
}
```

i
1^{st}
2^{nd}
3
4
⋮
k

$k^2 \geq n$

$k = \sqrt{n}$

$O(\sqrt{n})$

or

$\Theta(\sqrt{n})$

---

eg:
```
A()
{ i=1; s=1;
  while (s<=n)
  { i++;
    s = s+i;
    Pf("Ravi");
  }
}
```
So $O(\sqrt{n})$

s	i
1	1
3	2
6	3
10	4
15	5
21	6
⋮	⋮
k	n
(n+k)	

so increment in i is linear but s depends on i.

So

$\frac{k(k+1)}{2}$

$k = \frac{n(n+1)}{2}$

$\frac{k(k+1)}{2} > n \quad = \frac{n^2+n}{2}$

$k^2 + k > 2n$

k^2

$k^2 > n$

$k > \sqrt{n}$

$O(\sqrt{n})$

k

At $s > n$, it will stop

#

```
A()
{
  int i,j,k,n;
  for(i=1; i<=n; i++)          // it is running from 1 to n
  {
    for(j=1; j<=i; j++)        // it depends on i
    {
      for(k=1; k<=100; k++)    // it is default
      {
        Pf("ravi");
      }
    }
  }
```

i	j	k
1 time	1 time	100 times
2	2 times	200 times
3	3 times	300 times
⋮	⋮	⋮
n	n times	n × 100 times

+

Note :- After full execution of loop and at termination the value of j again becomes 1 or equal to what it was before

So total nos of times this Pf("ravi") will get executed is sum of all k

$100 + 200 + 300 + \dots n \times 100$

$\Rightarrow 1 \times 100 + 2 \times 100 + 3 \times 100 + \dots n \times 100$

$\Rightarrow 100(1+2+3+\dots n)$

$\Rightarrow 100 \times \frac{n(n+1)}{2}$

$\Rightarrow 50(n^2+n)$

So $O(n^2)$

#

```
A()
{
  int i,j,k,n;
  for(i=1; i<=n; i++)          // so it will run from 1 to n
  {
    for(j=1; j<=i^2; j++)      // it depends on i
    {
      for(k=1; k<=n/2; k++)    // it will run for n/2
      {
        Pf("Todeeb");
      }
    }
  }
```

i	j	k
1	1 time	$n/2 * 1$
2	4 times	$n/2 * 4$
3	9 times	$n/2 * 9$
⋮	⋮	⋮
n	n^2 times	$n/2 * n^2$

+

$n/2 * 1 + n/2 * 4 + n/2 * 9 + \dots n/2 * n^2$

$n/2 (1+4+9+16+\dots n^2)$

$n/2 \times \frac{n(n+1)(2n+1)}{6}$ $O(n^3)$

A()

```
{ for(i=1; i<n; i=i*2)
    Pf ("rodeeb");
}
```

$O(\log_2 n)$

i
1 or 2^0
2 or 2^1
4 or 2^2
8 or 2^3
⋮
2^k

$2^k \geq n$

$k \log_2 2 \geq \log_2 n$

$k \geq \log n$

Note :-

So always remember whenever i is running from 1 to n and incrementation is in terms of 2 i.e $i = i*2$ in for loop

then time complexity will be $O(\log_2 n)$

if $i = i*3$

then time complexity will be $O(\log_3 n)$

and so on...

A()

```
int i, j, k;
for (i=n/2; i<=n; i++)
{
    for (j=1; j<=n/2; j++)
    {
        for (k=1; k<=n; k=k*2)
            Pf ("ravi");
    }
}
```

i	j	k
$n/2$ times	$n/2$ times	$\log_2 n$

* So all loops are independent.

So no need to unroll the loops as we did in earlier questions. So

$n/2 \times n/2 \times \log_2 n$

$\Rightarrow \frac{n^2}{2} \log_2 n$

$O(n^2 \log_2 n)$

eg: 1 → 3 boxes, 2 → 3 boxes, 3 → 3 boxes ... 3×3 = 9 boxes

#
```
A()
{ int i,j,k;
  for (i=n/2; i<=n; i++)
  { for (j=1; j<=n; j=2*j)
    { for (k=1; k<=n; k=k*2)
        Pf("Todeb");
    }
  }
}
```

i	j	k
n/2 times	$\log_2 n$	$\log_2 n$

All are independent
so
$n/2 \times \log_2 n \times \log_2 n$

$O\left(\frac{1}{2}(\log_2 n)^2\right)$

Assuming $n \geq 2$

```
A()
{ while (n>1)
  { n=n/2
  }
}
```

n
2^1

Case 2) when n is increasing with powers of 2

2^2
twice

2^3
thrice

So when | 2^k | k times |

So my loop is stopping here, so
$2^k = n$
$k = \log_2 n$

Case 2) when n is not a power of 2
Say $n = 20$
then

20 ✓
10 ✓
5 ✓
2 ✓
1 ... ×

] 4 times so $\log_2 20$

So $O(\lfloor \log_2 20 \rfloor)$

#
```
A()
{ for (i=1; i<=n; i++)
  { for (j=1; j<=n; j=j+i)
    { Pf("ravi"); }
  }
}
```

$O(n \log n)$

$i = k$
$j = 1$ to n
n/k

i	j
1	1
2	2
3	3
4	4
...	...
n	n

so for each value of i
it will run for
n times then
n values of i
it will run for
$O(n^2)$

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

$n = 1 + (n-1)3$
$\frac{n-1}{3} + 1 = n$

$\frac{n+2}{3}$

$\frac{n^2(n-1)}{2}$

$O(n^2)$

$1 + n^2 = 1 + (n-1)d$

$\frac{n^2+n}{2}$

i = 1	i = 2	$\frac{n-3}{2}$	i = 3
j = 1 to n	j = 1, 3, 5, 7, ... 2n−1		j = 1, 4, 7, 10, ...
n times			

i = k
1, 1+k, 1+2k, 1+3k, ...

A()

```
{
int n = 2^k;
for(i=1; i<=n, i++)
{
    j=2
    while (j<=n)
    {
        j = j^2
        Pf("ravi");
    }
}
}
```

i	j
1	2^1
2	2^2
3	2^4
4	2^8
⋮	⋮
n	

$2^8 \times 8$

$2^5 6$

2 | 256
2 | 128
2 | 64
2 | 32
2 | 16
2 | 8
2 | 4
2 | 2
1

As it is depending on n so

$n = 2^k$ lets take

- $k=1$

  $n = 2^1 = 4$

  so for this value j will run 2 ✓ 4 ✓ two times.

  and because of for loop, it will get repeated n times so $n \times 2$

- if $k=2$

  $n = 2^2 = 16$

  j will run 2, 4, 16 → three times

  and $n \times 3$

- if $k=3$

  $n = 2^3 = 256$ so j will run

  2, 4, 16, 256 → 4 times

  $4 \times n$

So for $k=k$

$(k+1) \times n$

So $(k+1) \times n$ times this algorithm will run.

in terms of only n.

as $n = 2^k$ $\quad \log_2 n = k \log_2 2$

so $k = \log_2 n$ $\quad \boxed{\log_2(\log_2 n) = k}$

so

$\log_2(\log_2 n) + 1) \times n$

hence

$(\log_2(\log_2 n) + 1)\, n$

$n \log_2(\log_2 n) + n$

$O(n \log_2(\log_2 n))$ ✓

2^k

$2k \log_2 2$

$2k = \log_2 n$

Masters Theorem.

$T(n) = a(n/b) + \Theta(n^k \log^p n)$ ← If recurrsion statement is like this and follows

where $a \geq 1, b > 1, k \geq 0$, p = Real Number ←

Case 1: If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

Case 2: If $a = b^k$, then (i) If $p > -1$ then $T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n)$
(ii) If $p = -1$ then $T(n) = \Theta(n^{\log_b a} \log\log n)$
(iii) If $p < -1$ then $T(n) = \Theta(n^{\log_b a})$

Case 3: if $a < b^k$ then
(i) if $p \geq 0$ then $T(n) = \Theta(n^k \log^p n)$
(ii) if $p < 0$ then $T(n) = \Theta(n^k)$

Limitations → Only Recurrsion Statements following the condition can be solved using Masters theorem.

eg: $T(n) = 4T(n/2) + n$

$a = 4$
$b = 2$
$k = 1$
$p = 0$

now $a > b^k$

$4 > 2^1$ ✓

So Case 1 is satisfied then $T(n) = \Theta(n^{\log_2 4})$

$T(n) = \Theta(n^2)$

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

Ans.

$a = 2$
$b = 2$
$k = 1$
$p = 1$

Case 2 satisfies and $p > -1$ then $T(n) = \Theta(n^{\log_2 2} \cdot \log^2 n)$

$= \Theta(n \log^2 n)$

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

$a = 2$
$b = 2$
$p = -1$
$k = 1$

Case 2: $p = -1$ so $T(n) = \Theta(n^{\log_2 2} \log\log n)$

$= \Theta(n \log(\log n))$

a) $T(n) = 2T(n/2) + n^2$

$a = 2$
$b = 2$
$p = 0$
$k = 2$

case 3
$p \geq 0$

$\Theta(n^k \log^p n)$

$(n^2 \log^0 n)$

$\Theta(n^2)$

$T(n) = T(n/2) + n$

$a = 0$
$p = 0$
$k = 1$
$b = 2$

case 3
$a < b^k$

$0 < 2^1$

$p \geq 0$

$n \log^0$

b) $T(n) = 3T(n/2) + \frac{n^2}{\log n}$

$a = 3$
$b = 2$
$p = -1$
$k = 2$

$3 < 2^2$

so
$a < b^k$

case 3
$p < 0$

$\Theta(n^2)$

$T(n) = 16^n T(n/3) + 1/n$

$a = 16^n$
$b = 3$
$k = -1$
$p = 0$

$16^n \quad 3^{-1}$

$16^n > 1/3$

$\Theta(\log_3 16^n)$

We can't solve it using Masters theorem as it is not in that form.

Recursive Algorithm :→

```
# A(n)
{ if( )
    return( A(n/2) + A(n/2) )
}
```

The way we found out time Complexity for Iterative Algorithms is very much different for Recursive Algorithm i.e. the reason is there is nothing to count here.

Let us Assume, in order to solve A(n) time taken is T(n), for if condition the time taken will be constant, and now the time taken for calling A(n/2) is $2T(n/2)$ as time taken for A(n) is T(n).

So

Total time taken for the function is

$$T(n) = 2T(n/2) + 6$$ → This is the recursive function now how to solve them.

1st Way) Back Substitution (It can be applied on all Recursive Algo. but it is slow)

eg:

```
# A(n)
{ if (n>1)
    return (A(n-1))
}
```

So $T(n) = T(n-1) + 1 \cdots ①$

now
$T(n-1) = T(n-2) + 1 \cdots ②$
$T(n-2) = T(n-3) + 1 \cdots ③$

now substitute eqn ② in ①

So n is starting and gradually decreasing ↓

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

now substitute eqn ③ in the above eqn so

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

⋮ k times

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

So $n - k = 1$

$k = n - 1$

So for this value of k it will become 1

Now where to stop, so see eqn 1 is valid only when $n > 1$

i.e $T(n) = T(n-1) + 1 \quad n > 1$

So at some point it will reach 1 and we will stop.

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

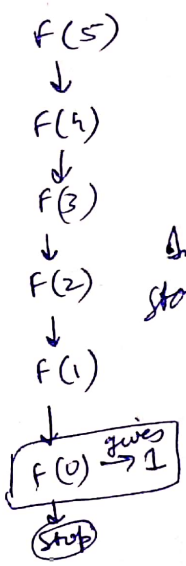
$= 1$

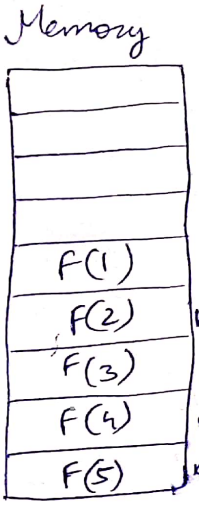
Complexity of Recursive programs.

```
Factorial(n)
{
  if n==0.
    { return 1; }
  else
    { return(n x Factorial(n-1)); }
}
```

Assume : n=5

so





So when n=5, 5 implicit stacks were formed in the memory.

now if n=n then n implicit stacks will be formed in the memory so

Space $\propto n$

So Space Complexity is $O(n)$

Why Recursive programs are not always good.

→ Check next page.

Complexity of Recursive programs.

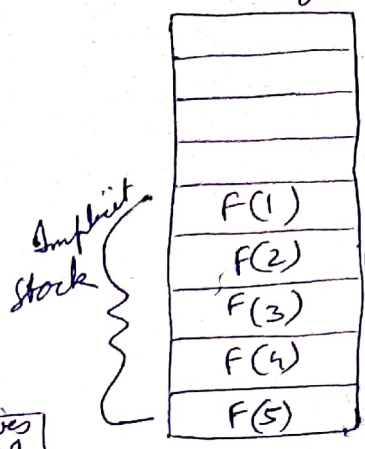
```
Factorial(n)
{
  if n==0.
    { return 1; }
  else
    { return(n x Factorial(n-1)); }
}
```

Assume: n = 5

so

F(5) → F(4) → F(3) → F(2) → F(1) → F(0) gives 1 → stop

Memory



finally the memory gets freed

So when n=5, 5 implicit stacks were formed in the memory.

now if n=n then n implicit stacks will be formed in the memory so

Space $\propto n$

So Space Complexity is $O(n)$

Why Recursive programs are not always good.

→ Check next page.

Recursion Tree method :-

$$T(n) = 2T(n/2) + c \quad ; \quad n > 1$$

↑ This is called online condition.

eg :- $T(n) = 2T(n/2) + n$

step 1) Make the given function as root node.

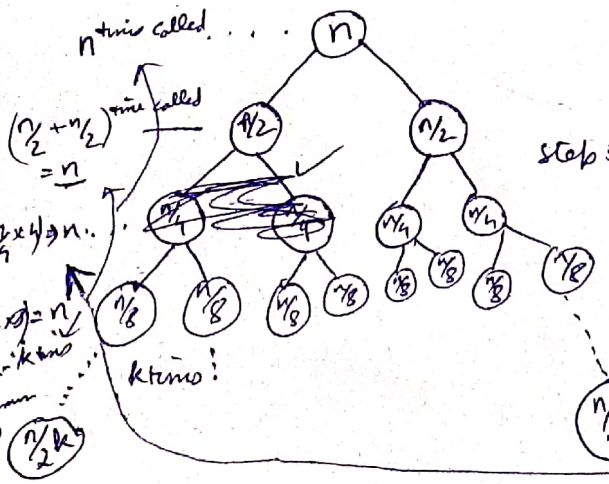
step 2) How many recursive terms are there.

- Here both values are $n/2$ but if values are different then lesser value goes to left side and higher value goes to right side.

step 3) Now how much time the function is called.

so

step 4) Assume



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

so $\boxed{k = \log_2 n}$

so $k * n = \log_2 n \times n \Rightarrow n \log_2 n$

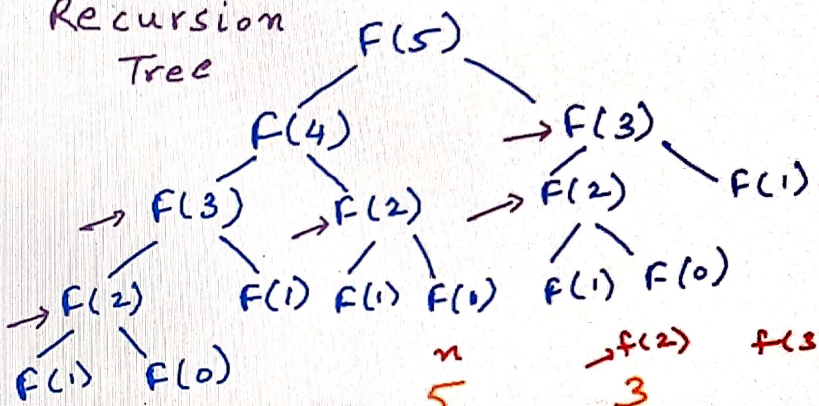
Albert (Iterative)

F(0)
F(1)
→ F(2)
→ F(3)
:
→ F(n)

F(5)
↳ F(0)
↓
F(1)
↓
F(2)
↓
F(3)
↓
F(4)
↓
→ F(5)

Pinto (Recursive)

Recursion Tree

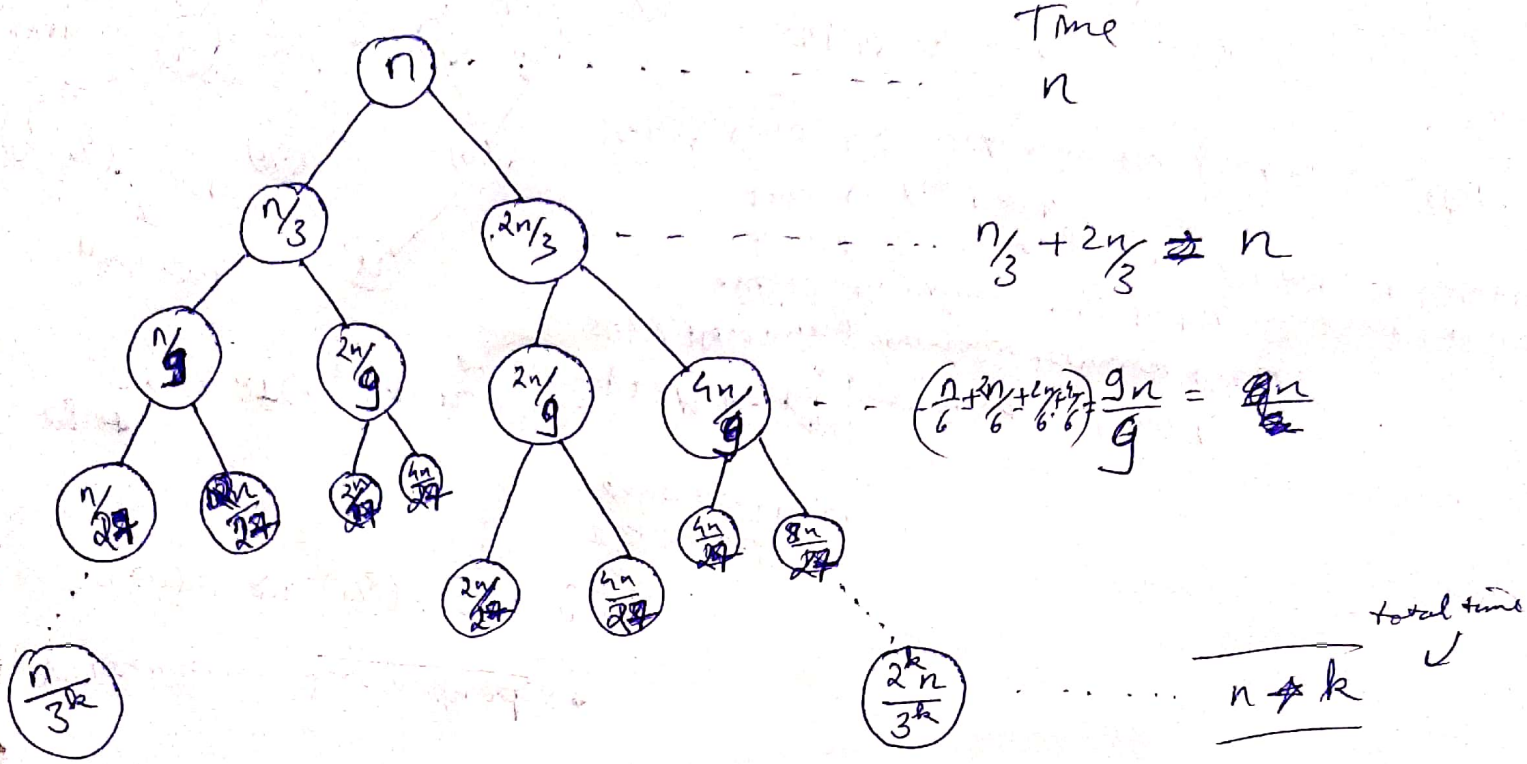


n	f(2)	f(3)
5	3	
6	5	
8	13	
40	63245986	

Q) Solve using Recursive tree method?

Ans)

$$T(n) = T(n/3) + T(2n/3) + n$$



Assume $\frac{n}{3^k} = 1$ to find the height of left subtree

$3^k = n$

$k = \log_3 n$

$\frac{2^k n}{3^k} = 1$ (to find height of Right subtree)

$2^k n = 3^k$

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

$\log_{3/2} n = k$

So total time is $n \times k$

$n \log_3 n$ ← this is the best case.

$n \log_{3/2} n$ ← this is worst case

Substitution Method Recursive Algo

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

so flog condition is always given.

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

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

- $T(n-2) = T(n-3) + (n-2)$

so $T(n) = T(n-3) + (n-2) + (n-1) + n$

⋮ k times

- $T(n-k) = T(n-k) + [n-(k-1)] + [n-(k-2)] + \dots + n$

and
$T(1) = 1$
$n-k = 1$
$k = n-1$

so

$= T(n-(n-1)) + [n-[(n-1)-1]] + \dots + n$

$= T(1) + 2 + 3 + 4 + \dots + n \quad [\because T(1) = 1]$

$= 1 + 2 + 3 + 4 + \dots + n$

So

$T(n) = \frac{n(n-1)}{2}$

$T(n) = \frac{n^2 - n}{2}$ so $O(n^2)$

#

$$T(n) = T(n-1) + \log n \quad \text{if } n > 1$$
$$T(1) = 1 \quad \text{if } n = 1$$

So

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

now:

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

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

now

$T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n$

⋮ k terms

- $T(n-k) = T(n-k) + \log(n-(k-1)) + \ldots$

now

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

as $T(1) = 1$

so

$n - k = 1$

$k = n-1$ now

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

$T(n) = T(1) + \log(2) + \log 3 + \log 4 + \ldots + \log n$

$= \log 1 + \log(2) + \log 3 + \ldots + \log n$

Note:

$\log(a \times b) = \log a + \log b$

So

$T(n) = \log(2 \times 3 \times \ldots \times n) + 1$

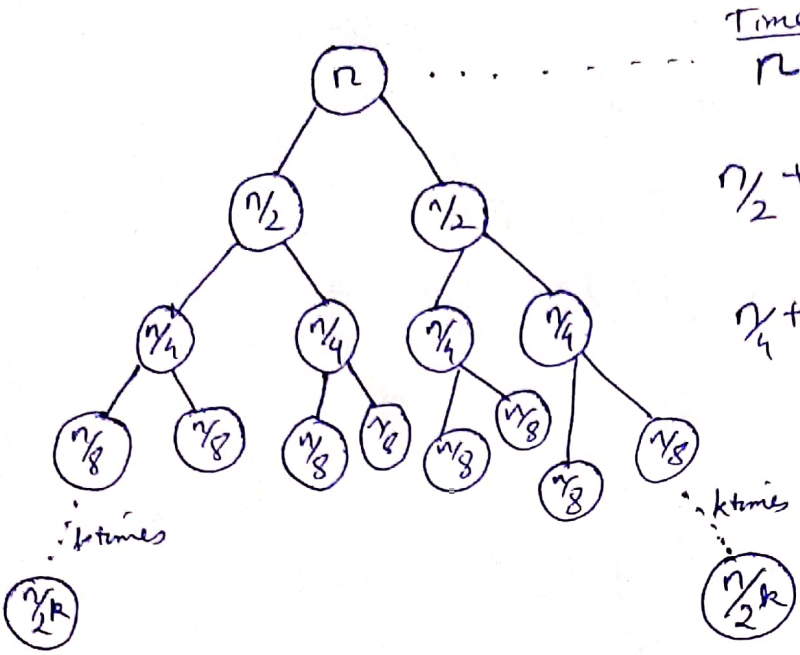
$T(n) = 1 + \log(n!) \Rightarrow 1 + \log n^n \Rightarrow 1 + n \log n$

$O(n \log n)$

#

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

TREE Method.



Time

n

$n/2 + n/2 = n$

$n/4 + n/4 + n/4 + n/4 = n$

$k * n$ = total time

Assume: $n/2^k = 1$

$2^k = n$

$k = \log_2 n$

so $T(n) = k \times n$

$T(n) = n \log_2 n$

so $O(n \log_2 n)$

Substitution Method.

- $T(n/2) = 2T(n/4) + n/2$

so $T(n) = 2 \times (2T(n/4) + n/2) + n$

$T(n) = 2^2 T(n/4) + \frac{n}{2} \times 2 + n$

- $T(n/4) = 2T(n/8) + n/4$

$T(n) = 2^3 T(n/8) + 2 \times \frac{n}{4} + n$

$\vdots$ k times

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

$T(n) = n\,T(1) + n + 2$

$[2^{\log_2 n} = n^{\log_2 2} = n]$

$[T(1) = ?]$

Tree Method

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

Time

1

1

1

k times

$\log_2 n = \log n$

k

$O(\log_2 n)$

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

$k = \log_2 n$

Substitution Method.

$T(n/2) = T(n/4) + 1$ so $T(n) = T(n/4) + 1 + 1$

$T(n/4) = T(n/8) + 1$ so $T(n) = T(n/8) + 1 + 1 + 1$

k times

$T(n) = T(n/2^k) + k$

so

$T(n) = T(1) + \log_2 n$

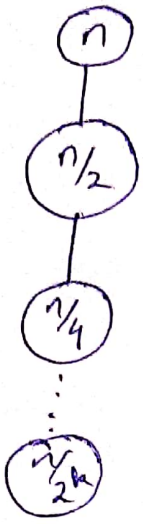
Assume

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

$k = \log_2 n$

Tree Method.

$T(n) = T(n/2) + n$



Time

n

$n/2$

$n/4$

: k times

So

$n + n/2 + \frac{n}{2^2} + \frac{n}{2^3} + \cdots \frac{n}{2^k}$

$n\left(1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^k}\right)$

Assume $\frac{n}{2^k} = 1$

$n = 2^k$

$k = \log_2 n$

So total time

$n(1+1)$

$T(n) = 2n$

$O(n)$

Substitution Method.

$T(n/2) = T(n/4) + n/2$ so $T(n) = T(n/4) + n/2 + n$

$T(n/4) = T(n/8) + n/4$ so $T(n) = T(n/8) + n/4 + n/2 + n$

: k times

$T(n) = T\left(\frac{n}{2^k}\right) + \frac{n}{2^k} + \frac{n}{2^{k-1}} + \cdots \frac{n}{4} + n/2 + n$

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

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

Introduction to Sorting Algorithms.

* Sorting is arranging the homogenous elements in increasing or decreasing order of some property.

eg: 2, 3, 9, 4, 6

1) Sorting on bases of increasing order
   2, 3, 4, 6, 9
2) Sorting on basis of decreasing order
   9, 6, 4, 3, 2
3) Sorting on basis increasing nos of factors.
   2, 3, 9, 4, 6

eg: "fork", "knife", "mouse", "screen", "key"
Sorting on basis of word order
"fork", "key", "knife", "mouse", "screen"

So Sorting helps in
1) Presentation of data
2) Retrieval of data
3) Computational power increases.

Different Sorting Algorithms are :-
1) Bubble Sort
2) Selection Sort
3) Insertion Sort
4) Merge Sort
5) Quick Sort
6) Heap Sort
7) Counting Sort
8) Radix Sort
   :
etc.

So we choose them on basis of time and space. and Stability

and Internal Sort and External Sort

Internal Sort: all records are in main memory or RAM

External Sort: records are on disks or tape.

Recursive or Non-recursive

Recursive: eg: Quick Sort, Merge Sort

Non-recursive: eg: Insertion sort, Selection Sort.

```
for (i=0; i<n, i++) {
key = A[i]
for (j=0; j<=n-1; j++) {
if (A[j] < A[j+1])
{
key = A[j+1]
}
else {
A[j] = A[j+1]
A[j+1] = key
}
}
}
```

My method turnedout to be a bubble sort method.

0	1	2	3	4	5
2	7	4	1	5	3

i=0 j=0

step 1: 2 is key

step 2: 7 (j=1)

step 3: 2 4 7 1 5 3

step 4: 2 4 1 7 5 3 (j=2)

step 5: i=0 j=3 → 2 4 1 5 7 3

step 6: i=0 j=3 → 2 4 1 5 3 7

step 6: key=4 i=1 j=0 → 4 4 1 5 3 7

step 7: i=1 j=1 → 4 1 4 5 3 7

step 8: j=2 i=1 → 4 1 ✗

step 7)

i=1 j=0 → 2 4 1 5 3 7

j will increment consequently.

2 1 5 3 4 7

i=2 j=0 → 2 1 5 3 4 7

2 1 5 3 4 7

j will increment

1 2 5 3 4 7

1 2 3 4 5 7

i=3 j=0 → 1 2 3 4 5 7

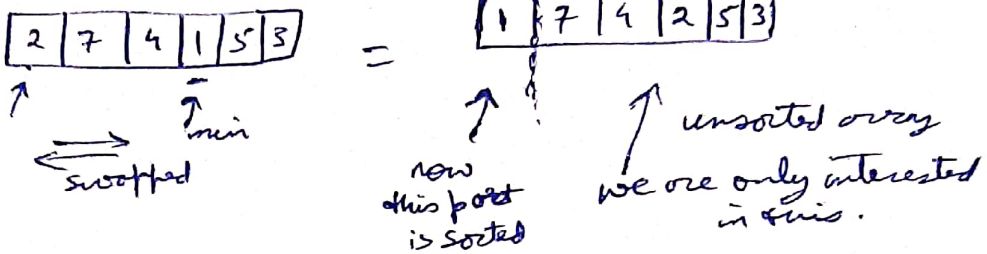
1) Selection Sort :→ The logic of selecting the minimum in each pass and putting at its appropriate position is Selection Sort Algorithm. (Inplace)

Lets say we need to sort this array

2	7	4	1	5	3
0	1	2	3	4	5

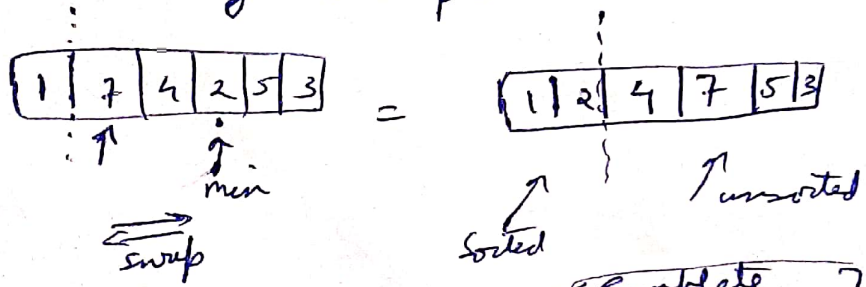
Step 1) We will select the 1st index value and swap it with the minimum number in the list

So



Step 2) Again repeating the Step 1 in unsorted array

So



Repeat this step until you get [Complete sorted array]

Pseudo-code :→

```
SelectionSort(n) {
    for (i = 0 to n-1)
    {
        indexofMinimum ← i
        for (j = i+1 to n)
        { if (A[j] < A[indexofMinimum]
                { indexofMinimum ← j
                }
        }
    }
```

$O(n^2)$

So Selection Algorithm is a slow Algorithm.

Bubble Sort :->

2	7	4	1	5	3

Comparing two adjacent ones at a time and then making the bigger one as bubble and moving it to its desired location.

Insertion Sort :->

It is not the best sorting algorithm, in terms of performance. But, it's a little more efficient than Selection sort and Bubble sort in practical scenarios. as nos of shifts and swaps are little less as compared to bubble and selection sort.

2 7 4 1 5 3

2 4 7 1 5 3

1 2 4 7 5 3

1 2 4 5 7 3

1 2 3 4 5 7

```
Insertion Sort (A, n) {
    for i ← 1 to n-1
    {   value ← A[i]
        hole ← i
        while (hole > 0 && A[hole-1] > value)
        {
            A[hole] ← A[hole-1]
            hole ← hole - 1
        }
        A[hole] ← value
    }
}
```

i	value	hole
1	2	1
2		

Merge Sort

We have talked about:

1) Selection sort
2) Bubble Sort
3) Insertion sort

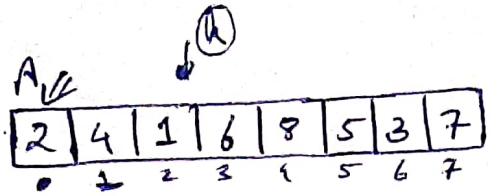
} $O(n^2)$ in average case

Merge sort - $O(n \log n)$ in worst case

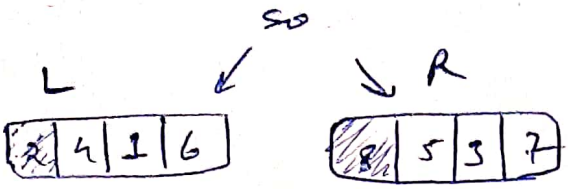
Merge Sort (O(nlogn)) :→ Dividing a bigger problem to smaller problem.

If array is odd then one half will have one extra element than other half.

eg:



So



We can know rearrange or sort these two arrays

So

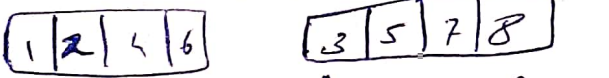
⟸ we can break it further also

So

i=0 j=0 and k=0

now we'll pick the smaller between the two and put it at 1st location of the array.

now check



i=1 j=0 k=2

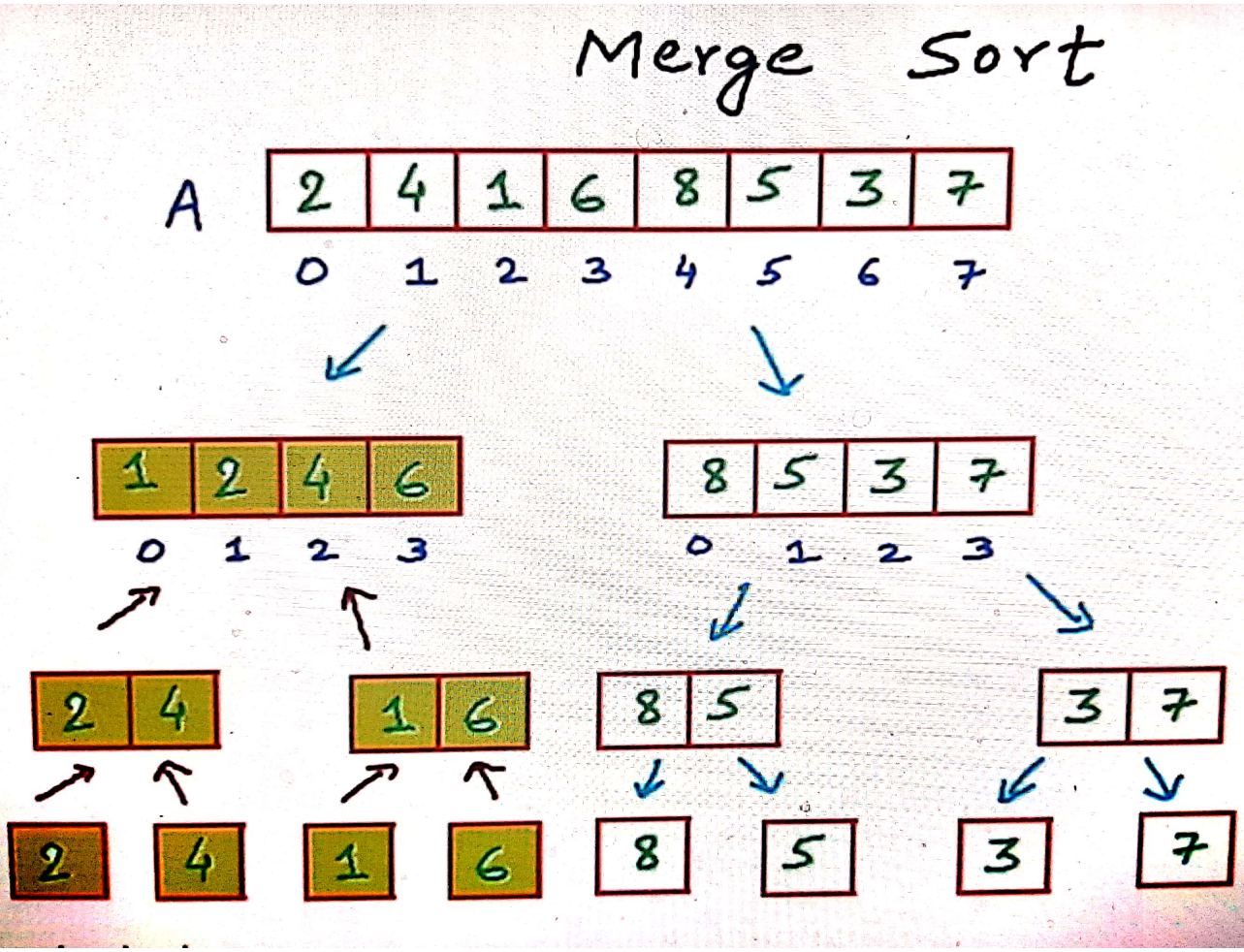
Sort those sublists and merge them back and they also can be divided

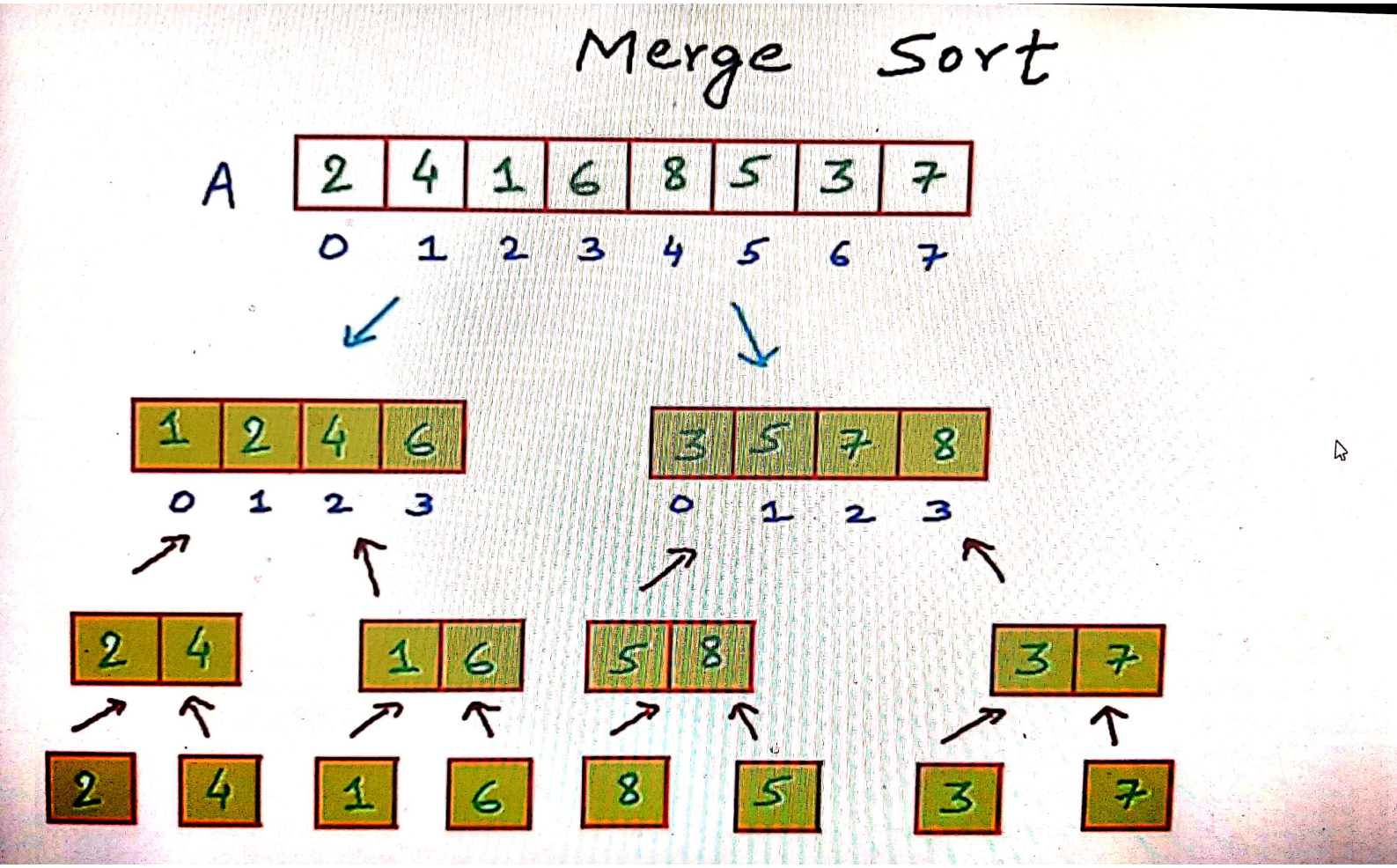
here our recursion ends now at this stage we can use our merge logic to merge them back.

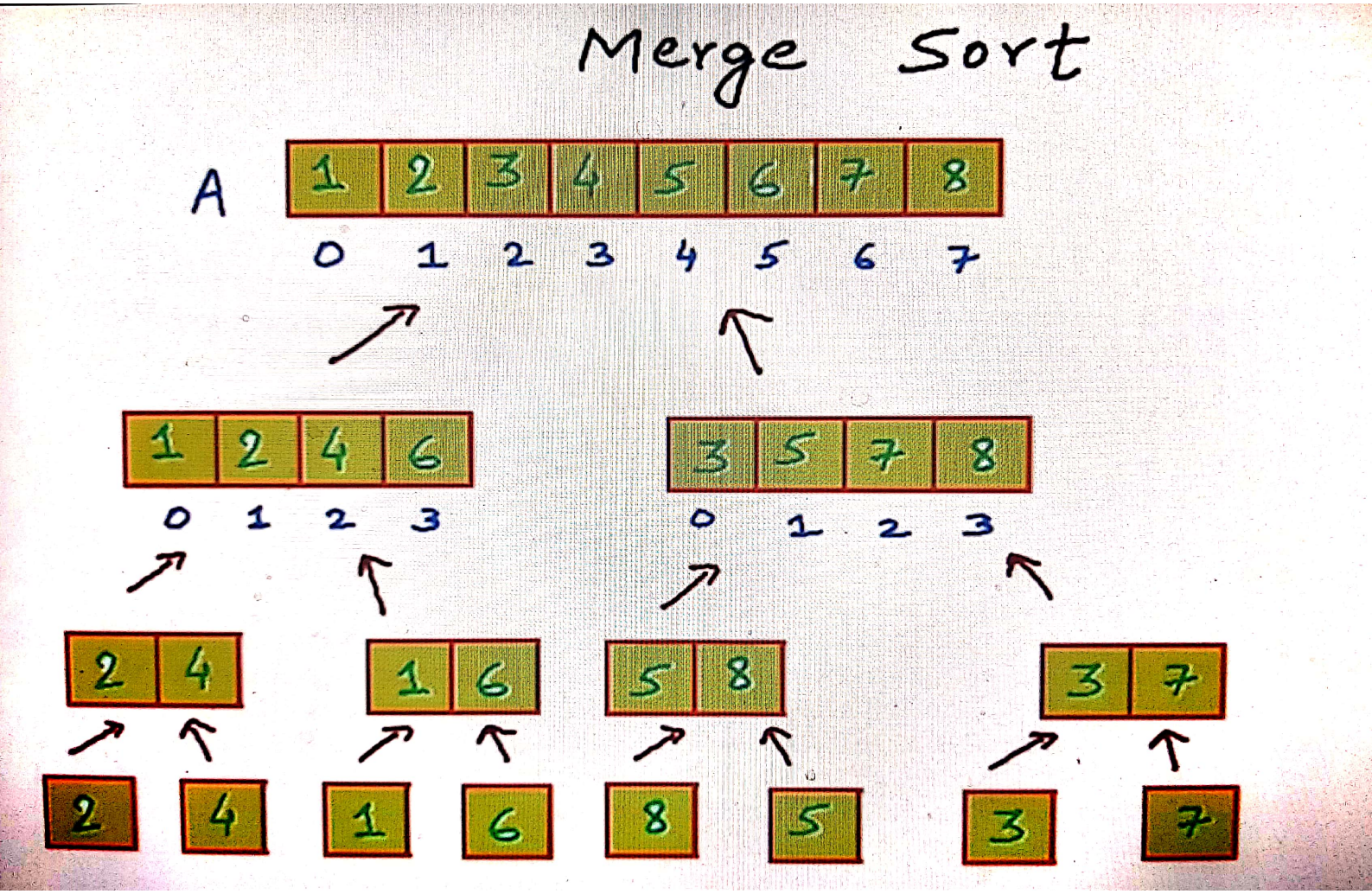
Pseudo-code

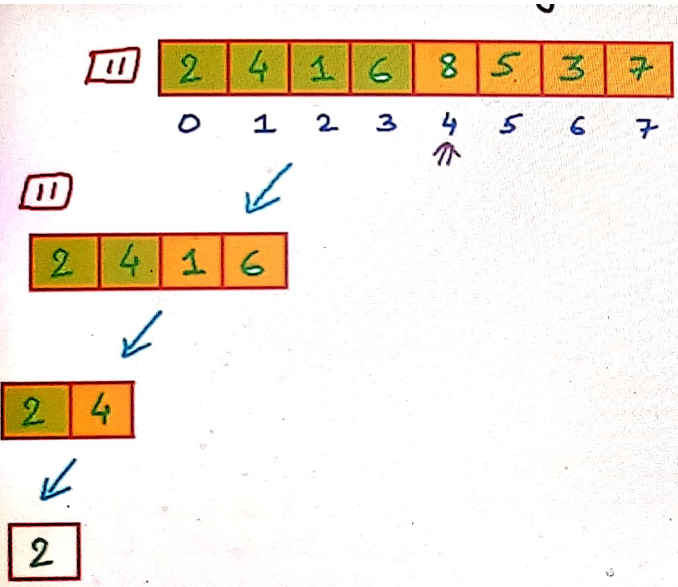
```
Merge (L, R, A)
{   nL ← length of L
    nR ← length of R
    i ← 0
    j ← 0
    k ← 0
    while (i < nL && j < nR)
    {   if ( L[i] ≤ R[j])
        {   A[k] ← L[i]
            k ← k+1
            i ← i+1
        }
        else {  A[k] ← R[j]
                k ← k+1
                j ← j+1
   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

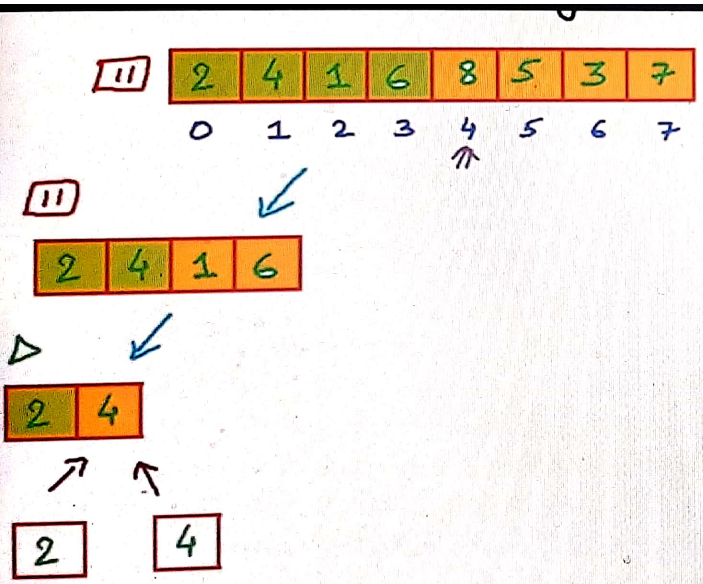



Merge Sort
A
2 4 1 6 8 5 3 7
0 1 2 3 4 5 6 7
1 2 4 6
0 1 2 3
3 5 7 8
0 1 2 3
2 4
1 6
5 8
3 7
2
4
1
6
8
5
3
7

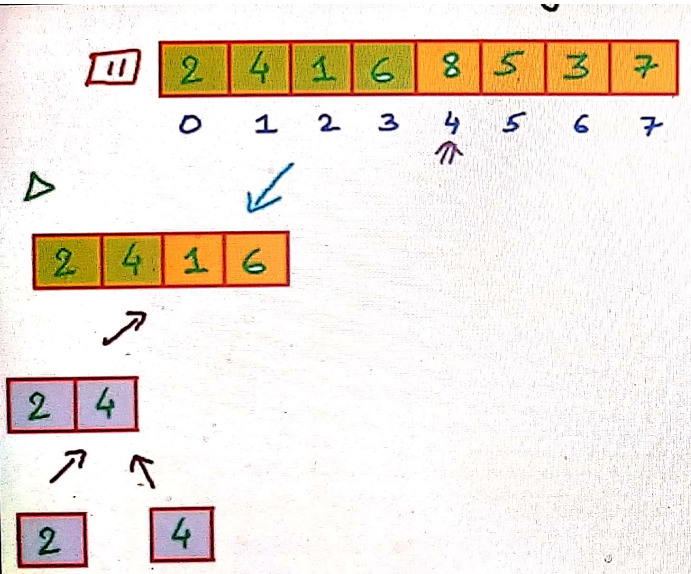

Merge Sort
A
1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7
1 2 4 6
0 1 2 3
3 5 7 8
0 1 2 3
2 4
1 6
5 8
3 7
2 4 1 6 8 5 3 7



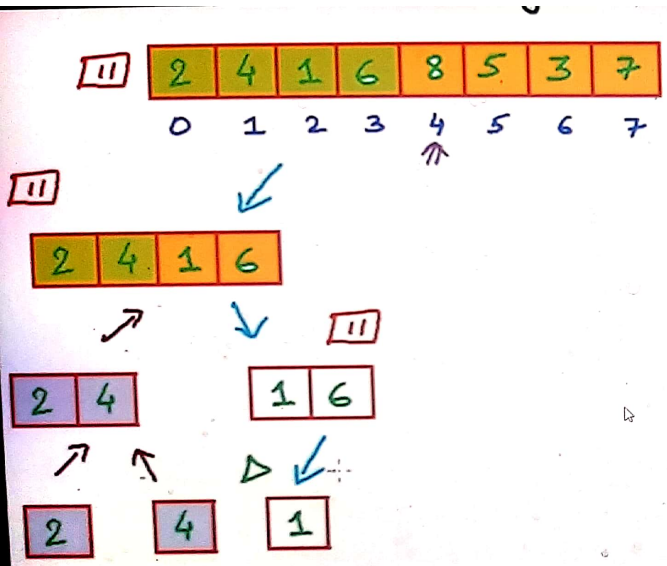

```
Mergesort (A)
{ n ← length(A)
base ← if (n < 2) return
Condition mid ← n/2
  left ← array of size(mid)
  right ← array of size(n-mid)
  for i ← 0 to mid-1
      left[i] ← A[i]
  for i ← mid to n-1
      right[i-mid] ← A[i]
  Mergesort (left)
  Mergesort (right)
} Merge(left, right, A)
```



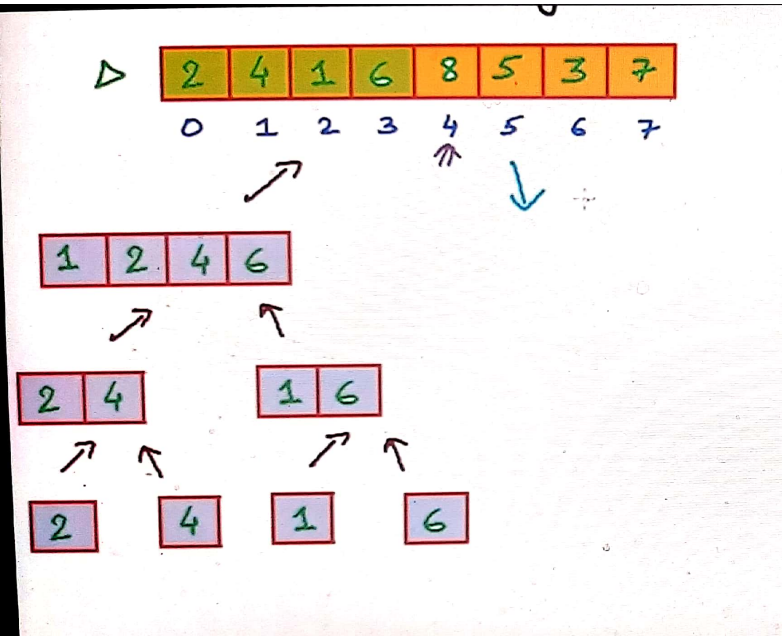

```
Mergesort (A)
{ n ← length(A)
base ← if (n < 2) return
condition mid ← n/2
  left ← array of size(mid)
  right ← array of size(n-mid)
  for i ← 0 to mid-1
     left[i] ← A[i]
  for i ← mid to n-1
     right[i-mid] ← A[i]
  Mergesort (left)
  Mergesort (right)
⇒ Merge(left, right, A)
}
```



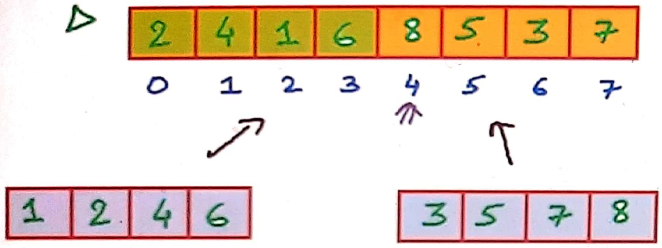

```
Mergesort (A)
{ n ← length(A)
base ← if (n < 2) return
Condition mid ← n/2
  left ← array of size(mid)
  right ← array of size(n-mid)
  for i ← 0 to mid-1
      left[i] ← A[i]
  for i ← mid to n-1
      right[i-mid] ← A[i]
  Mergesort (left)
  Mergesort (right)
⇒ Merge(left, right, A)
}
```



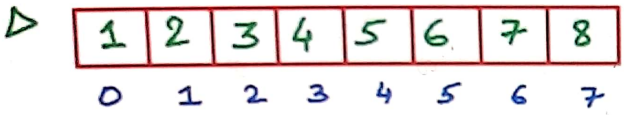

```
Mergesort (A)
{  n ← length(A)
base ← if (n < 2) return
Condition  mid ← n/2
   left ← array of size(mid)
   right ← array of size(n-mid)
   for i ← 0 to mid-1
      left[i] ← A[i]
   for i ← mid to n-1
      right[i-mid] ← A[i]
⇒  Mergesort (left)
   Mergesort (right)
}  Merge(left, right, A)
```




```
Mergesort (A)
{ n ← length(A)
base ← if (n < 2) return
Condition Mid ← n/2
   left ← array of size(mid)
   right ← array of size(n-mid)
   for i ← 0 to mid-1
      left[i] ← A[i]
   for i ← mid to n-1
      right[i-mid] ← A[i]
   Mergesort (left)
   Mergesort (right)
}  Merge(left, right, A)
```




```
Mergesort (A)
{ n ← length(A)
base ← if (n < 2) return
Condition mid ← n/2
  left ← array of size(mid)
 right ← array of size(n-mid)
 for i ← 0 to mid-1
     left[i] ← A[i]
 for i ← mid to n-1
     right[i-mid] ← A[i]
 Mergesort (left)
 Mergesort (right)
 Merge(left, right, A)
}
```




```
Mergesort(A)
{ n ← length(A)
base ← if (n < 2) return
Condition Mid ← n/2
    left ← array of size(mid)
    right ← array of size(n-mid)
    for i ← 0 to mid-1
        left[i] ← A[i]
    for i ← mid to n-1
        right[i-mid] ← A[i]
    Mergesort(left)
    Mergesort(right)
⇒ } Merge(left, right, A)
```

Analysis of Merge Sort

1) Divide and conquer
2) Recursive
3) Stable
4) Not In-place
   $\Theta(n)$ Space Complexity
5) $\Theta(n \log n)$ time-complexity

List of points in Cartesian plane:

(1,2), (2,5), (4,8), (2,3)

$\Rightarrow$ (1,2), (2,5), (2,3), (4,8)

equal keys?
relative order same as original list

udo-code

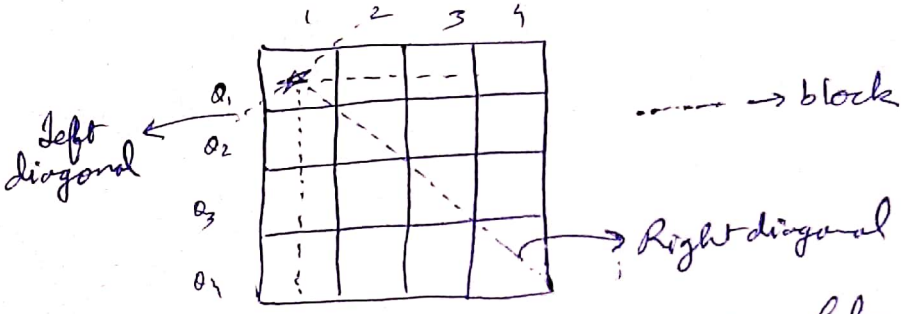
```
Merge (L, R, A)
{
    nL ← length of L
    nR ← length of R
    i ← 0
    j ← 0
    k ← 0
    while (i < nL && j < nR)
    {
        if ( L[i] ≤ R[j])
        {
            A[k] ← L[i]
            k ← k+1
            i ← i+1
        }
        else {
            A[k] ← R[j]
            k ← k+1
  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

Pseudo Code.

1) Create a 2D board
   int board[4][4];

2) Initialise the board with all values as 0.
   int board[4][4] = {0};

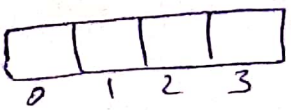
3) Now we will place our queen row-wise
So if the queen is in 1st row 1st column, we need to block the 1st column for all the coming queens and two diagonals. i.e.



So we will maintain 3 arrays one for Column bound, one for Right diagonal and one for left diagonal

Normally total nos of diagonals are 2*n-1
So we can make an array of this size.

Column bound
bool Col[4] = {0};



2n-1
2x4-1
7

(row + col - (bl-1))

← Right diagonal
bool Rd[7] = {0};

row+colum ← Left Diagonal
bool Ld[7] = {0};

```
void solve() (board[][], int row, bool col[], bool ld, rd) or
{
  for(col←0; col<4, col++)
  {
    if (Col[col] == false) and
       (Ld[row+col] == false
    and (Rd[row-col+bd by (-1)
         == false
    {
      board[row][col] = true;
      col[col] = true
      Solve(board, row+1, col,
            ld, rd);
```

Data Integrity :→ The following integrity exists in RDBMS :→

1) Entity Integrity :→ No two rows are identical
2) Domain Integrity :→ Each column must contain data of same type
3) Referential Integrity :→ Rows can't be deleted which are used by other Records.
4) User-Defined Integrity

Database Normalisation :→ It is the process of efficiently organising data in a database.

↳ Removes Redundancy.
↳ Ensures data-dependency.

1NF → A Cell should have atomic value.
2NF → There should n't be any partial dependency.
3NF → ① $\alpha \rightarrow \beta$
either α should be Super key
or β should be prime attribute.

BCNF :→ $\alpha \rightarrow \beta$ α should be a Super key.

Popular RDBMS :→ ① MySQL ② MS-Access ③ Oracle ④ MS SQL Server

↓
uses
SQL