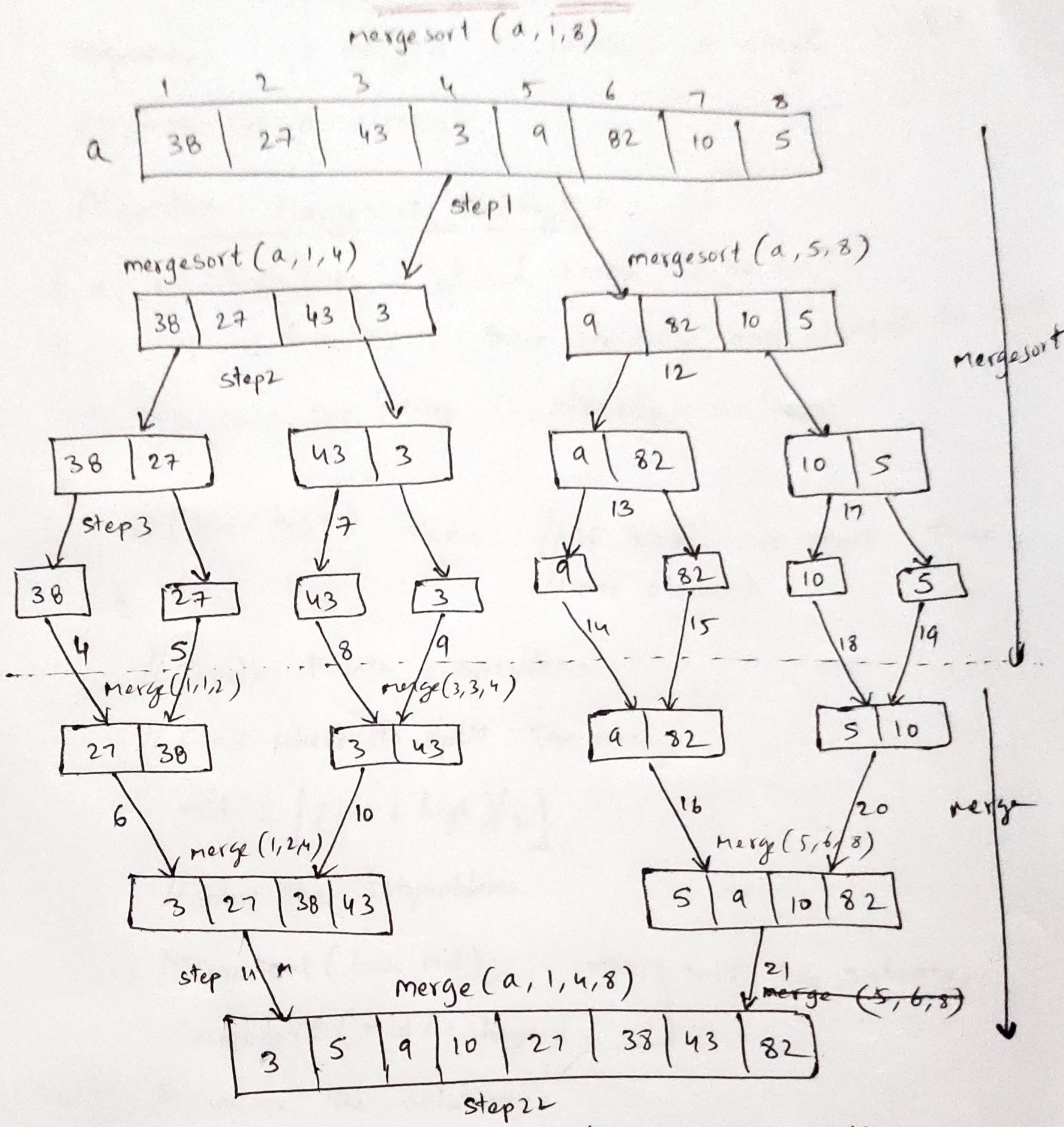
MERGE SORT



→ Given a sequence of n elements (also called keys)

a[1], a[2], ... a[n]

→ Elements are to be sorted in non decreasing (increasing) order

$\boxed{\text{Eg:- } 5, 7, 9, 10, 10, 12, 14, 14, 15}$

→ split a[ ] into 2 sets (subarrays)

$a[1], a[2], \ldots\ldots a\left[\lfloor n/2 \rfloor\right]$ and $a\left[\lfloor n/2 \rfloor + 1\right], \ldots\ldots a[n]$

Each set is individually sorted, and resulting sequences are merged to produce a single sorted sequence of n elements.

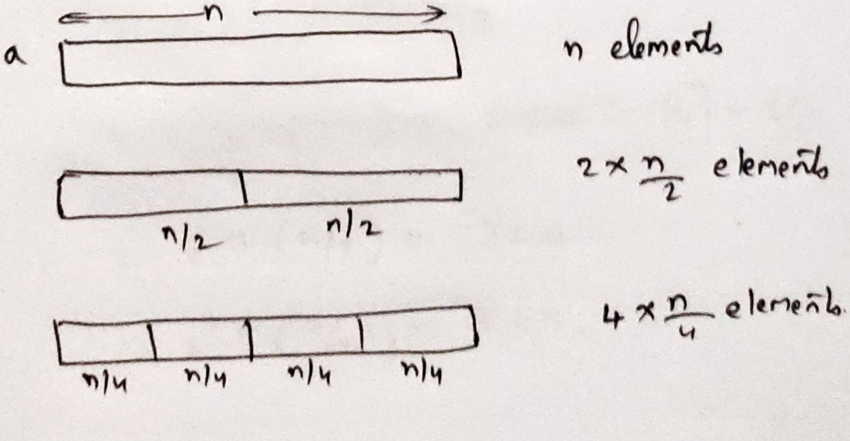
<u>Algorithm Mergesort(low, high)</u>

```
// a[low : high] is a global array to be sorted.
// small(P) is true, if there is only one element to sort.
// In this case the array is already sorted.
{
   if (low < high) then      // if there are more than
   {                         // one elements
     // Divide P into subproblems
     // Find where to split the array
      mid = ⌊(low + high)/2⌋;
     // solve the subproblems.
     Mergesort(low, mid);        ]  sort two subarrays.
     Mergesort(mid+1, high);     ]
     // Combine the solutions
     // merge 2 sorted subarrays.
      merge(low, mid, high);
   }
}
```

```
Algorithm Merge(low, mid, high)
{
  // a[low : high] is a global array containing
  // two sorted subarrays in a[low : mid] and
  // in a[mid+1, high]
  // The goal is to merge these 2 subarrays
  // into a single array residing in a[low : high]
  // we need an auxiliary (additional) array b[ ]
  // for merging.
  h := low ;   i := low ;   j := mid + 1 ;
  while ((h ≤ mid) and (j ≤ high)) do
  {
      if (a[h] ≤ a[j]) then
      {
          b[i] := a[h];
          h := h+1;
      }
      else
      {   b[i] := a[j];
          j := j+1;
      }
      i := i+1;
  }
```

```
if (h > mid) then
for k := j to high do
{
    b[i] := a[k];
    i := i+1;
}
else
{
  for k := h to mid do
  {
      b[i] := a[k];
      i := i + 1;
   }
   for k := low to high do
   {
       a[k] := b[k];
   }
}
```

Time complexity of Merge sort (Derivation)



If the time for the merging operation is proportional to n. Two sorted subarrays of size $n/2$ can be merged in time $O(n) \approx cn$, Then the computing time for merge sort is described by the recurrence relation.

$T(1) = a$

$$T(n/2) = \begin{cases} a & \text{if } n=1;\ a \text{ is a constant} \\ 2T(n/2) + cn & \text{if } n>1;\ c \text{ is a constant} \end{cases}$$

We assume that n is a power of 2.

$n = 2^k$

$k = \log_2 n$

	1	2	3
a	6	5	10

	1	2	3	4
a	6	5	10	0

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

$$= 2\left[2T(n/4) + c \cdot n/2\right] + cn$$

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

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

$$= 4\left[2T(n/8) + \cancel{2cn}\ c \cdot n/4\right] + 2cn$$

$$= 8T(n/8) + 3cn$$

$$= 2^3 T(n/2^3) + 3cn.$$

⋮

Similarly at k^{th} step, we can write

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

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

$$= n\,T(1) + \log_2 n \times cn$$

$$= n \times a + \log_2 n \times cn$$

$$T(n) = cn\log n + an$$

$$T(n) \propto n\log n$$

$$\boxed{\therefore T(n) = O(n\log n)}$$

This is the best case, average case, and worst case time complexity of merge sort.