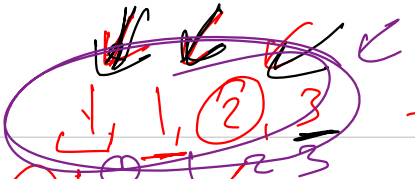
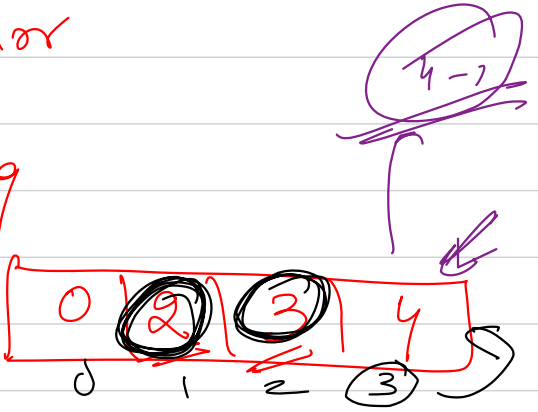
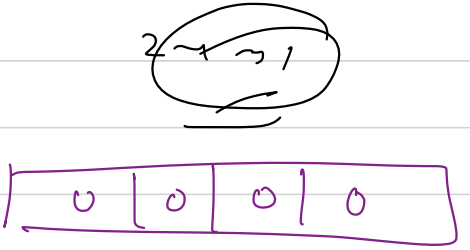


freq [arr [0]] += 1

freq [arr [1]] += 1

freq [arr [2]] += 1





output [freq [arr [3]] - 1] = arr [3]

output [freq [3] - 1] = arr [3]

output [4 - 1] = 3

```python
def counting_sort(arr):
    # in this implementation we will assume that we get only positive
    # Space complexity O(n+k)
    # Time complexity O(n + k)
    max_element = max(arr)
    frequency = [0]*(max_element+1)

    # this loop will help to create the frequency list
    for i in range(0, len(arr)):
        frequency[arr[i]] += 1

    # now we need to prepare prefix sum
    for i in range(1, len(frequency)):
        frequency[i] = frequency[i] + frequency[i-1]

    # frequency[arr[i]] -> the position of the current value arr[i]

    output = [0]*len(arr)
    for i in range(len(arr)-1, -1, -1): # going from previous
        output[frequency[arr[i]] - 1] = arr[i]
        frequency[arr[i]] -= 1

    return output
```

3, 1', 2, 1''

max_element = 3

freq = [0, 0, 2, 3]

i → [0, 3]

i = 0 ✗ 2 3

i → [1, 3]

i = 1 2 3

i = [3, 0]

i = 3 2 1 0

output = [1', 1'', 2, 3]

Final sorted list

disadvantage → if range of elements is very high

then it performs poorly

Merge Sort → Comparison based sort

First of all forget about sorting :

Q:) Given two sorted lists of size n and m, Merge these 2 sorted lists into one new sorted list & return the final list :

→ [1, 5, 7, 9]
[2, 3, 10]

$n, m \leq 10^5$

Ans → [1, 2, 3, 5, 7, 9, 10] ← new list

reduced list

a → [1, 5, 7, 9] → n = 4

b → [2, 5, 10] → m = 3

reduced

output [1, 2, 3, 5, 7, 9, 10] (n+m)

0 1 2 3 4 5 6

if op[k] = min(a[i], b[j])

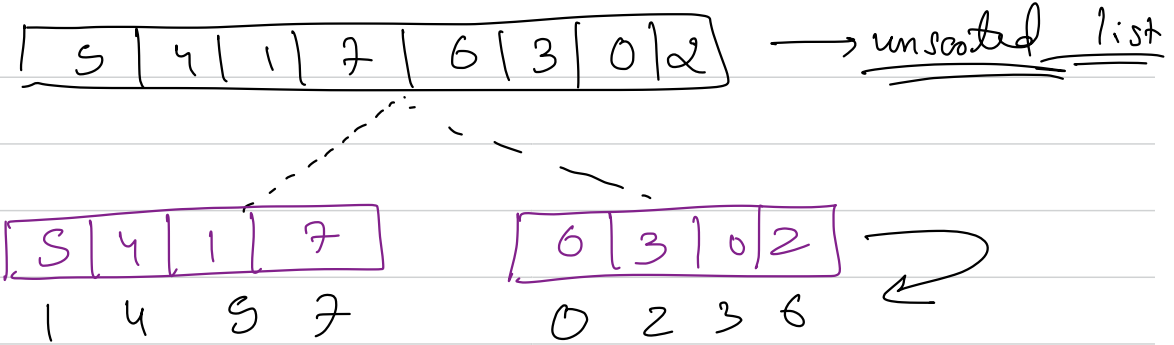
k++ i++ j++

O(n) → [1, 5, 7, 9]

[2, 3, 10]

of [0 0 0 0 0 0 0] n+m

Merge Sort

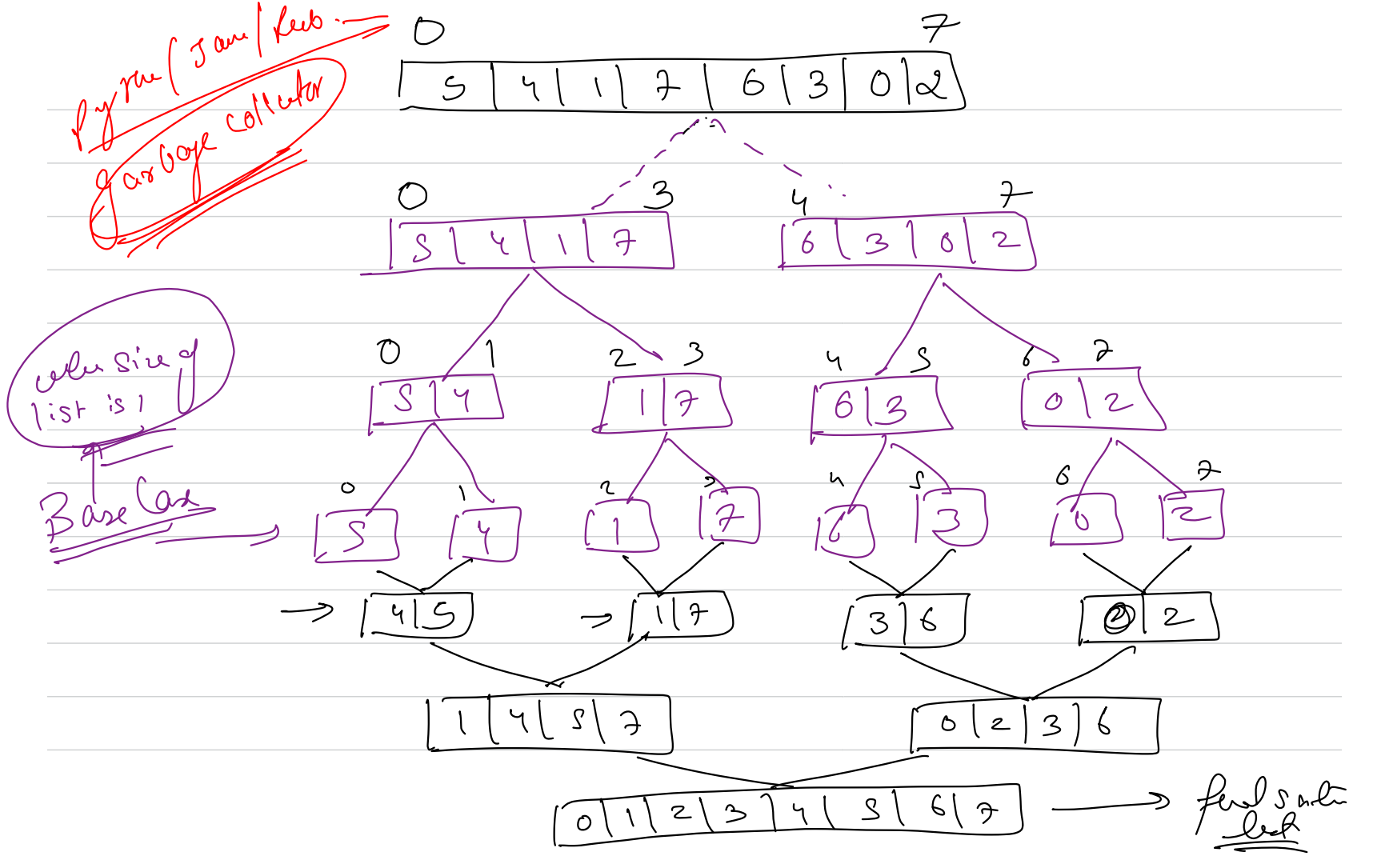


→ Divide Your List into 2 equal halves

→ Sort the left half recursively.

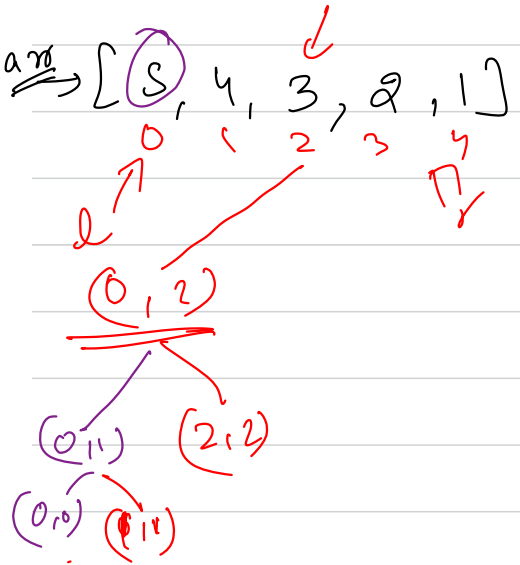
→ Sort the right half recursively

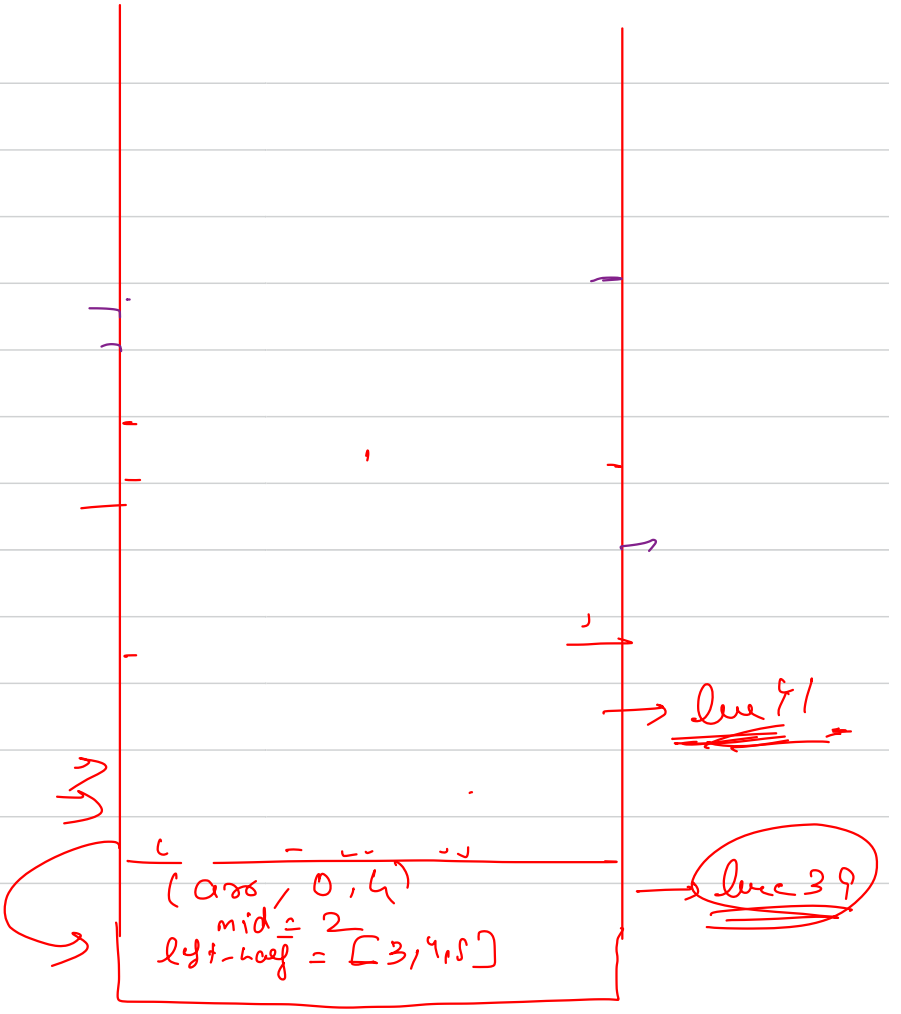
→ Selfwork → merge 2 sorted halves


Python / Java / Ruby
Garbage collector
0
7
5 4 1 7 6 3 0 2
0
3
4
7
5 4 1 7
6 3 0 2
0 1 2 3 4 5 6 7
when size of list is 1
Base Case
4 5
1 7
3 6
0 2
1 4 5 7
0 2 3 6
0 1 2 3 4 5 6 7
final sorted list

```python
def merge_sort(arr, left, right):
    if(left == right):
        # base case
        return [arr[left]] # or return [arr[right]]

    mid = (left + right) // 2
    left_half = merge_sort(arr, left, mid) # recursivel
    right_half = merge_sort(arr, mid+1, right) # recurs
    output = merge(left_half, right_half) # merged both
    return output
```





$$T(n) = \underbrace{T\left(\frac{n}{2}\right)}_{\text{left}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{right}} + \underbrace{O(n)}_{\text{merge}}$$

$T(n)$ is a function which calculates no. of operations to apply merge sort on a list of size n

no of operations reqd to sort a list of $n/2$ size

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

2° $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ → recurrence relation

$-2 \times T\left(\frac{n}{2}\right) = 4T\left(\frac{n}{4}\right) + 2O\left(\frac{n}{2}\right)$

$-4 \times T\left(\frac{n}{4}\right) = 8T\left(\frac{n}{8}\right) + 4O\left(\frac{n}{4}\right)$

K term

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

$2^{K-1}\ T(n) = 2^{K}T(1) + 2^{K-1}O\left(\frac{n}{2^{K-1}}\right)$

$T(n) = O(n) + O(n) + O(n) \cdots\cdots O(n)$

K times

$T(n) = K \times O(n)$

$K = \log_2 n$

$T(n) = O(n \log_2 n)$

$n \to \frac{n}{2} \to \frac{n}{4} \cdots \frac{n}{2^k}$

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

$K = \log_2 n$

Space Complexity

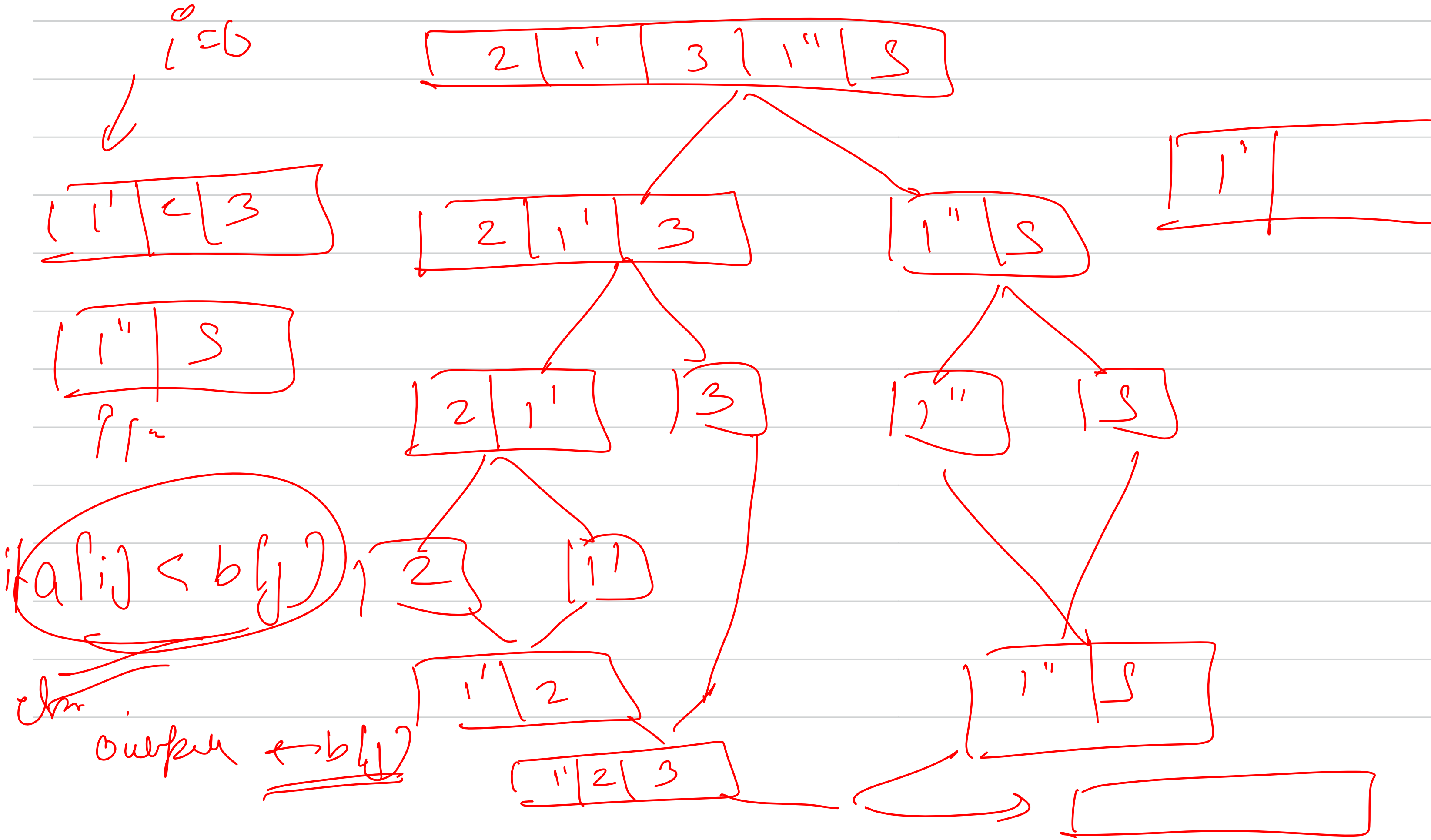
$O(n + \log_2 n) \to O(n)$

Inplace → No

Swaps → Not available

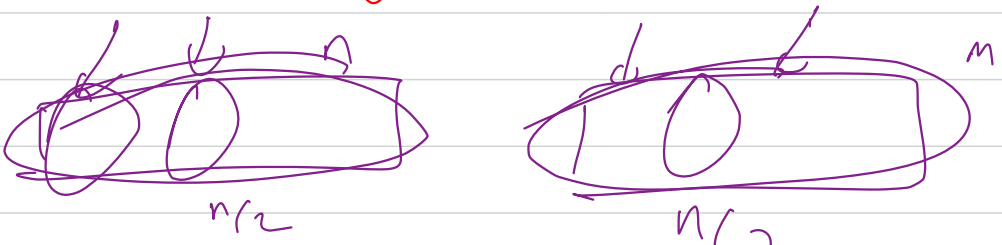
| 1 | 1 | 2 | 2 |

Stable → Yes

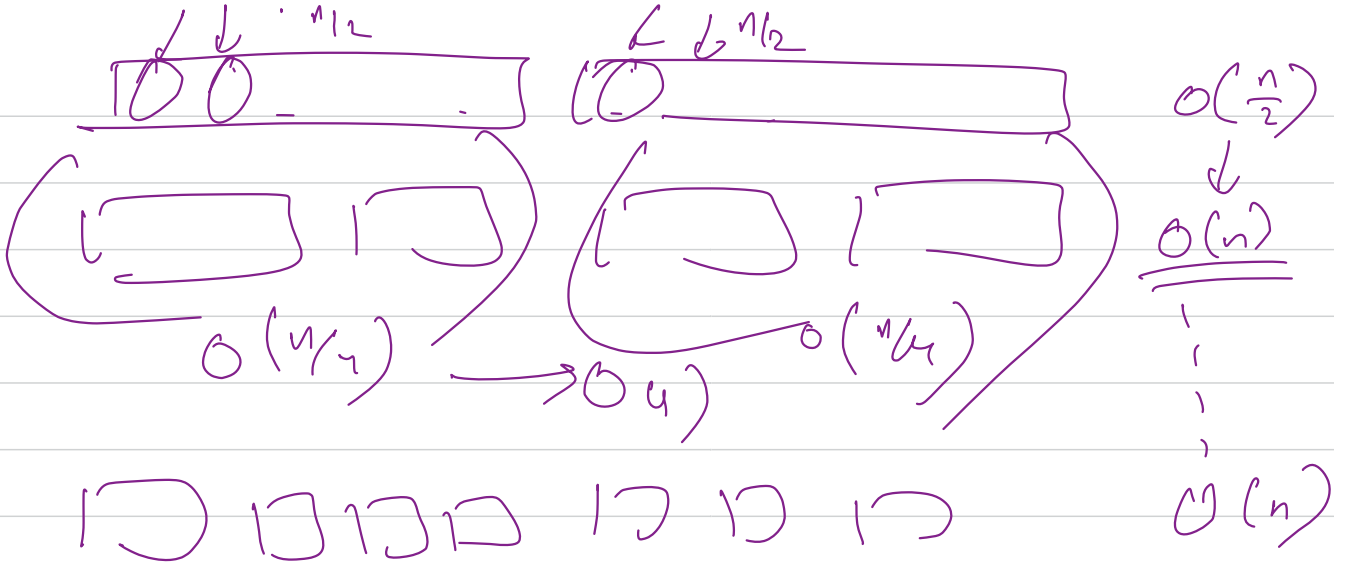


i=0
if(a[i] < b[j])
else
output ← b[j]

No. of Comparison of merge Sort



$O(n \log n)$



$O(n \log n)$