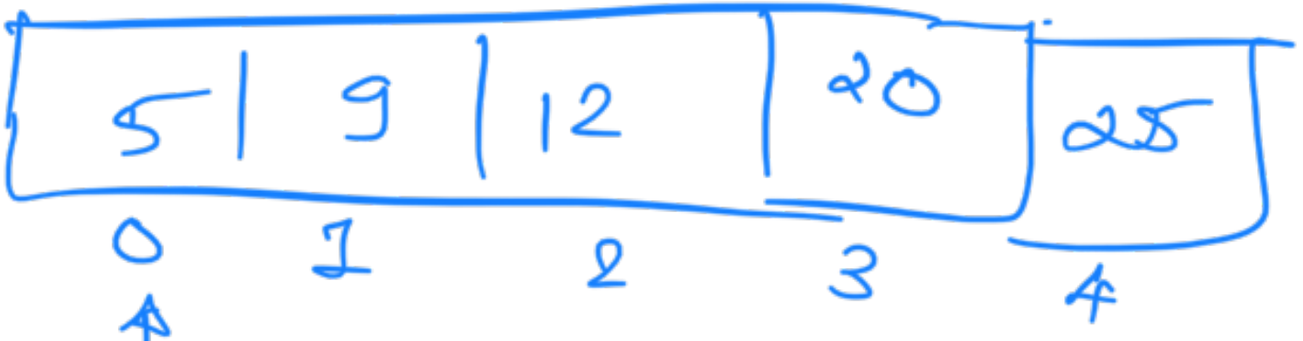
Linear Search

or Sequential Search



Algo

i

Seq Search (A, i, x, n)

- A → Array
- i → index
- x → element
- n → size of Array

You want to find

① Set $i = 0$

② if $i > n$

print ele. Not found

O(N)

③ if $A[i] = x$

print element Not found

④ $i = i + 1$

⑤ Exit

$i = 0$ if $i > 5$ $A[i] = 5$

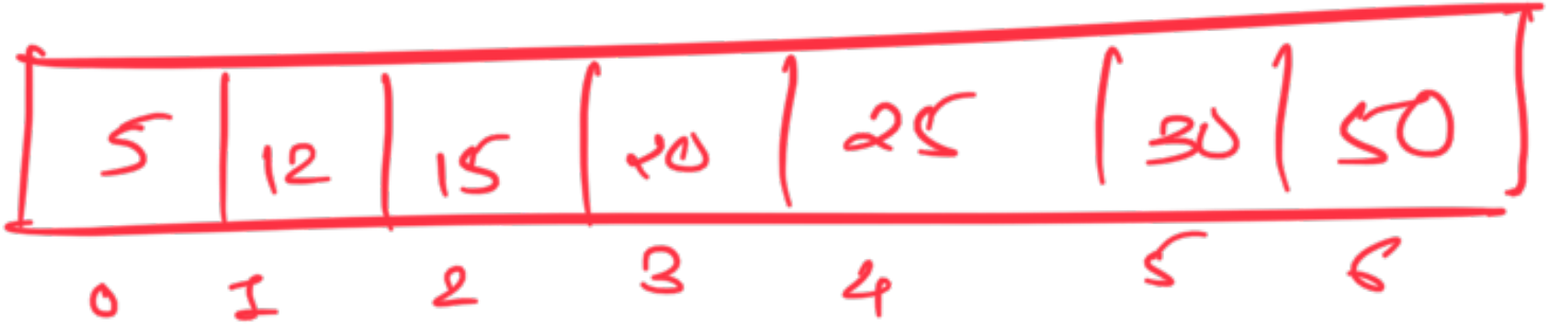
$i = 1$ if $i > 5$ $A[i] = 9$

$i = 2$ if $i > 5$ $A[i] = 12$

Found

⑪ Binary Search

- Time Complexity | O(log n)
- Based on divide & Conquer
- Data items | should be in sorted manner



Algo :

1. while (low ≤ high)

mid = low + high

mid = (low + high) / 2

(2) if x = A[mid] return mid

(3) if x > A[mid]

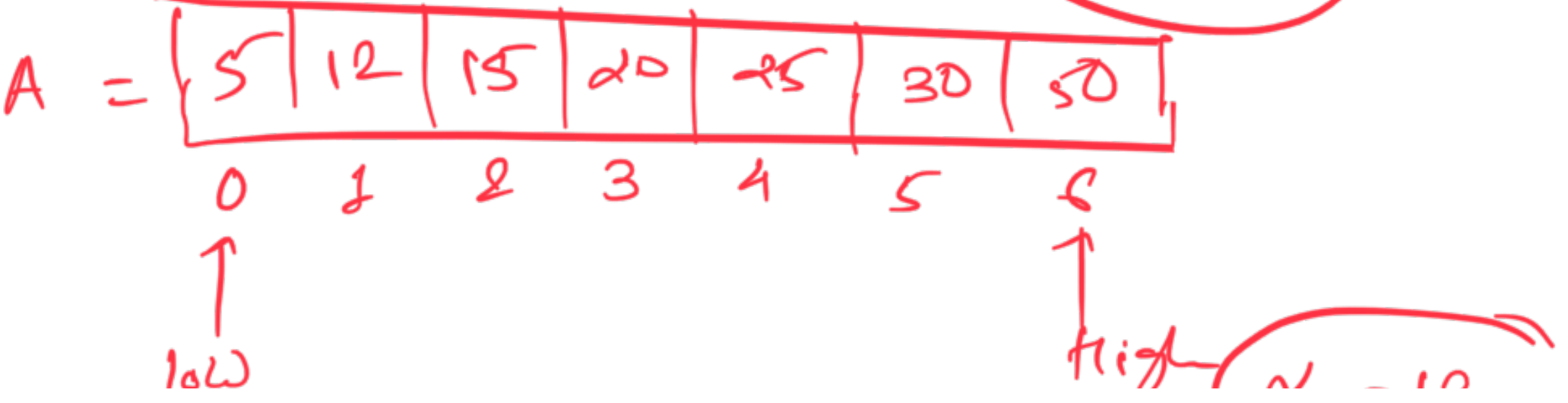
low = mid + 1

else

high = mid - 1

(4) exit

find 12



x = 12

(I) $mid = \frac{low + High}{2} = \frac{0+6}{2} = 3$

$x = A[3] = 20 \neq 12$

So,

if $x > A[mid]$ (x) } low = mid + 1

$12 > 20$ (x)

else

$x < A[mid]$

High = mid - 1 } بس

High = 3 - 1

= 2

(II) $mid = \frac{low + High}{2} = \frac{0+2}{2} = 1$

2

x = A[1] = 12

return 12

Bubble sort

→ each pair of Adjcent is compared and elements are swapped if they are not in order

Worst time Complexity = $O(n^2)$

worst case

N element → N swap
N times

Algo :

(1) While (till not get sorted)

if (list[i] > list[i+1])

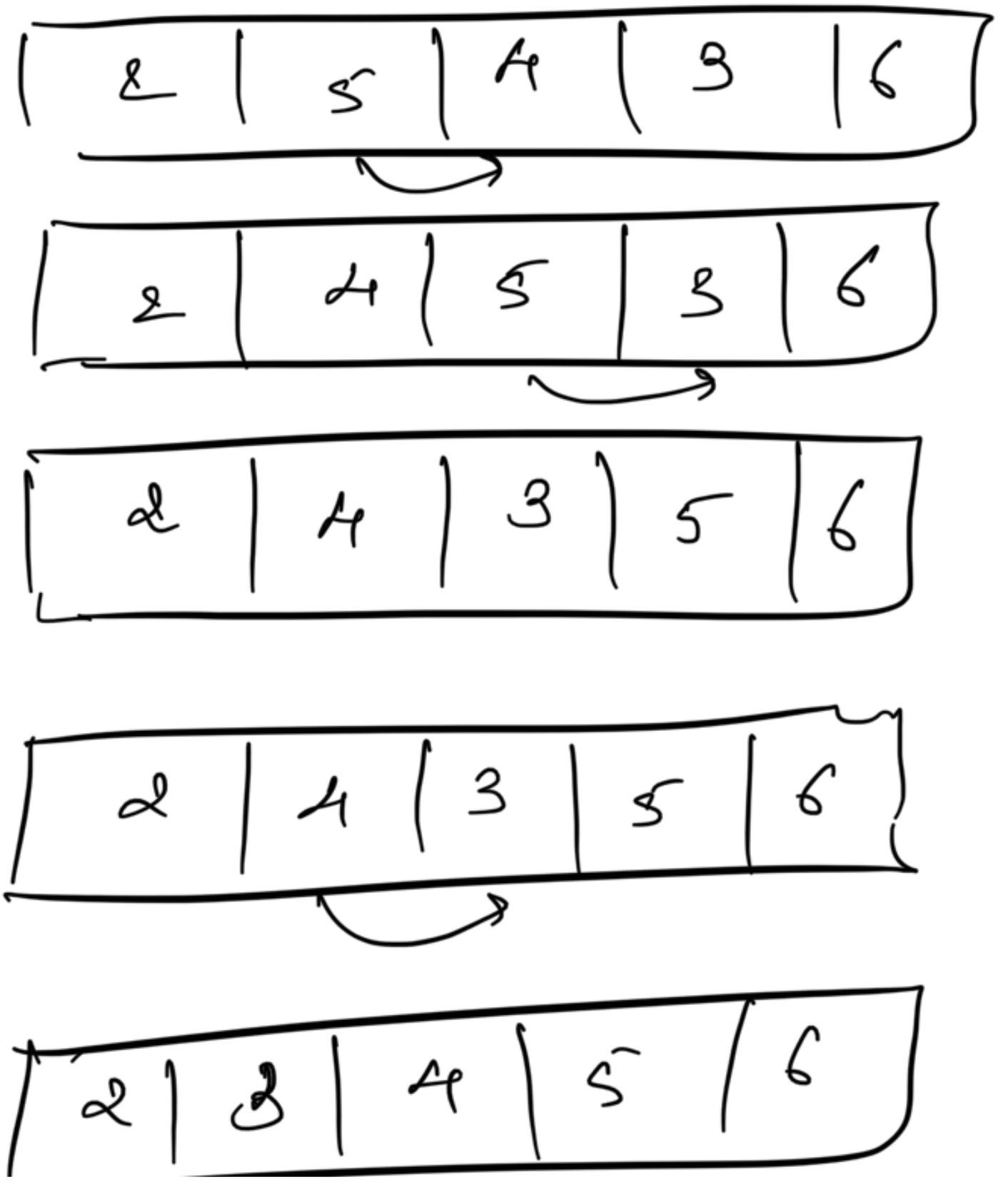
swap (list[i], list[i+1])

end if

end

exit

First Comparision with neighbour


2 5 4 3 6
2 4 5 3 6
2 4 3 5 6
2 4 3 5 6
2 3 4 5 6

Selection sort

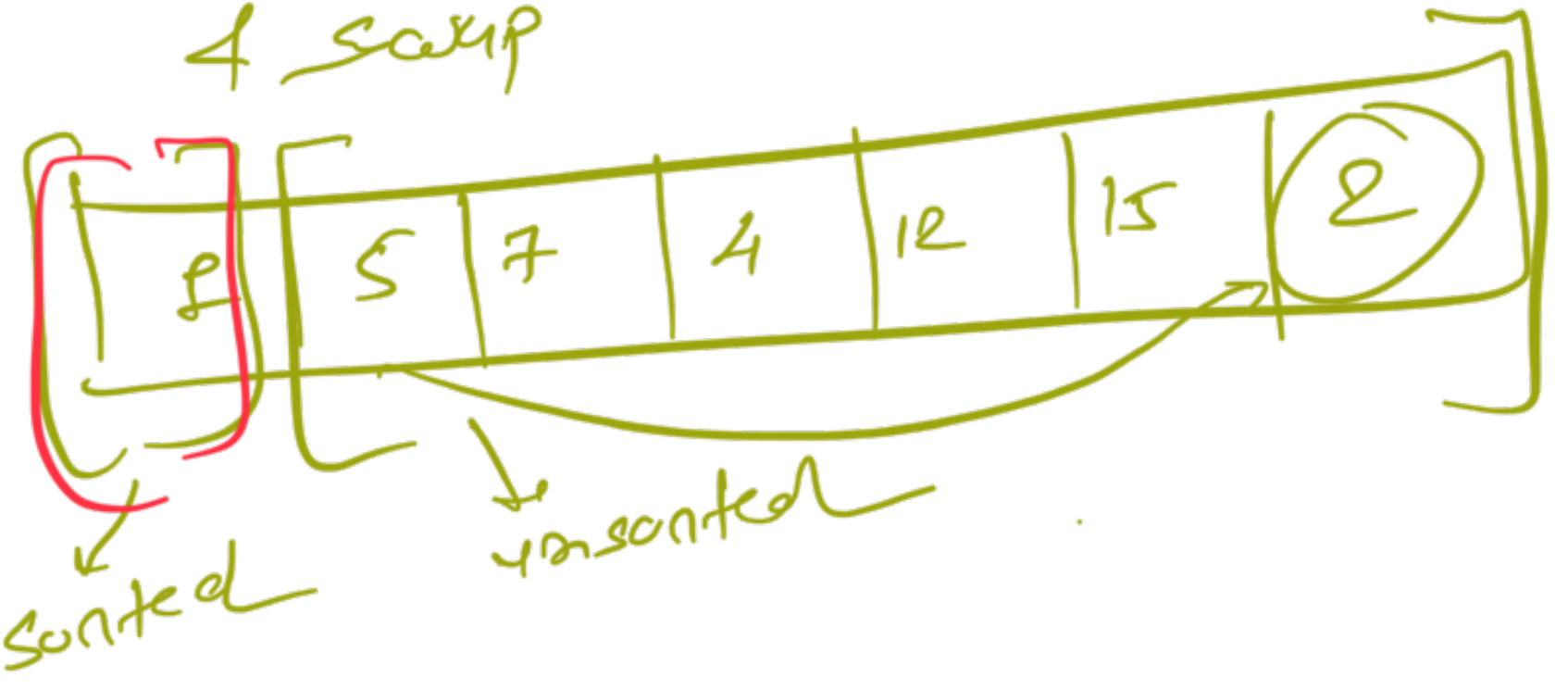
→ In this sorting technique list is devided into two parts

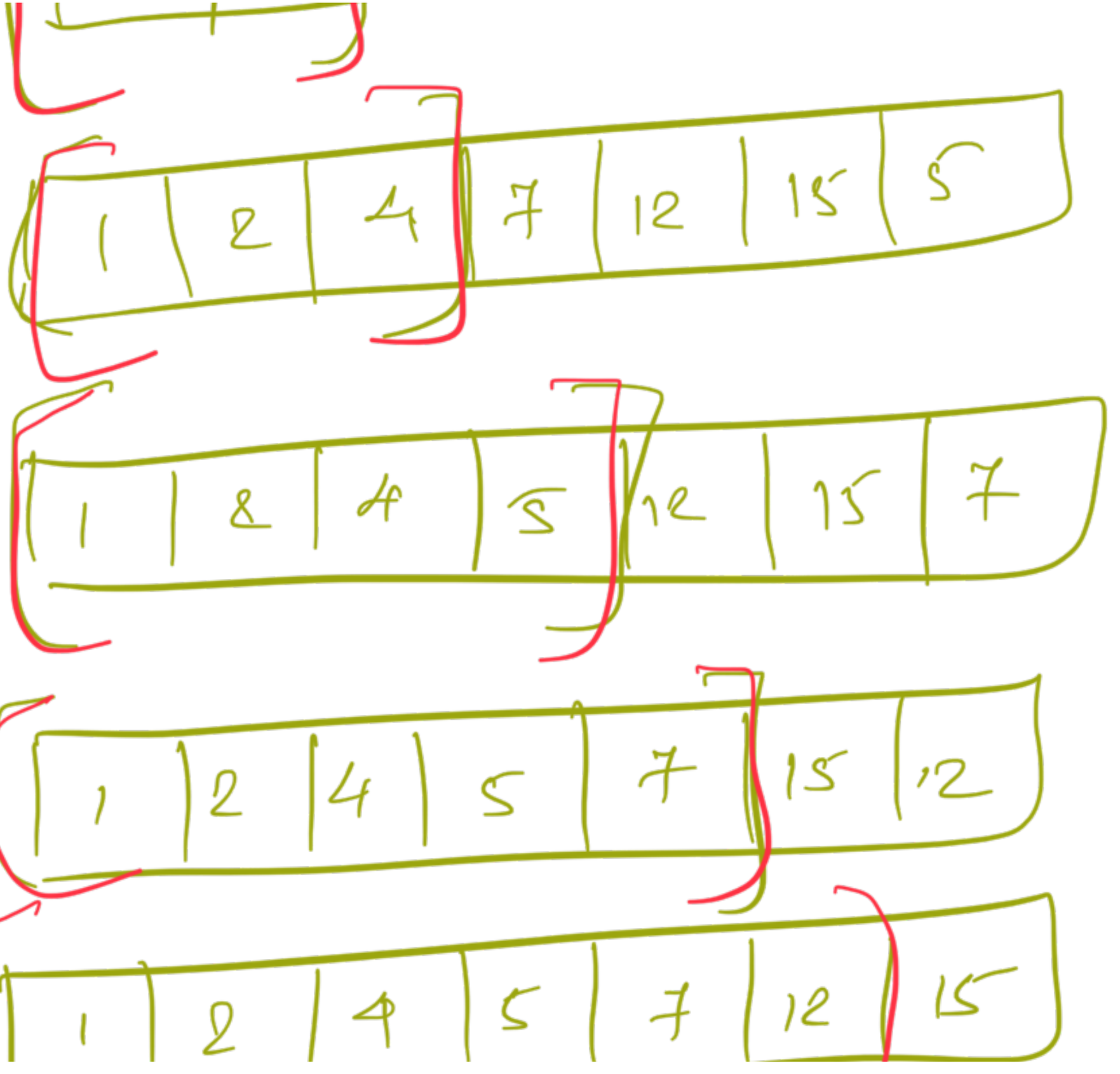
worst case time complexity $O(n^2)$

Algo:

1. Set min = 0^{th} index element
2. Search minimum element in the list
3. Swap with value at location min
4. Increase min to point to next element
5. Repeat until till is sorted

find minimum element




1 2 4 7 12 15 5
1 2 4 5 12 15 7
1 2 4 5 7 15 12
1 2 4 5 7 12 15

(4) Insertion sort

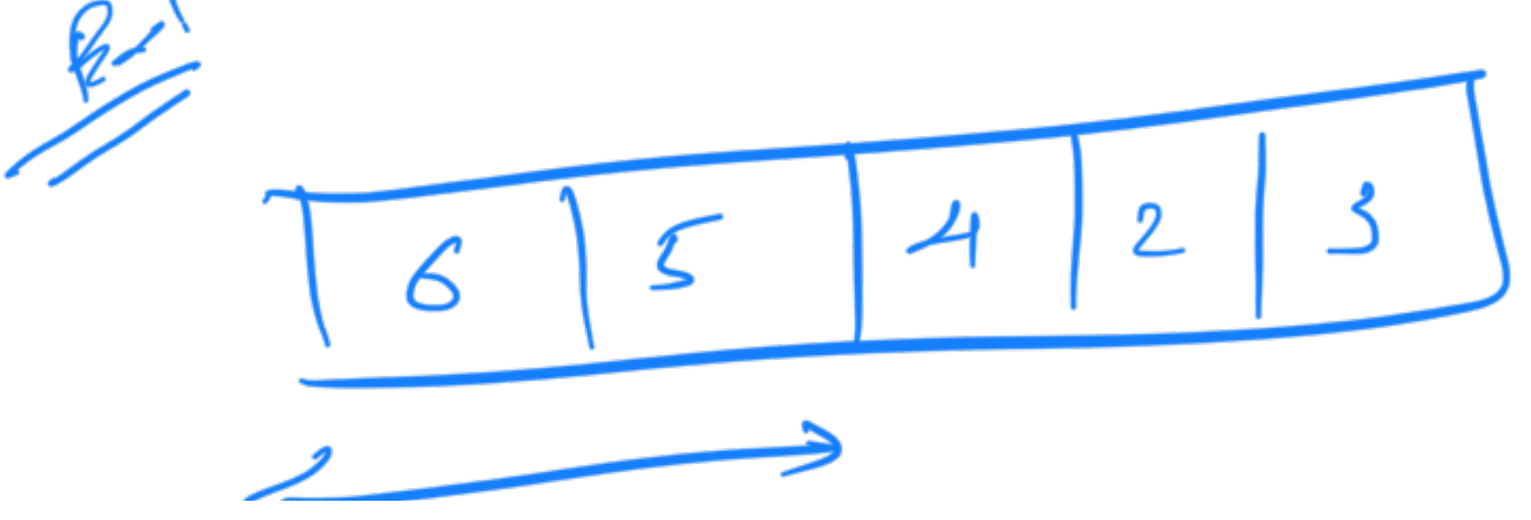
→ A sublist is [or sorted Array] is maintained which is always sorted

→ Not for large Array

→ time complexity $O(n^2)$

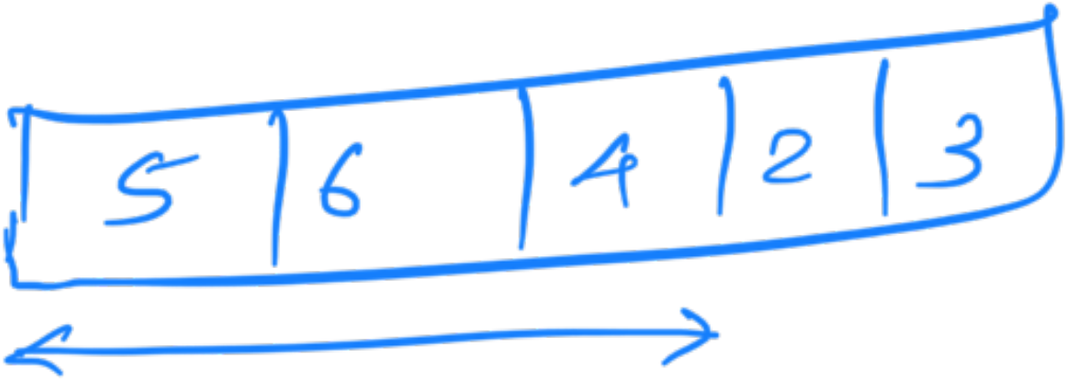
→ $(n-1)$ pass are req. to sort n elements.

→ In each pass we insert current element at appropriate place so that element in current range are in order

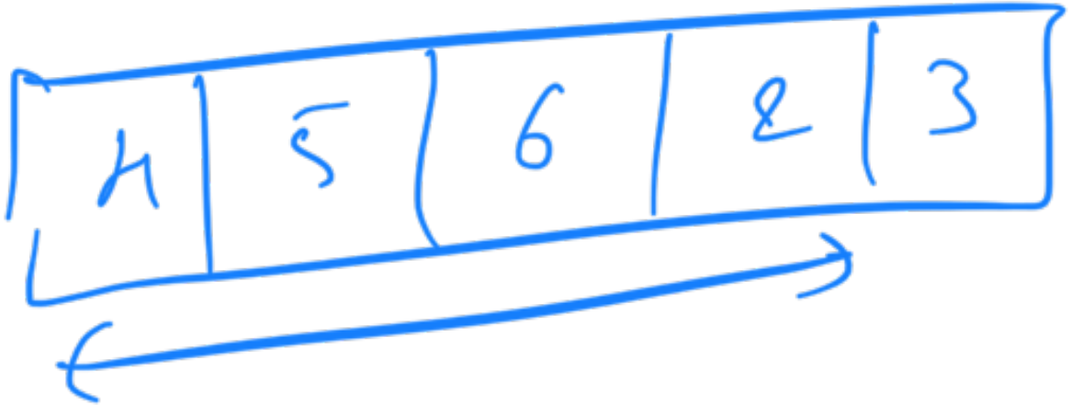


R

pass : 1

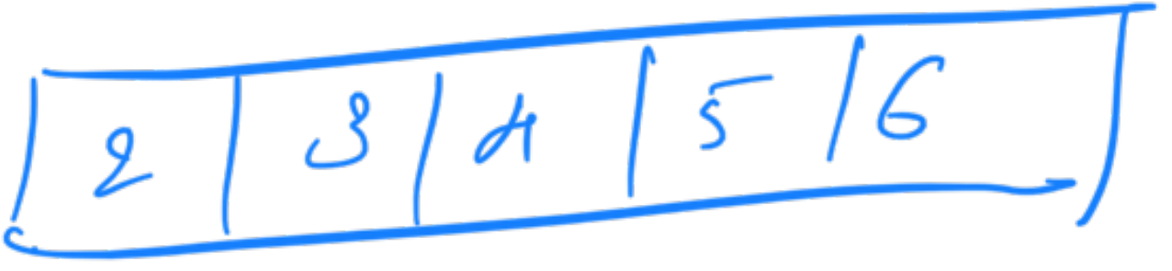


pass : 2



pass 3:

pass 1:



(#) Merge Sort :-

- Based on Devide & Conquer
- time complexity $O(n \log n)$
- It devides the array into halves & then merge them into sorted

manner.

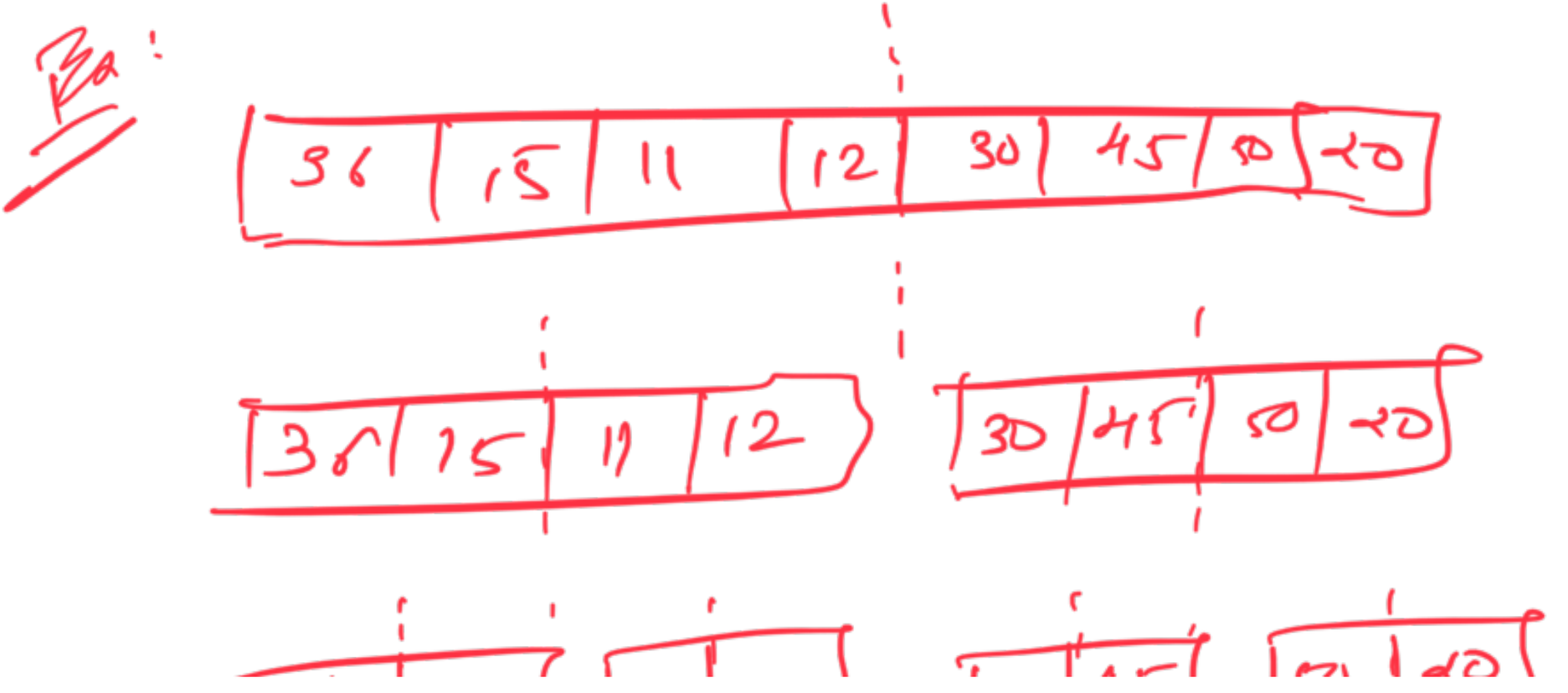
Algo: Beg : start of Array

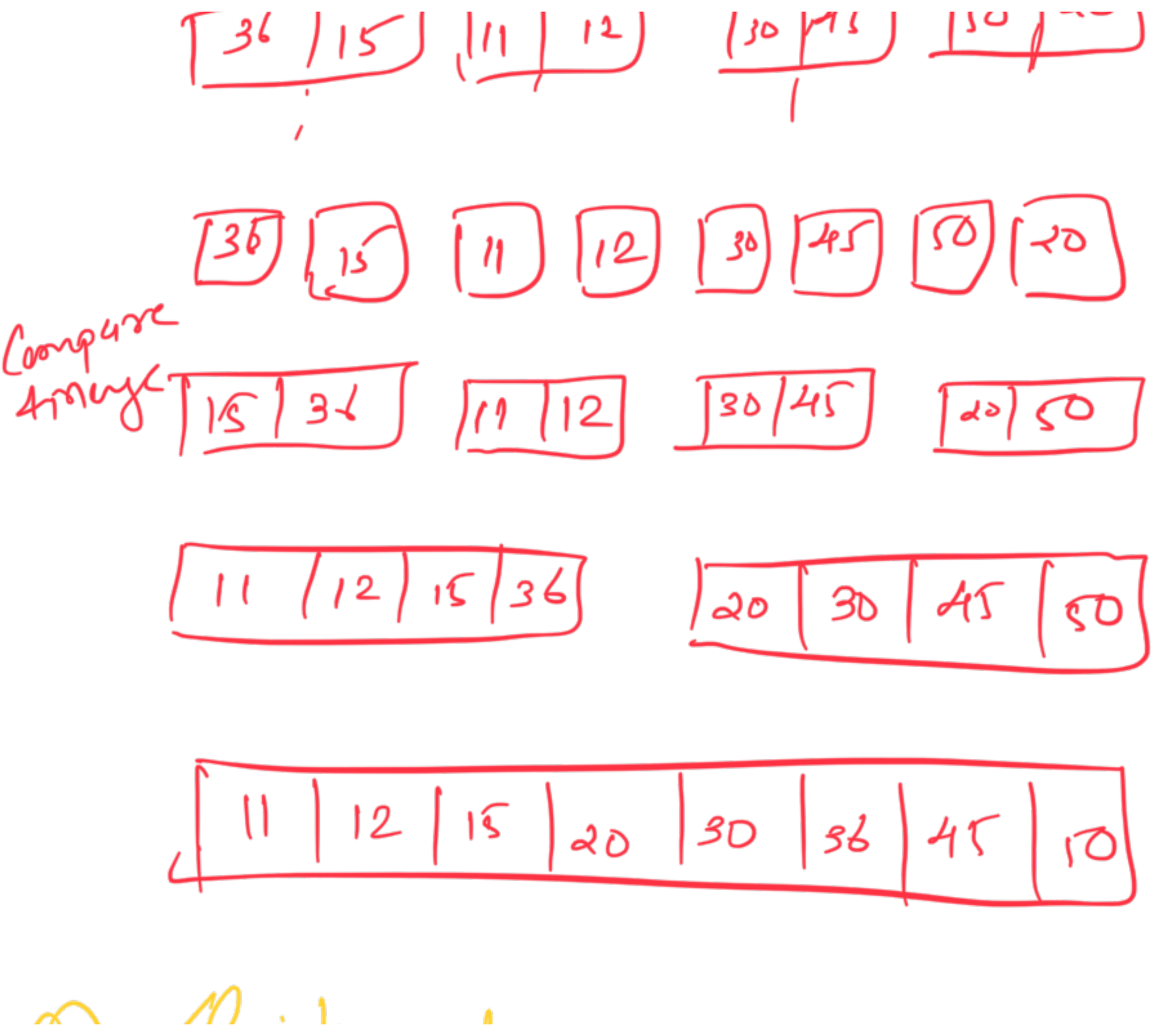
end : End of Array

if (beg < end)

then mid = $\left(\frac{beg+end}{2}\right)$

Ex:




36 15 11 12 30 45 50 20
Compare
15 36 11 12 30 45 20 50
11 12 15 36 20 30 45 50
11 12 15 20 30 36 45 50

(4) Quick sort

→ Based on partioning of array of data in smaller array

→ divide & conquer

→ Efficient for large size data

O(nlogn)

left
45 33 11 55 77 90 40 60 99 22 88 66
pivot
j
Right to left scan
→ if smaller then pivot then exchange
i
left to Right scan
→ if Bigger then pivot swap

44 33 11 55 77 90 40 60 70 22 88 66
22 33 11 55 77 90 40 60 70 44 88 66
22 33 11 44 77 90 40 60 90 55 88 66
22 33 11 40 77 90 44 60 90 55 88 66
22 33 11 40 44 90 77 60 90 55 88 66
sublist
sublist

11 33 22 40
11 22 33 40
90 77 60 90 55 88 66
66 77 60 90 55 88 90
66 77 60 90 55 88 90
66 77 60 88 55 90 90
61 77 60 88 55
55 77 60 88 66
55 66 60 88 77

⑧ Heap Sorting

→ The sorting is done by first creating a heap tree from the given sorting if max heap

array 4 then

— min Heap

— max Heap

↳ parent Heap > child Heap

50
10
20
30
heapify
50
10
20
10
40
heapify
40
20
10
30
60
heapify
60

Ex:

60	30	15	20	25

max heap tree

→ Interchange the latest element with parent Node

heapify

25
15
20

Insert el. interchange

20
25
15
30
20 | 25 | 15 | 30 | 10

interchange

Verify
25
20
15
25 | 20 | 15 | 30 | 60

interchange
15
20
25
15 | 20 | 25 | 30 | 40
latest element
interchange
heapify
15
20
15 | 20 | 25 | 30 | 40
heapify
20

⑥ Hashing (Searching)

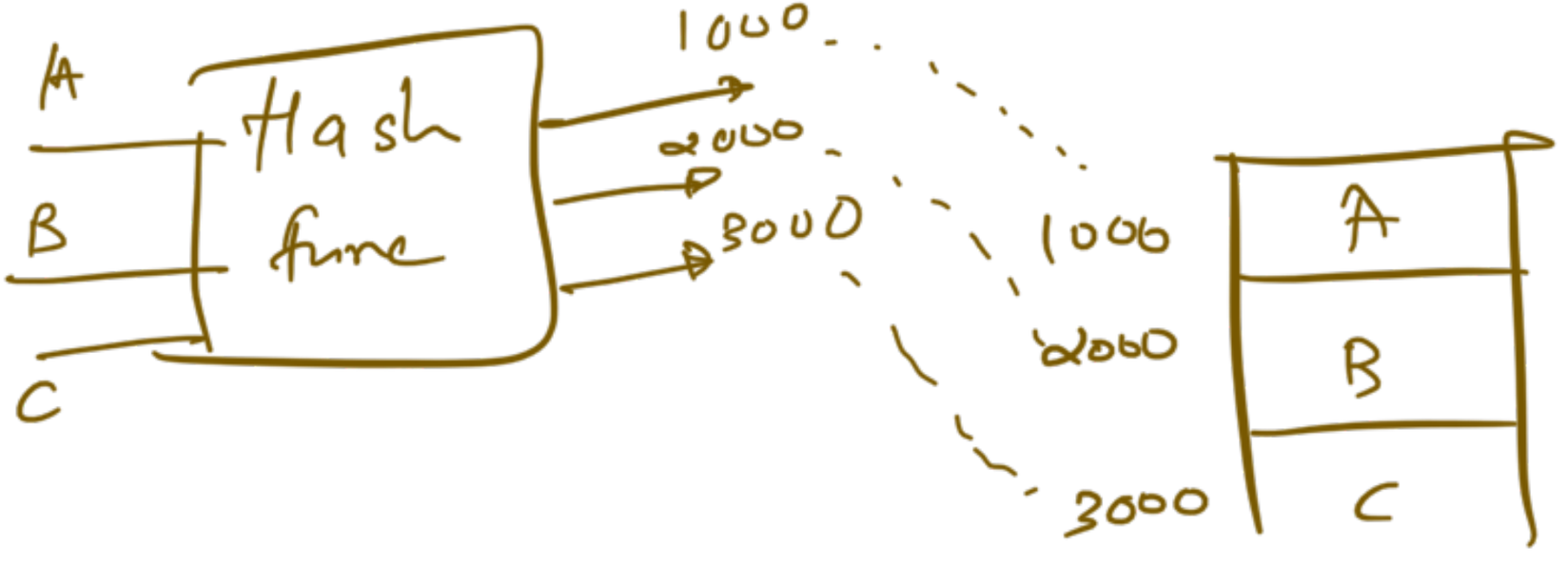
- Hashing involves less key comparisons

→ Hashing [illegible]

and searching can be performed in constant time

→ The goal of hashed function is to find target data in only one test

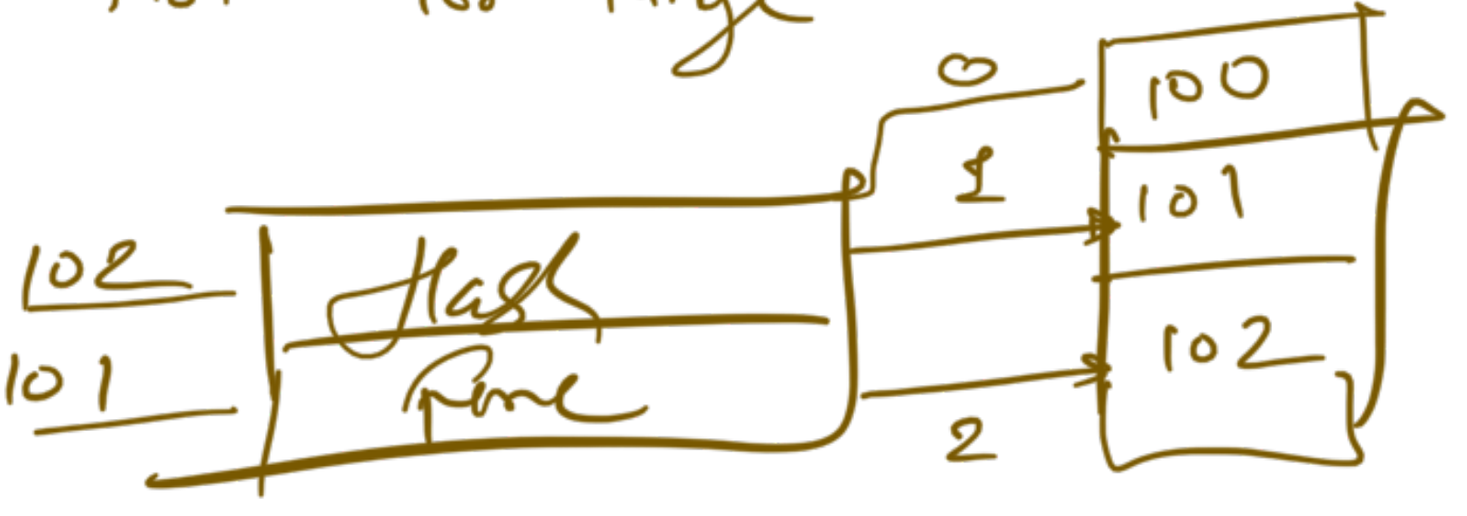
O(1)



⑧ Types of Hash function

① Direct Hashing

- No Algorithm manipulation
- Min No. of collision
- limited No. of keys
- Not for large



(2) Modulo-Division

→ Division remainder
→ works with any list size
→ if (list size) is prime no.
then few collision

$$h(k) = k \% n$$

(10) if $n = 10$

$$\begin{cases} 101 \% 10 = 1 \\ 100 \% 10 = 0 \end{cases}$$

(11) $111 \% 10 = 1$

→ collision

(3) Mid-Square hashing

→ Middle of Square

→ Square the key and the middle of the required No. of address will be the Address

If $k = 35$

$k^2 = 1225$

$n = 2$

$h(k) = 22$

(4) Folding Hashing

Fold Shift Hashing

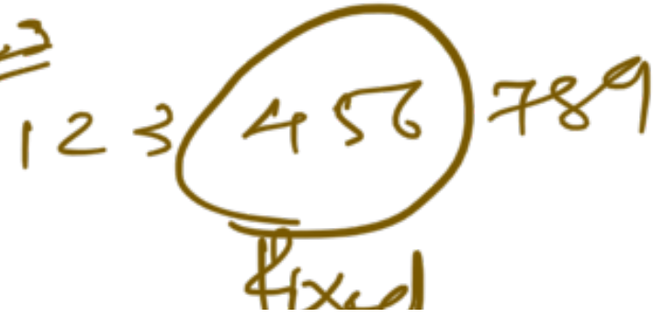
→ Key value is divided into parts and then added to req. address

eg 123 456 789

Fold Boundary Hashing

left & Right nos are folded on a fixed boundary I/O

eg



123
456
789

If n=3

~~123~~

321
987
456

n=3

~~123~~ 368

h(k1) = 368

~~1~~764

h(k1) = 764

(3) Subtraction Hashing

Subtraction a fix no. from key

h(k1) = k - C

- No collision
- for small size

$h(k) = k - c$

$c = 1000 - 1000$
$2 = 1002 - 1000$
$10 = 1010 - 1000$

(4) Collision Resolution

↳ A collision occurs when a hashing algorithm produces an address for a key and that address for a key is already occupied

address is already

Eg $h(k) = k \% n$

$n = 10$

$h(k) = k \% 10$

$h(101) = 101 \% 10$

$= 1$

collision

$h(111) = 111 \% 10$

$= 1$

collision Resolution

- linear Probe
- Quadratic Probe
- Pseudo Random

$$y = ax + c$$

$c \rightarrow$ const

$x \rightarrow$ key

$$h(k) = y \% n$$

(#) Open Addressing

→ Here area address are searched for an open or unoccupied element where new data can be placed

⇒ linear probe

$h(k) = k \% n$

$h(101) = 101 \% 10 = 1$

$h(111) = 111 \% 10 = 1$

$i = 1$

$i = (i + k) \% n$

$k = 1, 2, 3, 4$

④ Quadratic Probe

$h(k) = k \% n$

$h(101) = 101 \% 10 = 1$

$h(111) = 111 \% 10 = 1$

$i = (i + k^2) \% n$

$k = 1, 2, 3$

$5 \% 10 = 2$

(II) Linke list