Quick Sort

n-elements

```
mergeSort(int arr[], int low, int high)
{
    if(low<high)
    {
        int mid=low+(high-low)/2;
        mergeSort(arr,low,mid);
        mergeSort(arr,mid+1,high);
        merge(arr,low,mid,high);
    }
}
```

mergeSort(int arr[], int low, int high) ⟶ T(n)

① mergeSort(arr,low,mid); ⟶ T(n/2) — n/2 elements

② mergeSort(arr,mid+1,high); ⟶ T(n/2) — n/2 elements

merge(arr,low,mid,high); ⟶ n

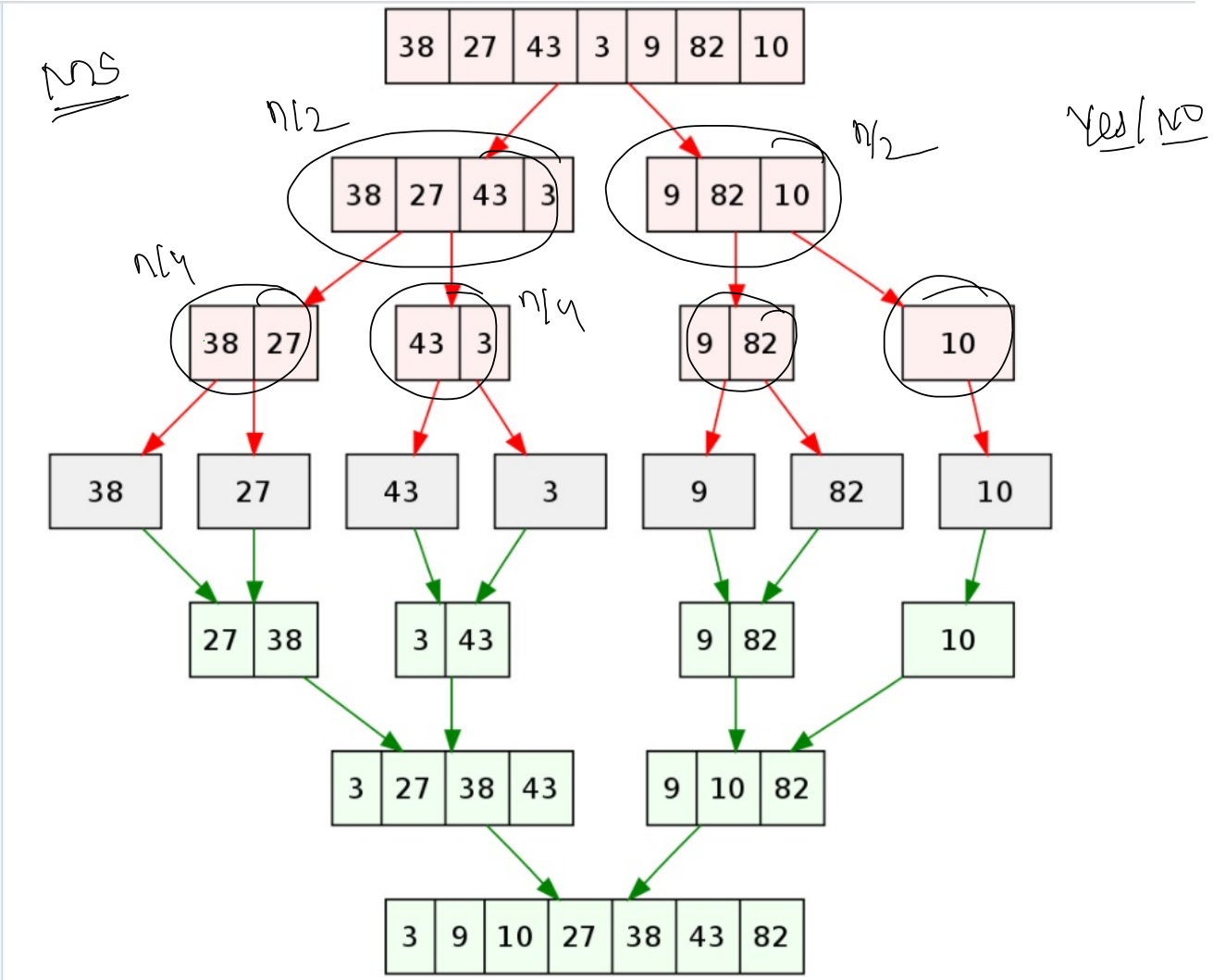
H/W

$$T(n) = \begin{cases} 2T(n/2) + n & ; n \geq 2 \\ 1 & ; n == 1 \end{cases}$$

$O(n \log_2 n)$

T(n) : time take to sort array of n elements using merge sort

T(1) = O(1) // constant time


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

merge: /

↑ i/p: 2 sorted arrays

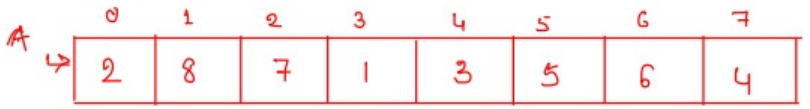
o/p :- final sorted array

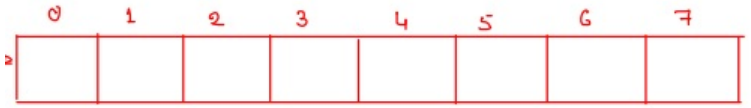
Quick Sort

↳ ? ✓ "partition procedure"

↳ we are having 2 x Algorithms (diff given by 2 diff people)

Partition Procedure





① Lomuto partition scheme

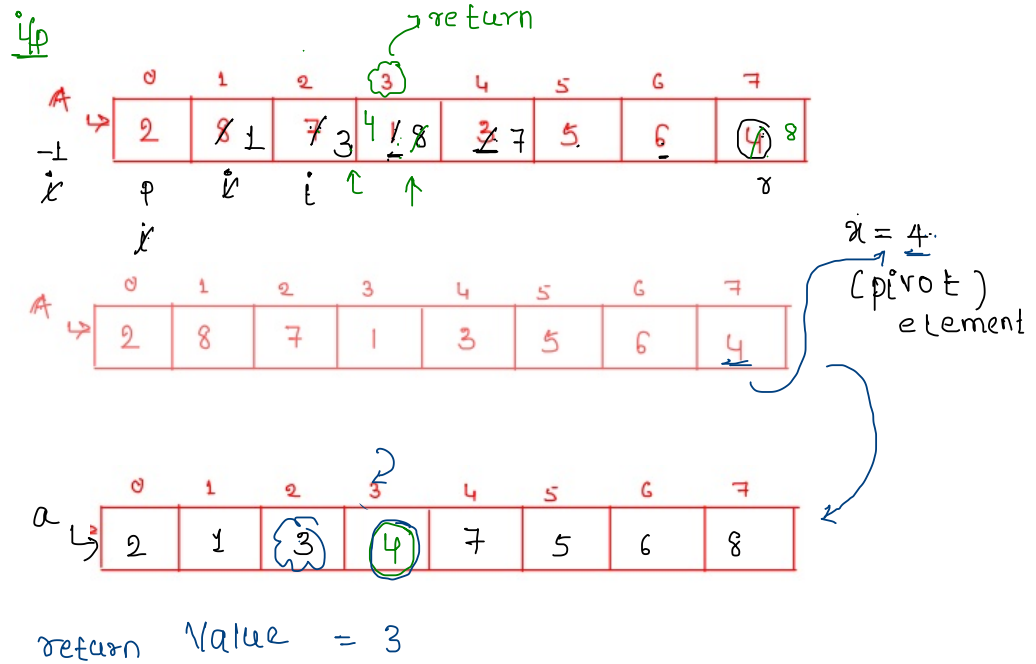
1) i++

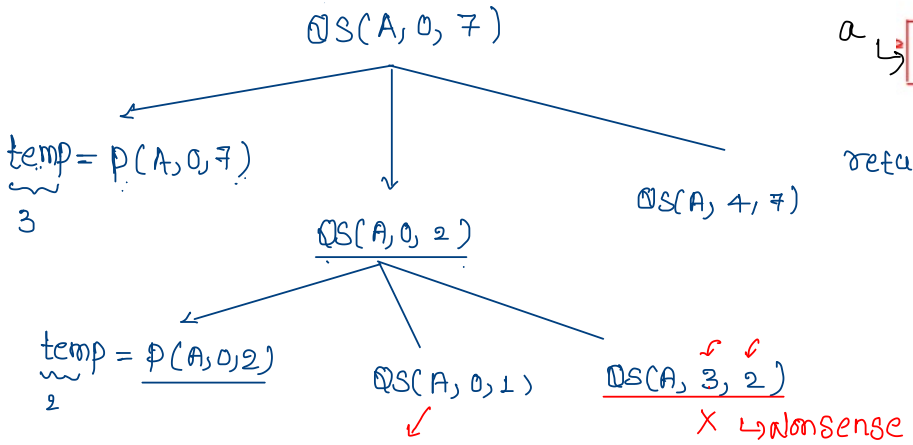
2) swap(A[i], A[j])

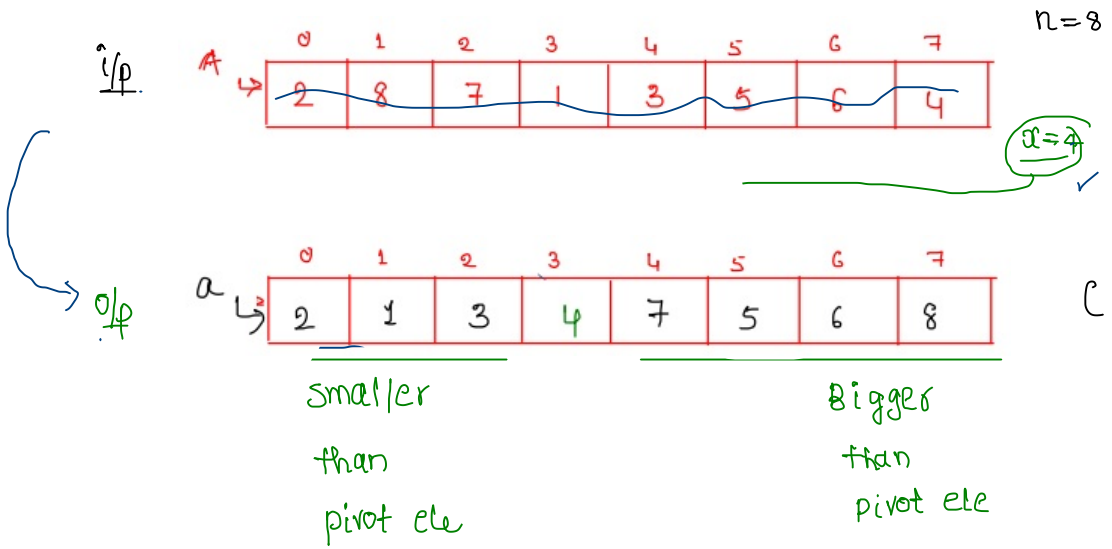
Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p..r]$ in place.

```
PARTITION(A, p, r)                    (last ele index)
1  x = A[r]                           (x: pivot)
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]    (swap)
7  exchange A[i + 1] with A[r]
8  return i + 1
```







(After completion of 1 partition procedure)

* ~~index~~ of pivot on its exact position ✓

lesser than pivot are at left and greater than pivot are at right

biggest element is at the last

3 is also at right position

4is come at right position

smaller elements in the first half and greater elements at the second half

all elements smaller than 4 are on left and greater elements on right

All the elements smaller than the pivot are at its left

```
main()
{
        // read input array : Arr

        // n is array length

        quickSort(Arr,0,n-1);

quickSort(int Arr[],int low,int high)
{
        if(low<high)
        {
                temp=partition(Arr,low,high);
                quickSort(Arr,low,temp-1);
                quickSort(Arr,temp+1,high);
        }
}
```

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p..r]$ in place.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

→ O(n)

→ Place one ele @ Correct position

```
main()
{
        input array reading
        QuickSort(A,0,n-1);
}

function QuickSort(A,beg,end)
{
        if(beg<=end)
        {
        temp=PARTITION(A,beg,end)
        QuickSort(A,beg,temp-1);
        QuickSort(A,temp+1,end);
        }

}
```

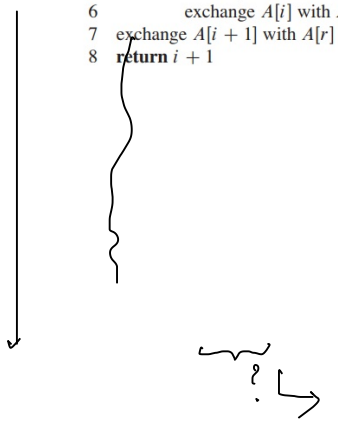
② Hoare partition scheme

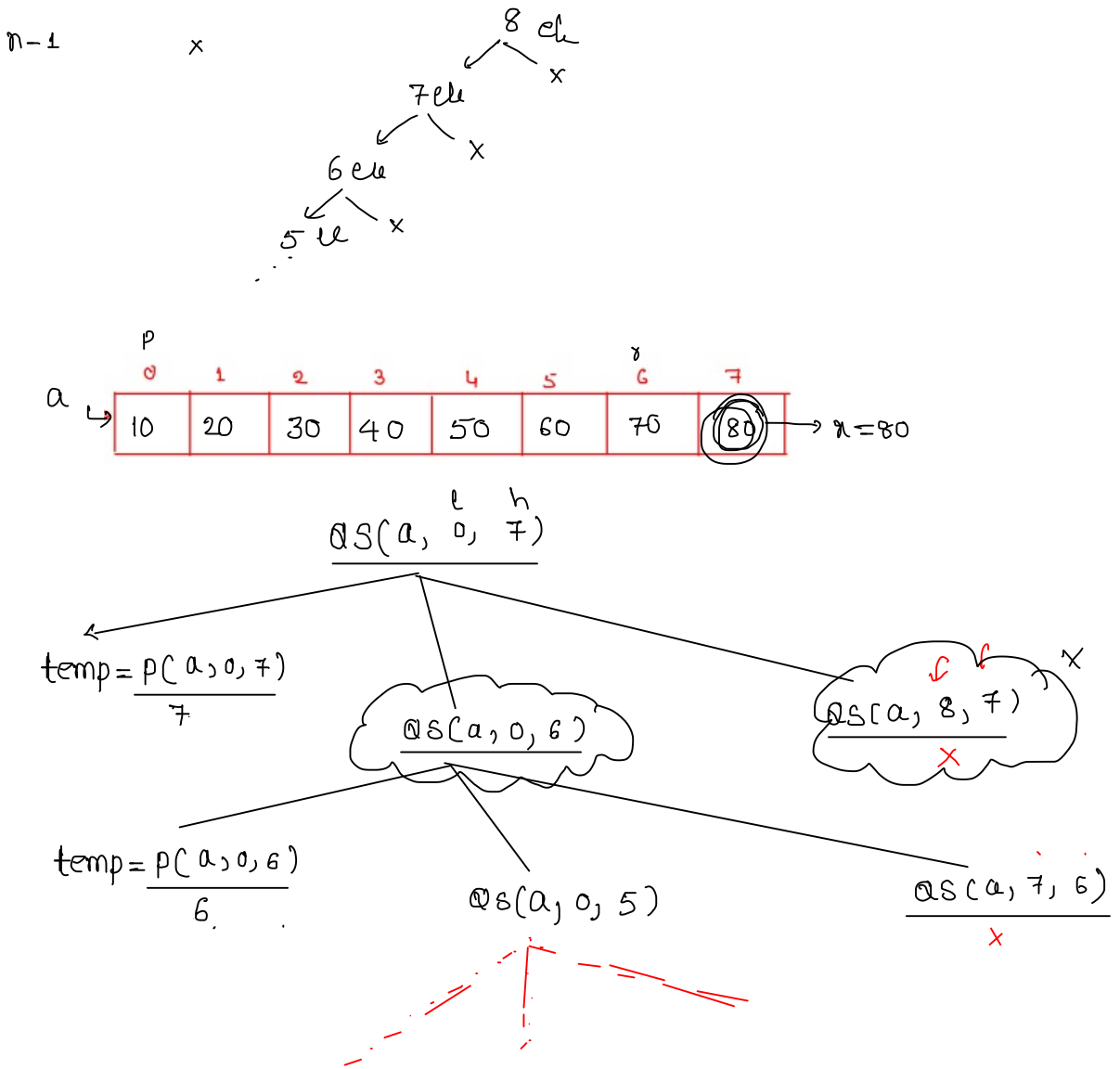
```
// Sorts a (portion of an) array, divides it into partitions, then sorts those
algorithm quicksort(A, lo, hi) is
  if lo >= 0 && hi >= 0 && lo < hi then
    p := partition(A, lo, hi)
    quicksort(A, lo, p) // Note: the pivot is now included
    quicksort(A, p + 1, hi)

// Divides array into two partitions
algorithm partition(A, lo, hi) is
  // Pivot value
  pivot := A[ floor((hi - lo)/2) + lo ] // The value in the middle of the array

  // Left index
  i := lo - 1

  // Right index
  j := hi + 1

  loop forever
    // Move the left index to the right at least once and while the element at
    // the left index is less than the pivot
    do i := i + 1 while A[i] < pivot

    // Move the right index to the left at least once and while the element at
    // the right index is greater than the pivot
    do j := j - 1 while A[j] > pivot

    // If the indices crossed, return
    if i >= j then return j

    // Swap the elements at the left and right indices
    swap A[i] with A[j]
```

Partitioning the array

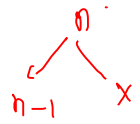
The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p..r]$ in place.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```





```
main()
{
        input array reading
        QuickSort(A,0,n-1);
}

function QuickSort(A,beg,end)
{
        if(beg<=end)
        {
        temp=PARTITION(A,beg,end)
        QuickSort(A,beg,temp-1);
        QuickSort(A,temp+1,end);
        }

}
```



T(n) :

→ O(n)

counts one

$$T(n) = \begin{cases} n + T(n-1) & ; n \geq 2 \\ 1 & ; n == 1 \end{cases}$$

n-1

x

2nd one

$$T(n) = \begin{cases} n + T(n/2) + T(n/2) & ; n \geq 2 \\ 1 & ; n == 1 \end{cases}$$

NOT ALWAYS

→ may be (some times) possible

when split is n; $\frac{n}{2}$ $\frac{n}{2}$

↳ $O(n \log_2 n)$ ✓

$$T(n) = \begin{cases} n + T(n-1) & ; n \geq 2 \\ 1 & ; n == 1 \end{cases}$$

↑ base case

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

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

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

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

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

$$T(n) = T(n-4) + n-3 + n-2 + n-1 + n$$

$\vdots$

after k steps

$$= T(n-k) + n-(k-1) + \cdots + n-1+n$$

$$T(n-k) + n-(k-1) + \cdots + n-1+n$$

(1 over $T(n-k)$)

$= n-k=1$

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

$$= 1 + 1 + 1+2+3+ \cdots + n-1+n$$

$$= 2 + 1+2+3+ \cdots + n$$

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

wc of quick sort.

① All eles are in ↑ order

10, 20, 30, 40, 50, ...

② All eles are in ↓ order

50, 40, 30, 20, 10, ...

③ All eles are same

TC: $O(n^2)$

---

remaining type :- TC: $O(n \log_2 n)$

Why Quick sort is called as Quick :-

1 ele perspective:-

wait

→ 1 partition time :- $O(n)$

Why MS is called as Merge ?

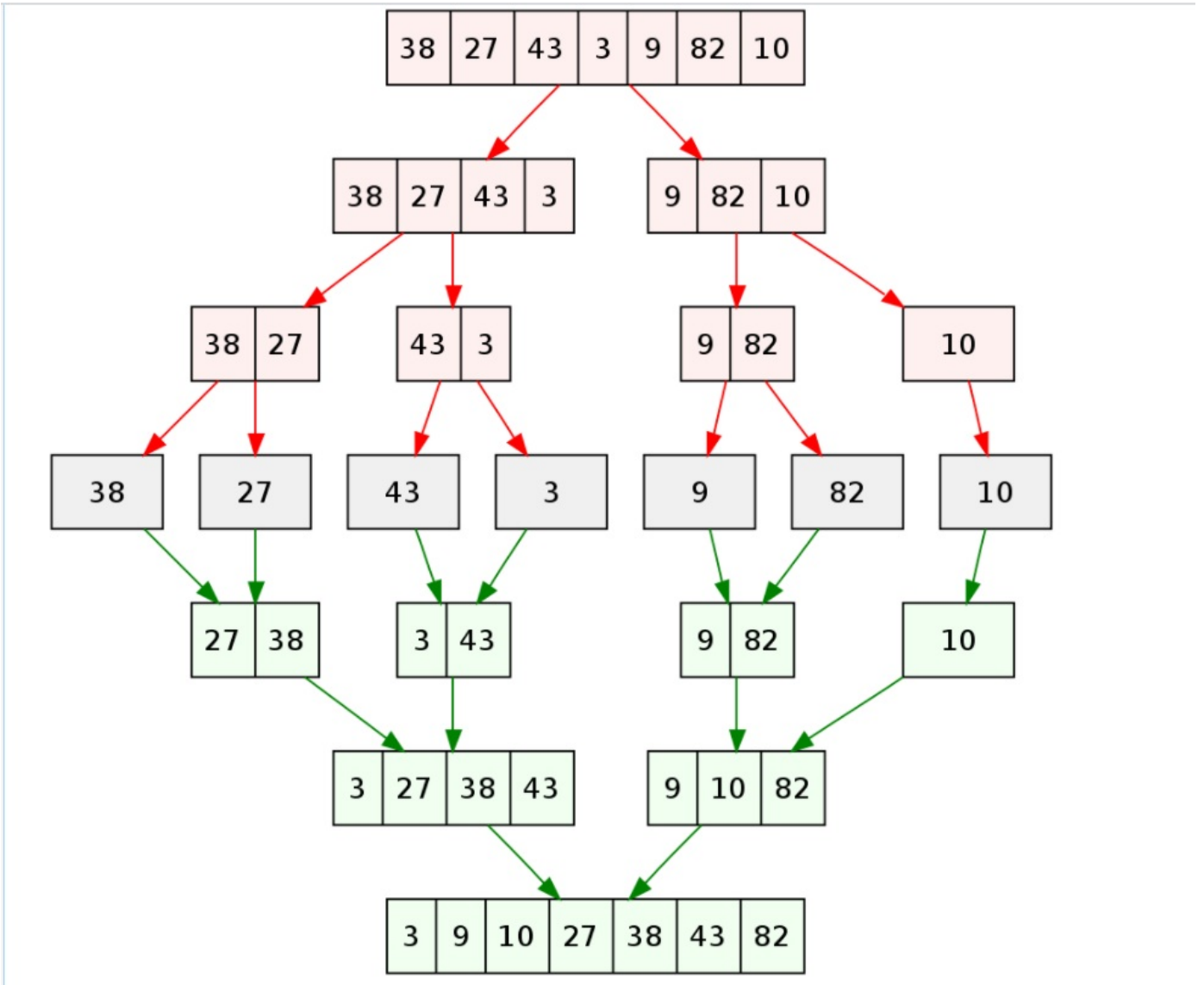
NO-LUCK

↳ since we are applying merge process.

100%

just to place 1 ele in its correct position.

wait $O(n \cdot \log_2 n)$


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82