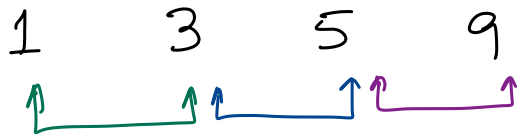
Best case and worse case

Best case: Data is already sorted.

Worse case: Data is sorted in reverse order.

	Time Complexity		Space Complexity
	Best Case	Worse Case	
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort (optimised)	$O(n)$	$O(n^2)$	$O(1)$

Bubble Sort → Best case scenario.



swap Count = 0

bubble Sort (arr) → optimised.

- n = number of elements to sort.

- while (n > 1) do

  - left = 0
  - → swapCount = 0
  - while (left < (n-1)) do

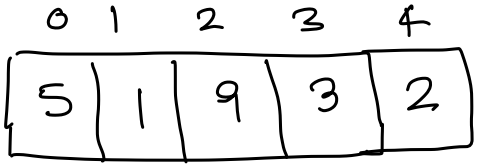
    - if (arr [left] > arr [left + 1]) then
      - swap Count = swapCount + 1
      - Swap elements at left and (left + 1).

    - left = left

  - if (swapCount = 0) then STOP.
  - n = n - 1

Insertion Sort

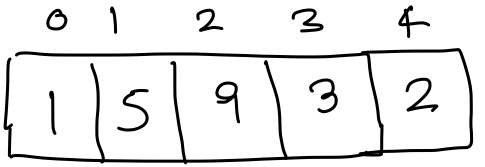
Given sorted elements, insert new element such that all elements remain sorted.



$n = 5$

Sorted array

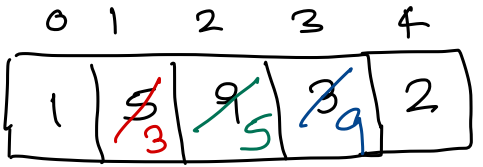
insert (1)



$n = 5$

Sorted array

insert (9)



$n = 5$

Sorted array

sortedSize = 3

insert (3)

insertElement = arr [sortedSize] = 3

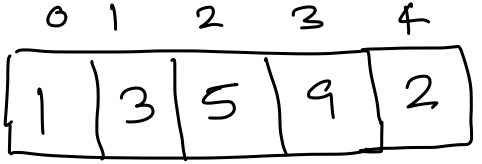
i = sortedSize - 1 = ~~2~~ ~~1~~ 0

if (arr [i] > insertElement)

arr [i+1] = arr [i]

Looking for first element that is smaller that new element to be inserted.

New element will be added to the right of it.



n = 5

Sorted Array

sortedSize = 4

Insertion Sort (arr)

- sortedSize = 1
- while (sortedSize < n) do
  - insertElement = arr [sortedSize]
  - i = sortedSize - 1
  - while (i >= 0) do
    - if (arr [i] > insertElement)
      - arr [i+1] = arr [i]
    - else
      - end the lwp.
  - arr [i+1] = insertElement.
  - sortedSize = sortedSize + 1
- Stop.

Space Complexity = O(1)

Time Complexity = $O(n^2)$

Worse case = $O(n^2)$

Best case (Data already sorted)

= O(n)

sorted Size	i	#
1	0	1
2	1 0	2
3	2 1 0	3
:	:	:
(n-1)	(n-2) .. 1 0	n

1+2+3+.. n

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

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

0	1	2	3	4
1	3	5	9	13

n=5

Best case

sorted Size	i
1	0
2	1
3	2
:	:
(n-1)	(n-2)

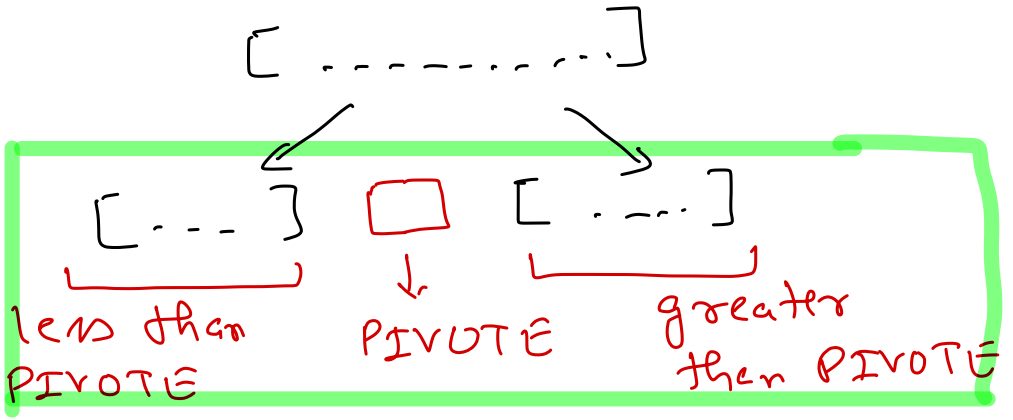
⇒ (n-1)

sorted Size = ~~1~~ 2

insert Element = ~~3~~ 5

i → ~~0~~ 1

Quick Sort ⇒ Divide and Conquer



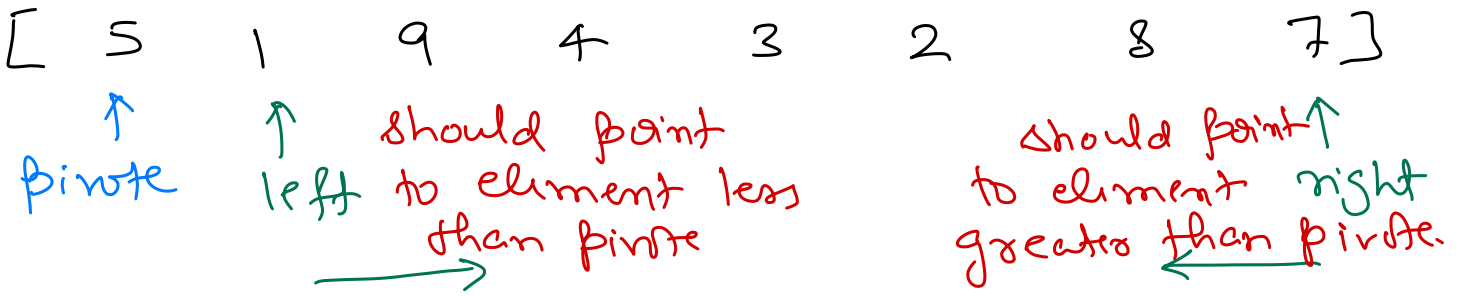
Partitioning.

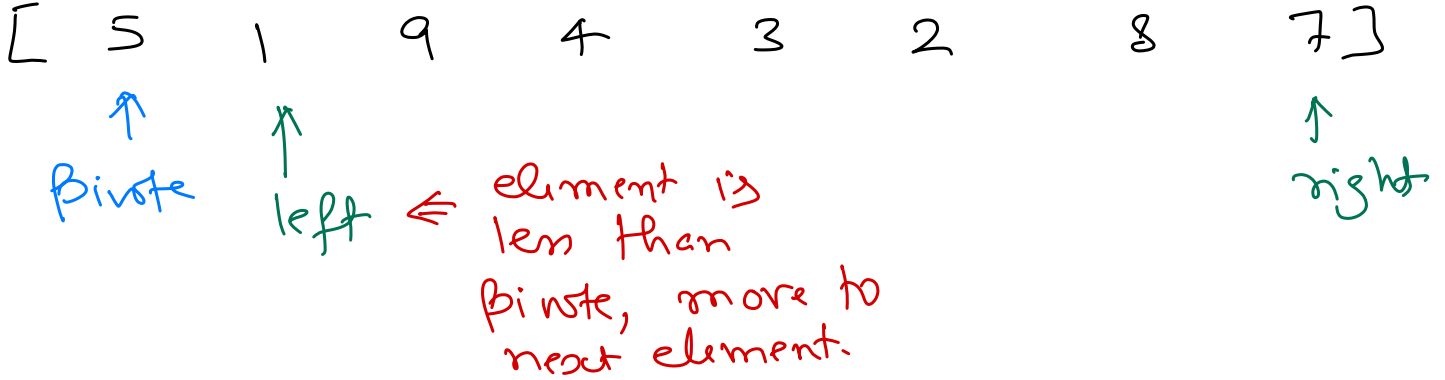
↓

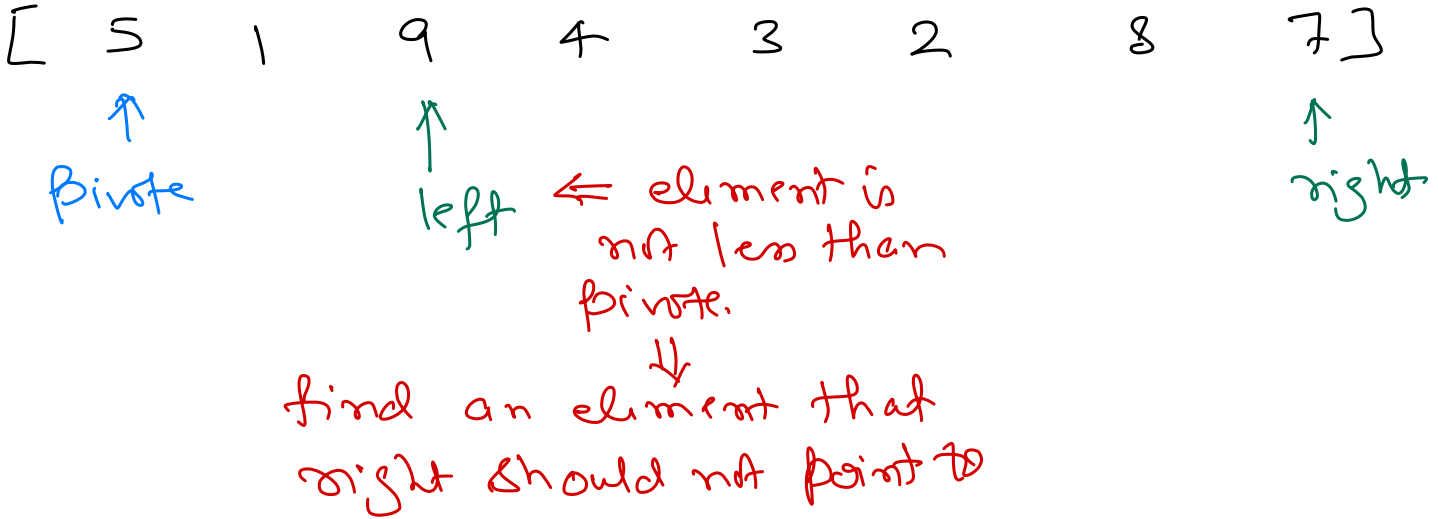
with respect to pivot element, we partition elements into two parts such that

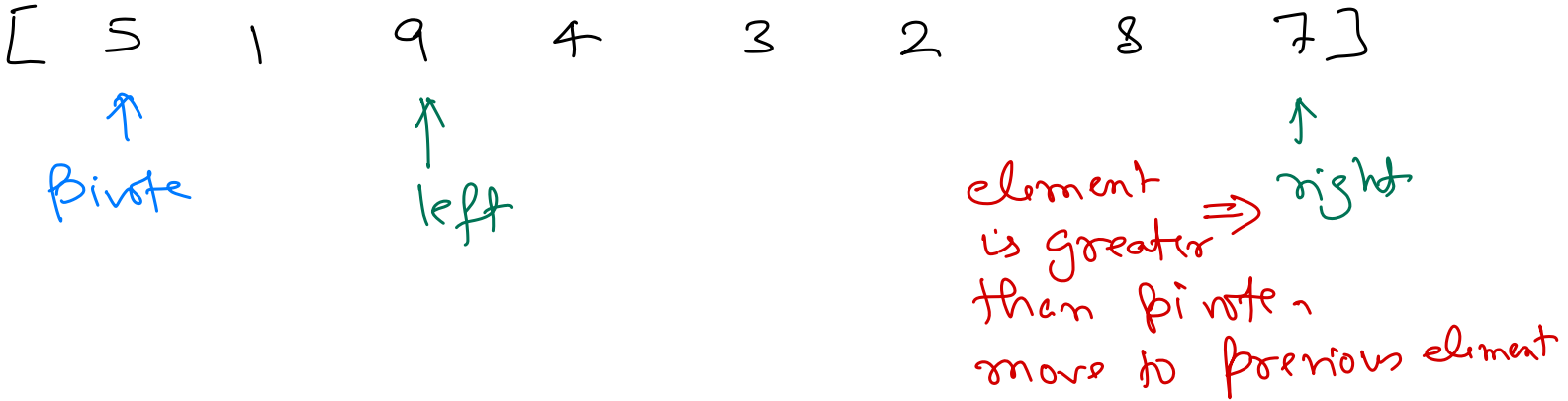
elements less than pivot are in left part and elements greater than pivot are in right part.

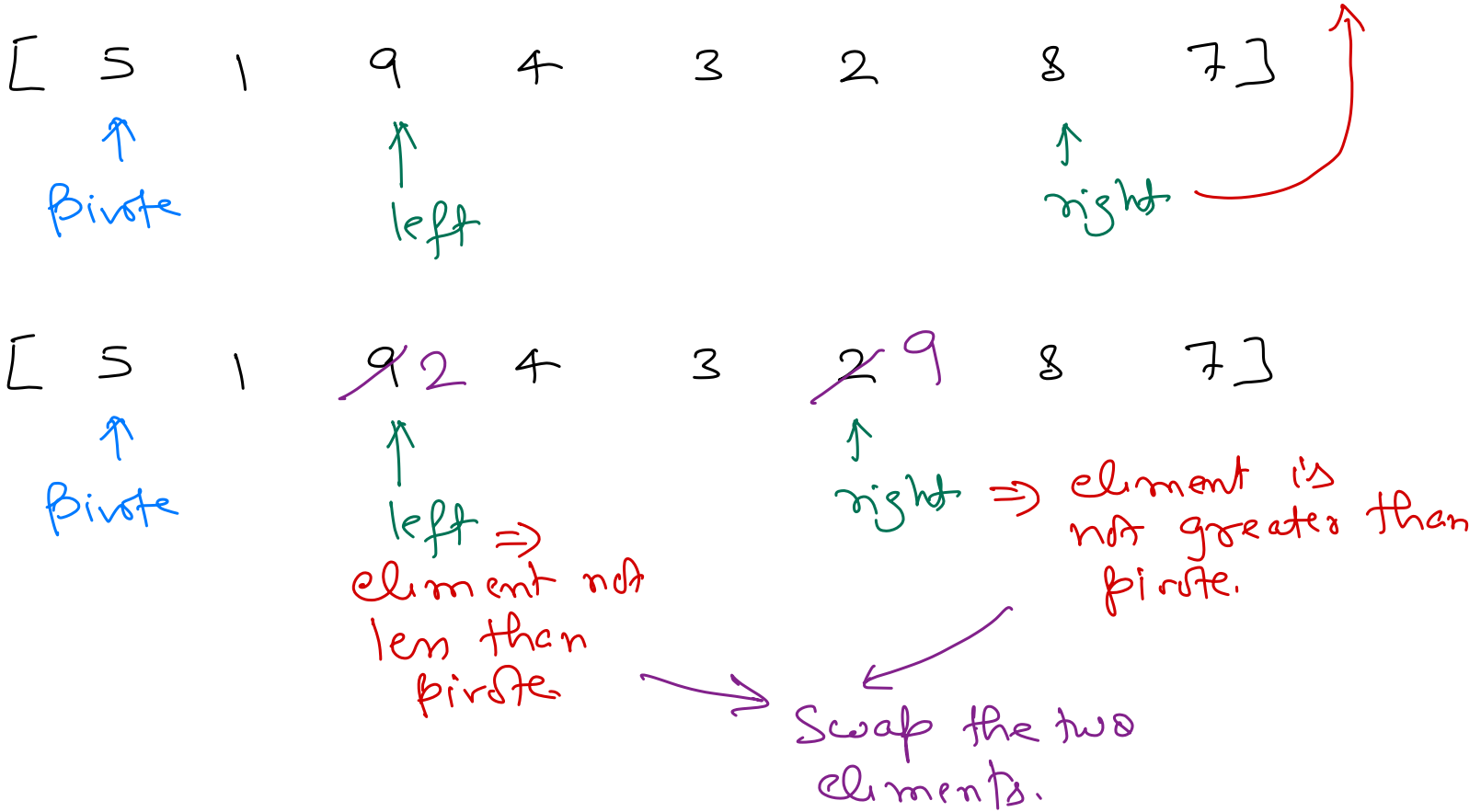
→ Sorting in ascending order.

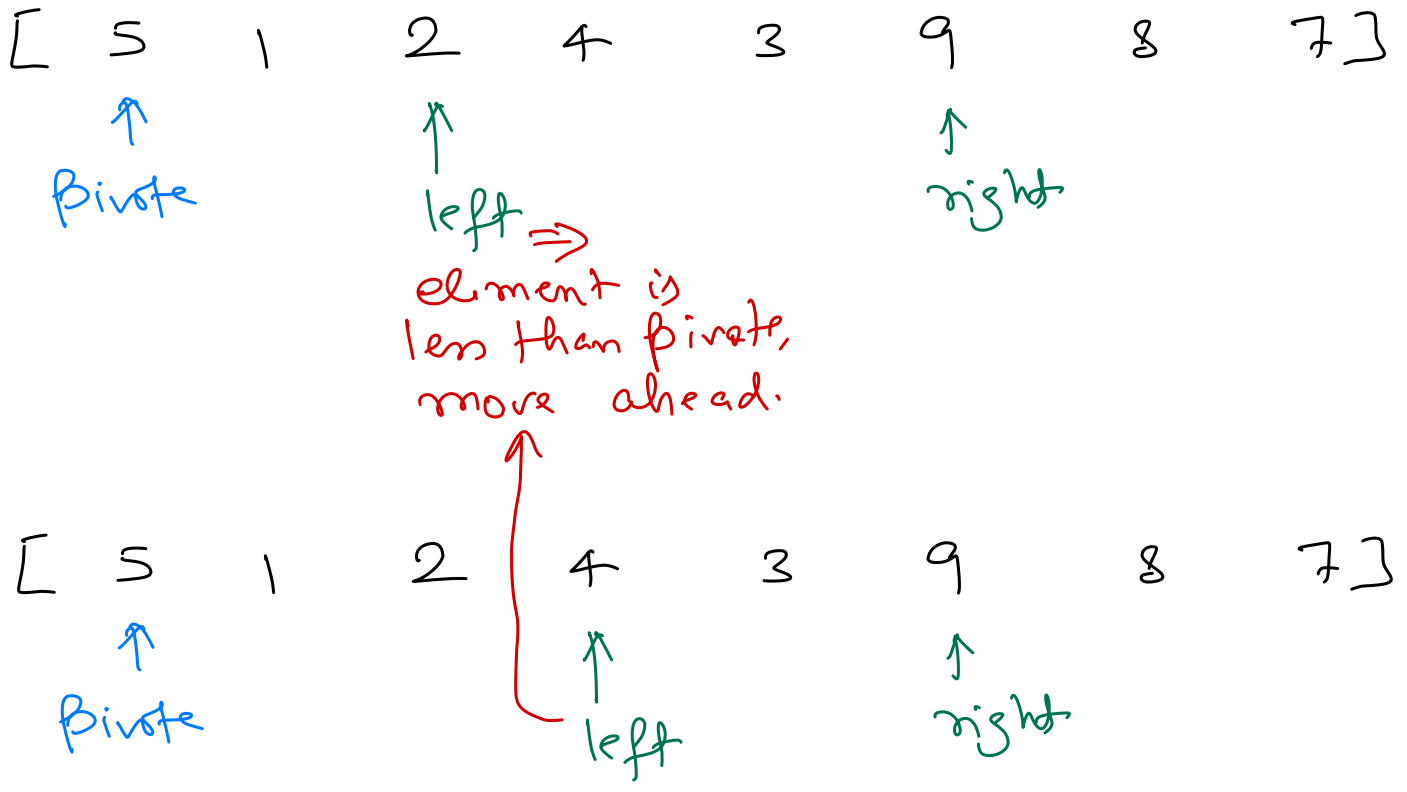

[5 1 9 4 3 2 8 7]
pivote
left
should point left to element less than pivote
should point to element right greater than pivote.

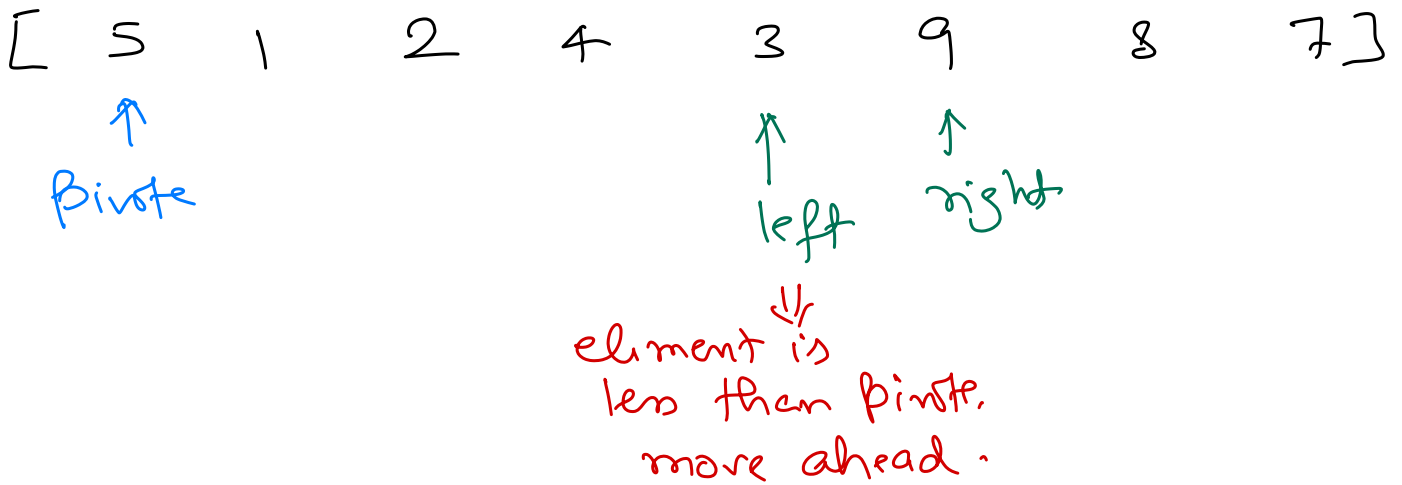

[5 1 9 4 3 2 8 7]
pivote
left
element is less than pivote, move to next element.
right

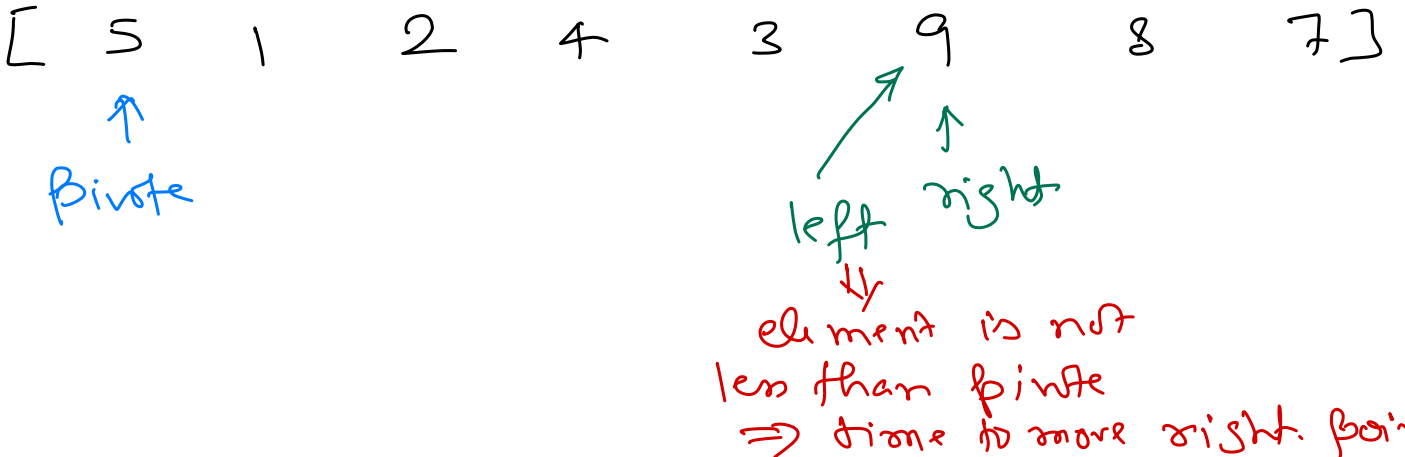

[5 1 9 4 3 2 8 7]
Pivote
left
right
⇐ element is not less than pivote.
⇓
find an element that right should not point to

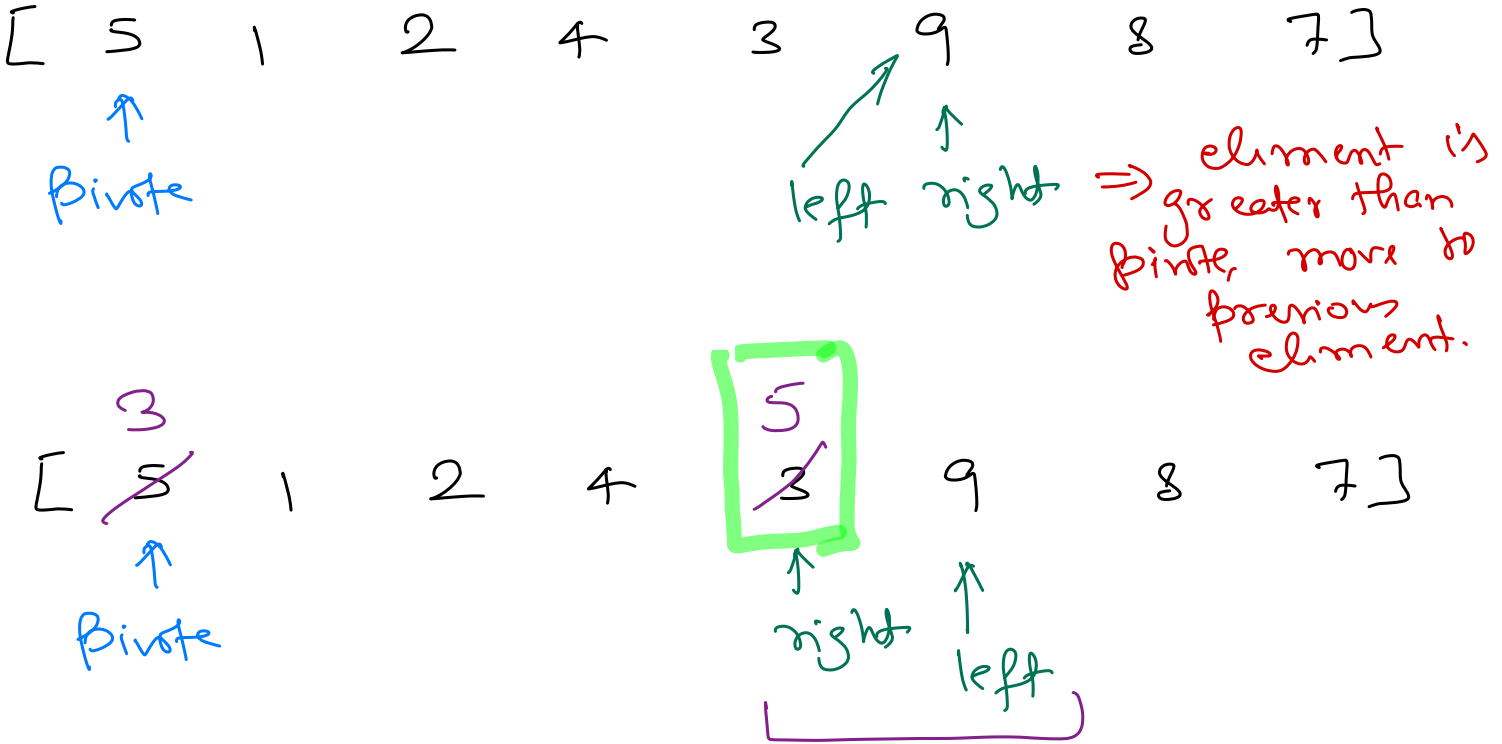

[5 1 9 4 3 2 8 7]
Pivote
left
right
element is greater than pivote ⇒
move to previous element


[5 1 9 4 3 2 8 7]
Pivote
left
right
[5 1 9 2 4 3 2 9 8 7]
Pivote
left ⇒ element not less than pivote
right ⇒ element is not greater than pivote.
Swap the two elements.

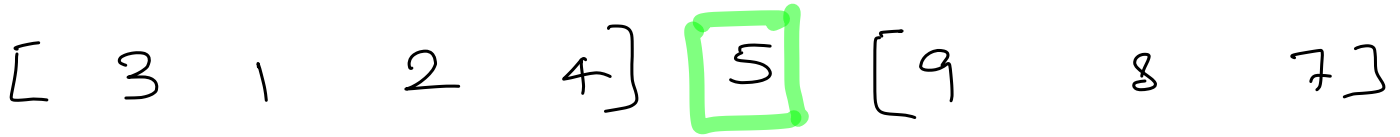

[5 1 2 4 3 9 8 7]
Pivot
left
right
element is less than pivot, move ahead.
[5 1 2 4 3 9 8 7]
Pivot
left
right


[5 1 2 4 3 9 8 7]
Pivote
left
right
element is
less than pivote.
move ahead.


[5 1 2 4 3 9 8 7]
Pivote
left
right
element is not
less than pivote
⇒ time to move right pointer.


[5 1 2 4 3 9 8 7]
Pivot
left
right
⇒ element is greater than pivot, move to previous element.
3
[5 1 2 4 5 3 9 8 7]
Pivot
right
left
left and right has crossed over → STOP.
→ Swap pivot and right element.

→ Partition around right.



int Partition(arr, startPos, endPos)
- Set pivotPos to startPos
- Set left to startPos + 1
- Set right to endPos
- while (left <= right)
  - while ((left <= right) and (arr[left] < arr[pivotPos]))
    - Increment left by 1
  - while ((left <= right) and (arr[pivotPos] < arr[right]))
    - Decrement right by 1
  - if (left < right)
    - Swap elements at left and right position
- Swap element at pivotPos and right
- Return right // After swap, pivot element at right

Space = O(1)

⇒ O(n)

↓

number of elements in [startPos, endPos]

QuickSort(arr, startPos, endPos)
// base case = array sorted (number of elements <= 1).
- if (endPos - startPos + 1) <= 1
  - Stop

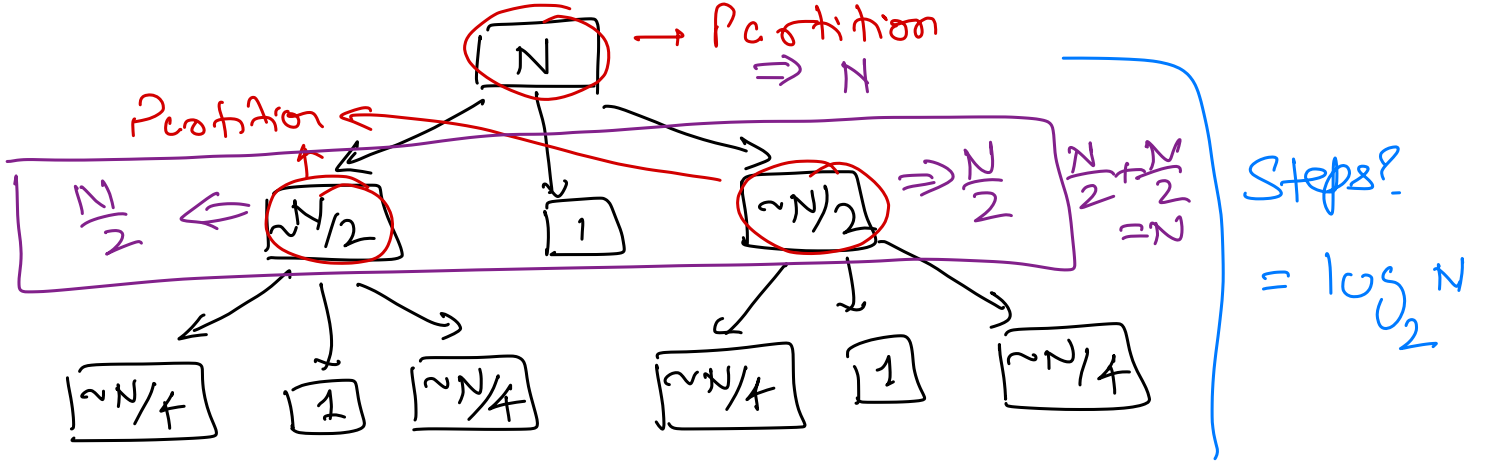
Space = O(h)

h ↓

$\log_2 n$

// Array too big - Partition them into two parts around pivot and sort each of them.
- pivotPos = Partition(arr, startPos, endPos)
- QuickSort(arr, startPos, pivotPos - 1)
- QuickSort(arr, pivotPos + 1, endPos)
- Stop

Time complexity of Quick Sort



$\log_2 N$ number of steps.

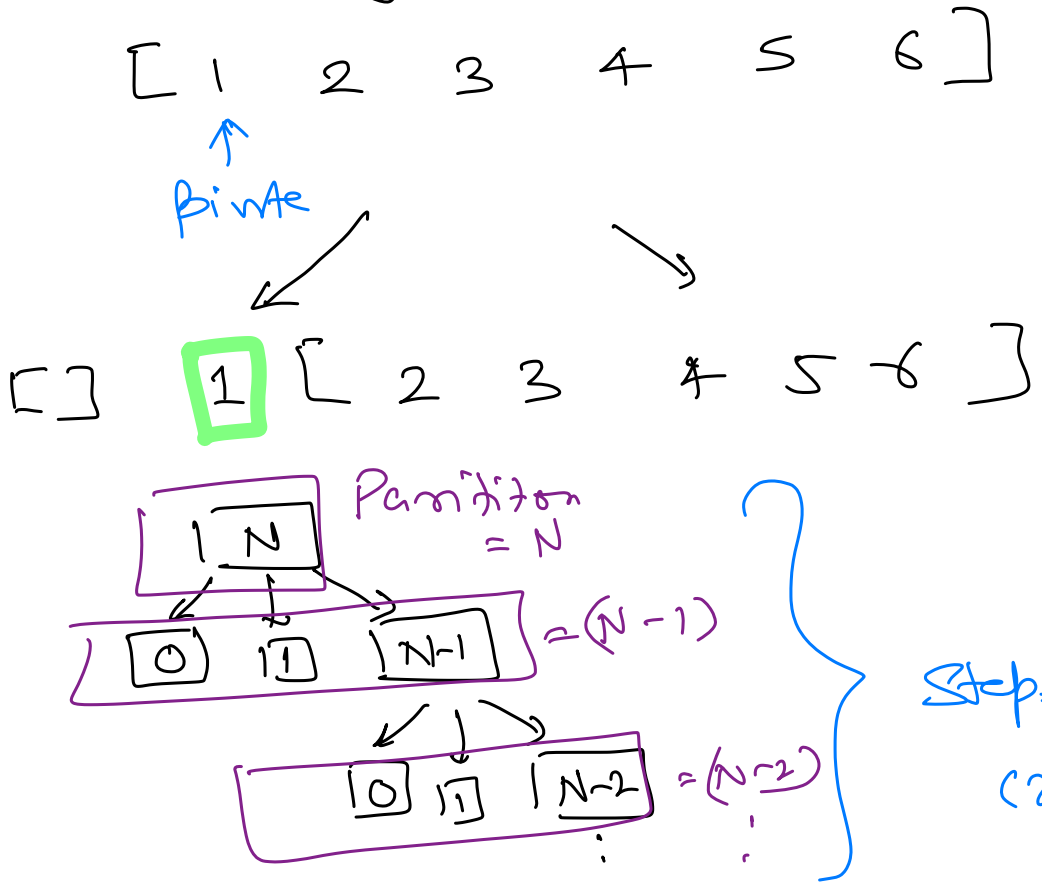
At each step, total work done by partitioning = N

Time complexity = $O(n \log_2 n)$

Space complexity = $O(\log_2 n)$

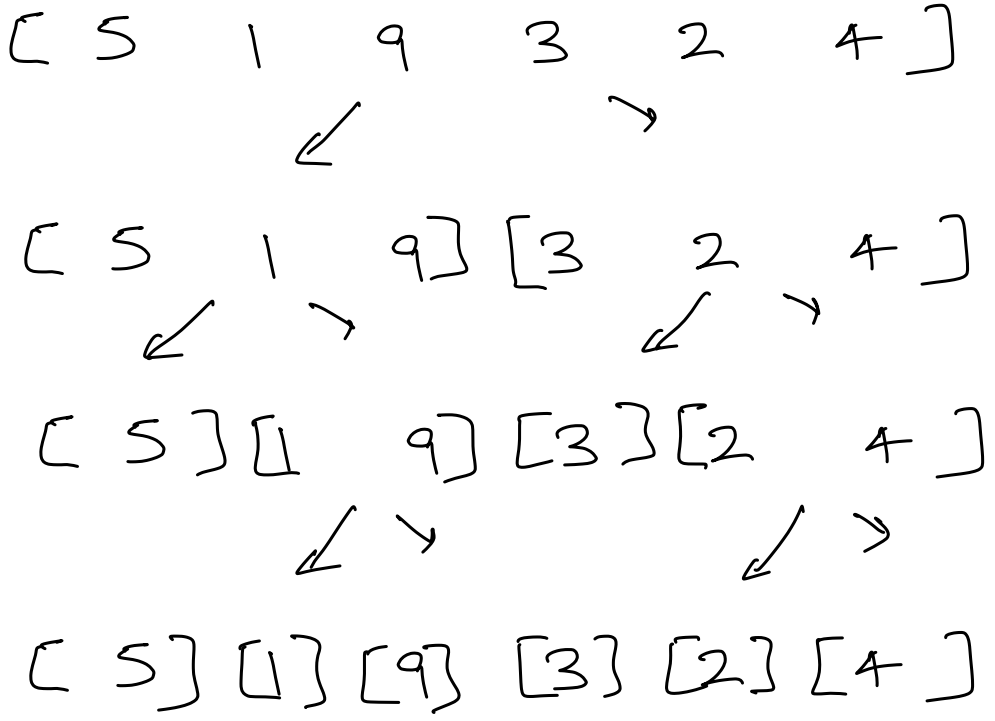
Best & worse case → when partitioning divides into two equal parts ⇒ $O(n \log n)$

Data is already sorted. ← worse case scenario



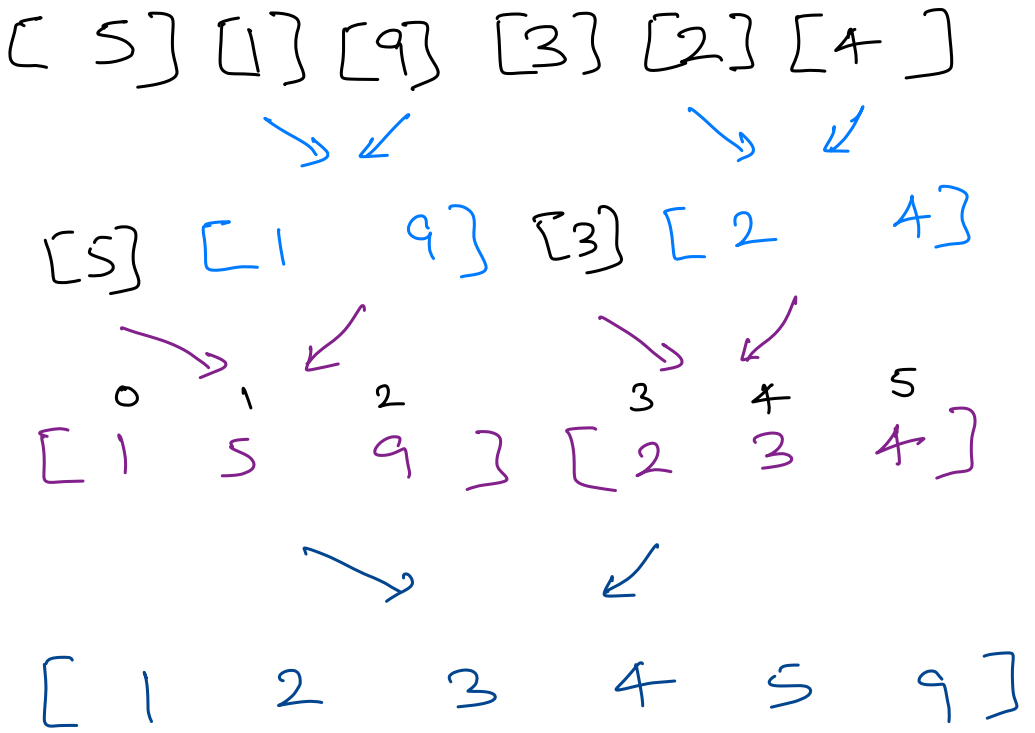

1
Time = O(n²)

Merge Sort ⇒ Divide and Conquer



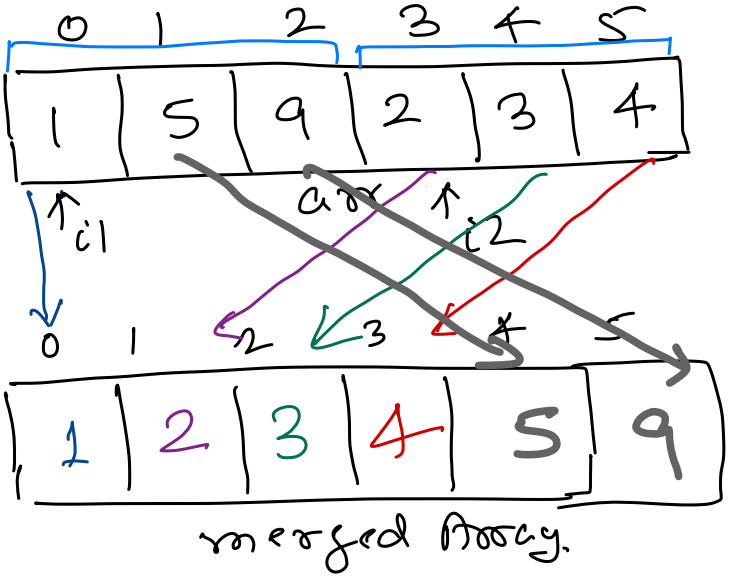
① Divide input into two smaller parts until the smaller parts are sorted.

② Merge two sorted parts into one bigger sorted part.



In place merging ⇒ do not use extra space.

Out of place merging ⇒ uses extra memory.



startPos1 → 0

endPos1 → 2

i1 → ~~0~~ 1

r → ~~0~~ ~~1~~ ~~2~~ 3

startPos2 → 3

endPos2 → 5

i2 → ~~3~~ ~~4~~ ~~5~~ 6

MergeSort(arr, startPos, endPos)
// base case = array sorted (number of elements <= 1).
- if (endPos - startPos + 1) <= 1
  - Stop

// Array too big - Divide them into two parts and sort each of them.
- middlePos = (startPos + endPos) / 2
- MergeSort(arr, startPos, middlePos)
- MergeSort(arr, middlePos + 1, endPos)

// We got two sorted arrays, merge them into one
- Merge(arr, startPos, middlePos, middlePos + 1, endPos)
- Stop

Space = ?

$\log_2 N$ due to recursion.

Exercise

Can be eliminated in iterative merge Sort.

Space O(n)

Merge(arr, startPos1, endPos1, startPos2, endPos2)
- Set i1 to startPos1
- Set i2 to startPos2
- Set r to 0
- while ((i1 <= endPos1) and (i2 <= endPos2))
  - if (arr[i1] < arr[i2])
    - mergedArray[r] = arr[i1]
    - Increment i1 by 1
  else
    - mergedArray[r] = arr[i2]
    - Increment i2 by 1
  - Increment r by 1

// Copy remaining elements from other sorted array into mergedArray
- while (i1 <= endPos1)
  - mergedArray[r] = arr[i1]
  - Increment i1 by 1
  - Increment r by 1
- while (i2 <= endPos2)
  - mergedArray[r] = arr[i2]
  - Increment i2 by 1
  - Increment r by 1

// Copy merged elements back to arr
- Set i to startPos1
- Set j to 0
- while (i < r)
  - arr[i] = mergedArray[j]
  - Increment i by 1
  - Increment j by 1
- Stop

⇒ Takes extra space for merged elements = number of elements

Check two elements in two sorted arrays and put right one in merged array.

→ only one loop will run.

→ $x + y$

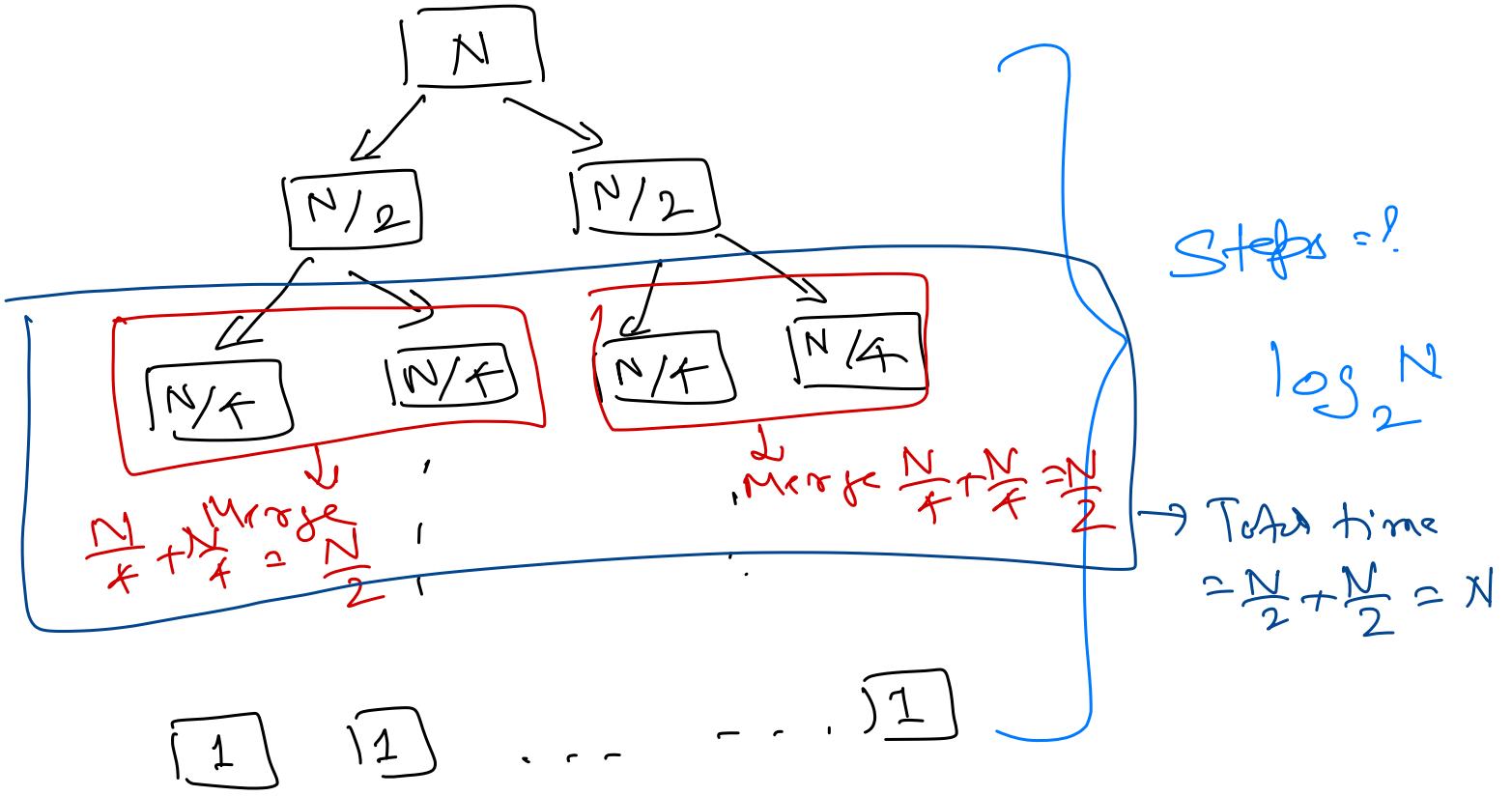
x → size of first array

y → size of second array.

$2(x+y)$

Time = $O(x+y)$

→ $x + y$



we have $\log_2 N$ number of levels.

At each level merge does N amount of work

Total = $n \times \log_2 n$

Time = $O(n \log_2 n)$

Best Worse case as well.

Space = $O(n)$

External Sorting => data to be sorted is stored on secondary storage.

e.g. External Merge Sort

m-way Merge Sort

Parallel merge sort