SC2001/CE2101/CZ2101: Algorithm Design and Analysis

Mergesort

Instructor: Assoc. Prof. ZHANG Hanwang

Courtesy of Dr. Ke Yiping, Kelly’s slides

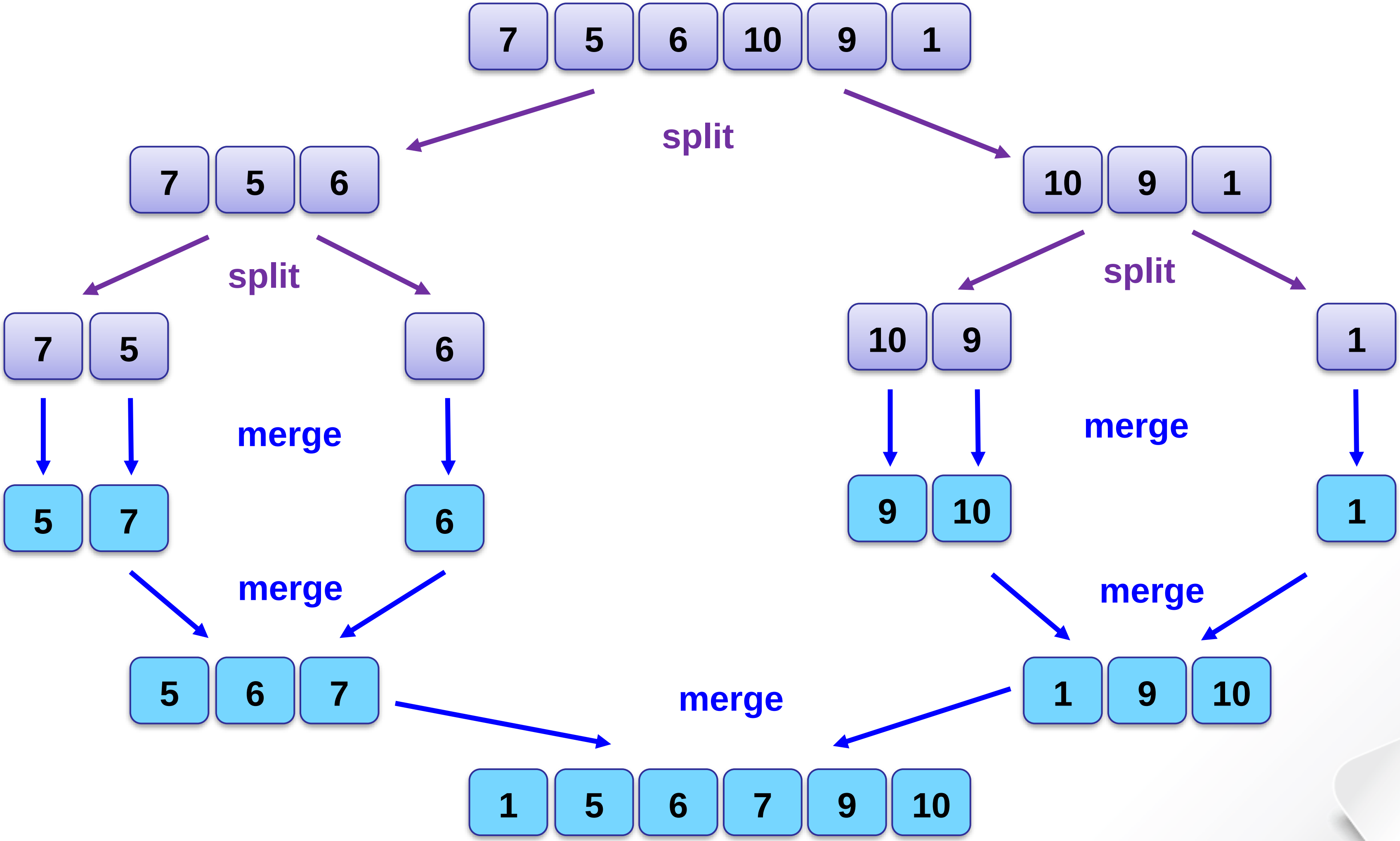
Learning Objectives

At the end of this lecture, students should be able to:

- Explain the approach of **Divide and Conquer**
- Describe how Mergesort works by:
  - Recalling the pseudo code
  - Manually executing the algorithm on a toy input array
- Analyse the **time complexity** of Mergesort, by using:
  - Recurrence equation
  - Recursion tree

Mergesort
(Divide and Conquer Approach)

Mergesort in a nutshell



Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
    {   if (n <= minimum size)
                solve the problem directly;
        else {
                divide the problem into $p_1$, $p_2$, … , $p_k$;
                for each sub-problem $p_s$
                        $solution_s$ = solve ($p_s$);
                combine all $solution_s$;
        }
    }
```

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
    {   if (n <= minimum size)
                solve the problem directly;
        else {
                divide the problem into $p_1$, $p_2$, … , $p_k$;
                for each sub-problem $p_s$
                        $solution_s$ = solve ($p_s$);
                combine all $solution_s$;
        }
    }
```

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
    {   if (n <= minimum size)
                **solve the problem directly;**
        else {
                divide the problem into $p_1$, $p_2$, … , $p_k$;
                for each sub-problem $p_s$
                        $solution_s$ = solve ($p_s$);
                combine all $solution_s$;
        }
    }
```

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
    {   if (n <= minimum size)
                solve the problem directly;
        else {
                divide the problem into $p_1$, $p_2$, … , $p_k$;
                for each sub-problem $p_s$
                        $solution_s$ = solve ($p_s$);
                combine all $solution_s$;
        }
    }
```

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
    {   if (n <= minimum size)
                solve the problem directly;
        else {
                divide the problem into $p_1$, $p_2$, … , $p_k$;
                for each sub-problem $p_s$
                        $solution_s$ = solve ($p_s$);
                combine all $solution_s$;
        }
    }
```

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
    {   if (n <= minimum size)
                solve the problem directly;
        else {
                divide the problem into $p_1$, $p_2$, … , $p_k$;
                for each sub-problem $p_s$
                        $solution_s$ = solve ($p_s$);
                combine all $solution_s$;
        }
    }
```

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
    {   if (n <= minimum size)
                solve the problem directly;
        else {
                divide the problem into $p_1$, $p_2$, … , $p_k$;
                for each sub-problem $p_s$
                        $solution_s$ = solve ($p_s$);
                combine all $solution_s$;
        }
    }
```

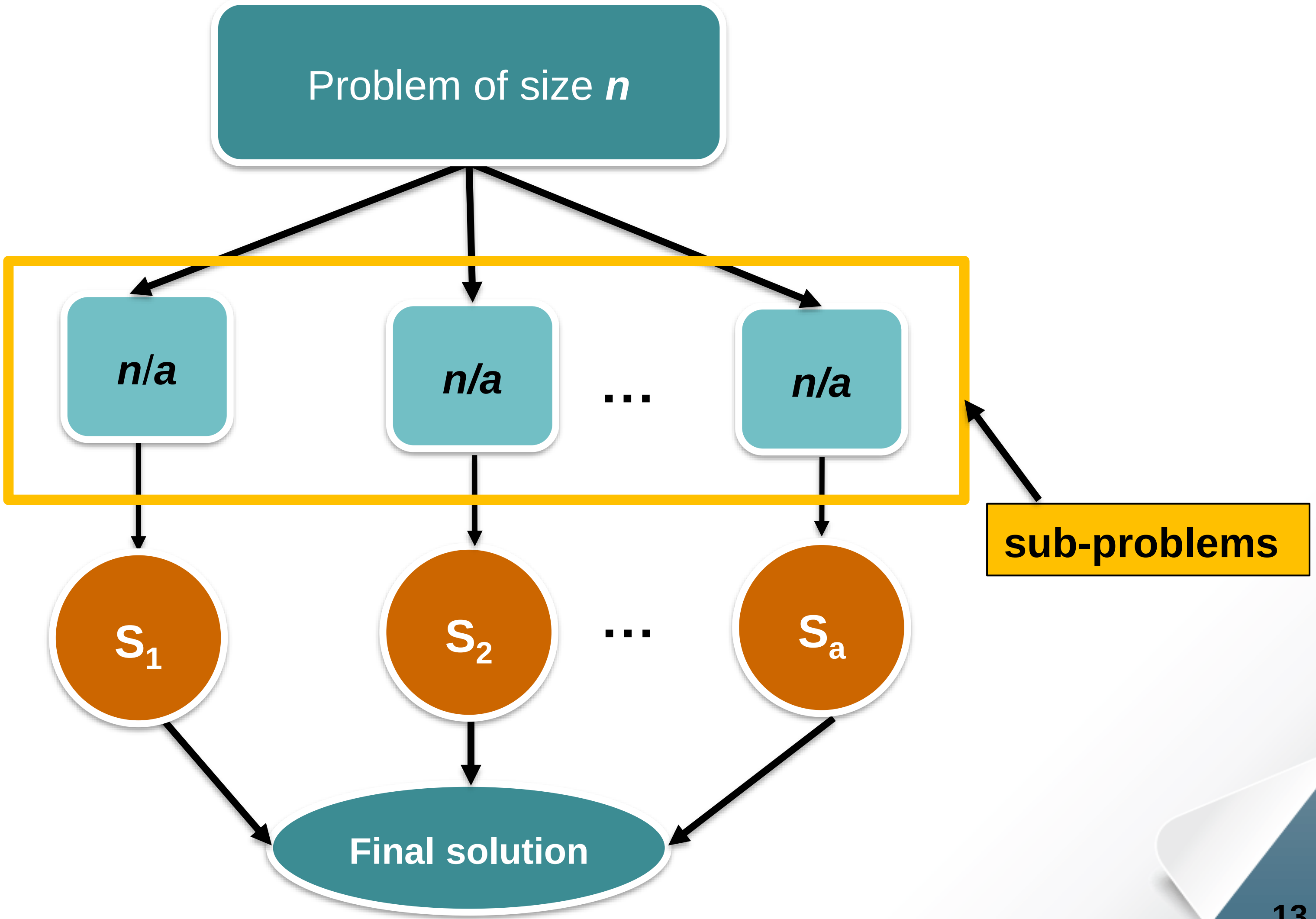
Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
    {   if (n <= minimum size)
                solve the problem directly;
        else {
                divide the problem into $p_1$, $p_2$, … , $p_k$;
                for each sub-problem $p_s$
                        $solution_s$ = solve ($p_s$);
                combine all $solution_s$;
        }
    }
```

Mergesort


Problem of size *n*
n/*a*
n/*a*
…
n/*a*
sub-problems
S_1
S_2
…
S_a
Final solution

Mergesort (Algorithm)

Mergesort (Algorithm)

```
mergeSort(list) {
    if (length of list > 1) {
        Partition list into two (approx.) equal sized
            lists, L1 & L2;
        mergeSort (L1);
        mergeSort (L2);
        merge the sorted L1 & L2;
    }
}
```

Mergesort (Algorithm)

```
mergeSort(list) {
        if (length of list > 1) {
                Partition list into two (approx.) equal sized lists, L1 & L2;
                mergeSort (L1);
                mergeSort (L2);
                merge the sorted L1 & L2;
        }
}
```

Mergesort (Algorithm)

```
mergeSort(list) {
        if (length of list > 1) {
                Partition list into two (approx.) equal sized
                        lists, L1 & L2;
                mergeSort (L1);
                mergeSort (L2);
                merge the sorted L1 & L2;
        }
}
```

Mergesort (Algorithm)

```
mergeSort(list) {
        if (length of list > 1) {
                Partition list into two (approx.) equal sized
                        lists, L1 & L2;
                mergeSort (L1);
                mergeSort (L2);
                merge the sorted L1 & L2;
        }
}
```

Mergesort (Algorithm)

```
mergeSort(list) {
    if (length of list > 1) {
        Partition list into two (approx.) equal sized
            lists, L1 & L2;
        mergeSort (L1);
        mergeSort (L2);
        merge the sorted L1 & L2;
    }
}
```

Mergesort (Algorithm)

```
mergeSort(list) {
        if (length of list > 1) {
            Partition list into two (approx.) equal sized
                lists, L1 & L2;
            mergeSort (L1);
            mergeSort (L2);
            merge the sorted L1 & L2;
        }
}
```

Mergesort (Algorithm)

```
mergeSort(list) {
        if (length of list > 1) {
            Partition list into two (approx.) equal sized
                lists, L1 & L2;
            mergeSort (L1);
            mergeSort (L2);
            merge the sorted L1 & L2;
        }
}
```

Mergesort (Algorithm)

```
mergeSort(list) {
        if (length of list > 1) {
                Partition list into two (approx.) equal sized
                        lists, L1 & L2;
                mergeSort (L1);
                mergeSort (L2);
                merge the sorted L1 & L2;
        }
}
```

Mergesort (Overview of Pseudo Code)

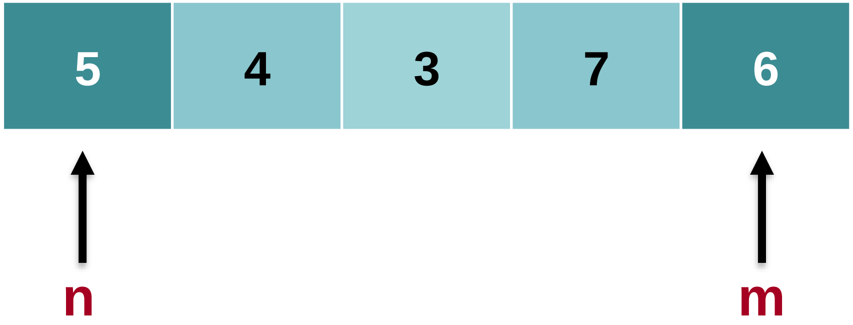
Mergesort

```
void mergesort(int n, int m)
{   int mid = (n+m)/2;
    if (m-n <= 0)
          return;
    else if (m-n > 1) {
          mergesort(n, mid);
          mergesort(mid+1, m);
    }
    merge(n, m);
}
```

5	4	3	7	6

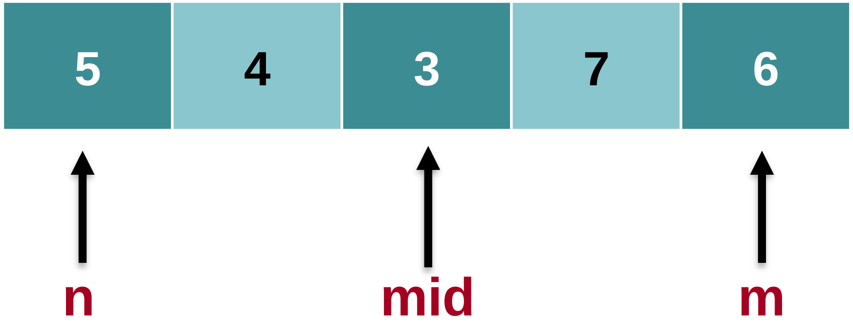
Mergesort

```
void mergesort(int n, int m)
{   int mid = (n+m)/2;
    if (m-n <= 0)
          return;
    else if (m-n > 1) {
          mergesort(n, mid);
          mergesort(mid+1, m);
    }
    merge(n, m);
}
```



Mergesort

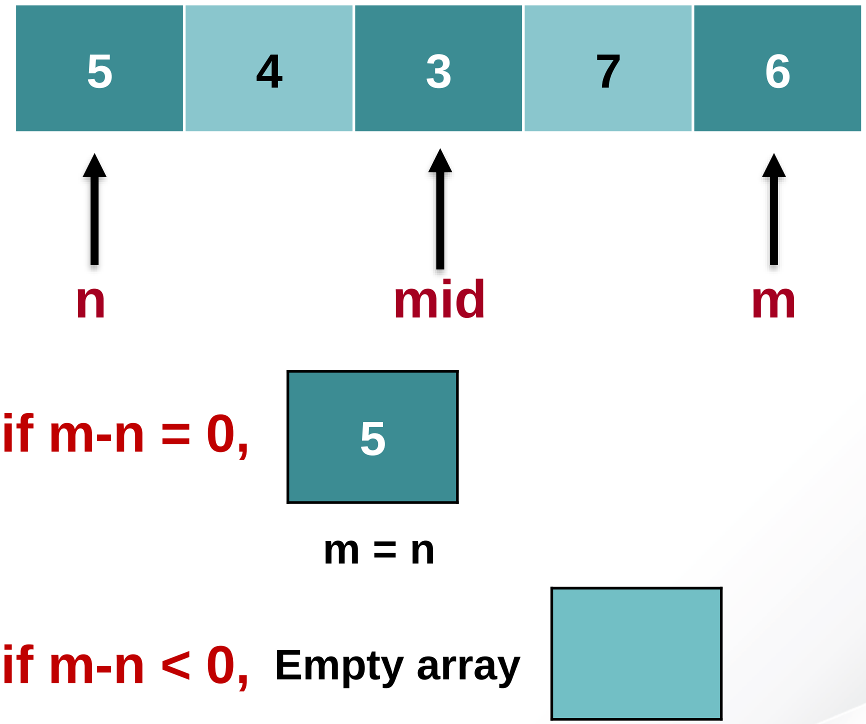
```
void mergesort(int n, int m)
{   int mid = (n+m)/2;
    if (m-n <= 0)
          return;
    else if (m-n > 1) {
          mergesort(n, mid);
          mergesort(mid+1, m);
    }
    merge(n, m);
}
```



5	4	3	7	6

Mergesort

```
void mergesort(int n, int m)
{   int mid = (n+m)/2;
    if (m-n <= 0)
          return;
    else if (m-n > 1) {
          mergesort(n, mid);
          mergesort(mid+1, m);
    }
    merge(n, m);
}
```



5	4	3	7	6

n mid m

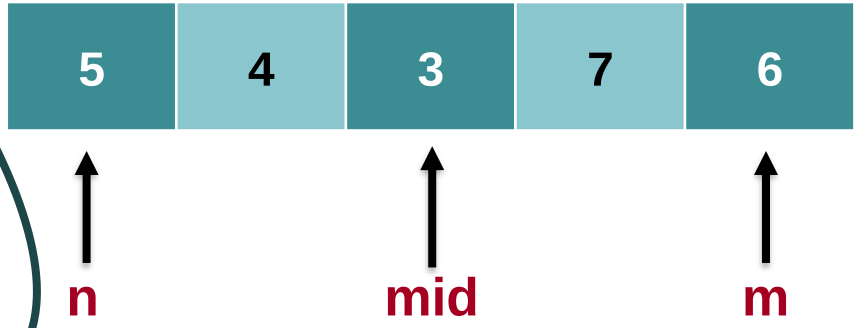
if m-n = 0, 5

m = n

if m-n < 0, Empty array

Mergesort

```
void mergesort(int n, int m)
{   int mid = (n+m)/2;
    if (m-n <= 0)
        return;
    else if (m-n > 1) {
        mergesort(n, mid);
        mergesort(mid+1, m);
    }
    merge(n, m);
}
```



Mergesort (Example)

Mergesort

Sort in ascending order

0	1	2	3	4	5	6	7
90	25	10	71	94	22	59	74

Mergesort

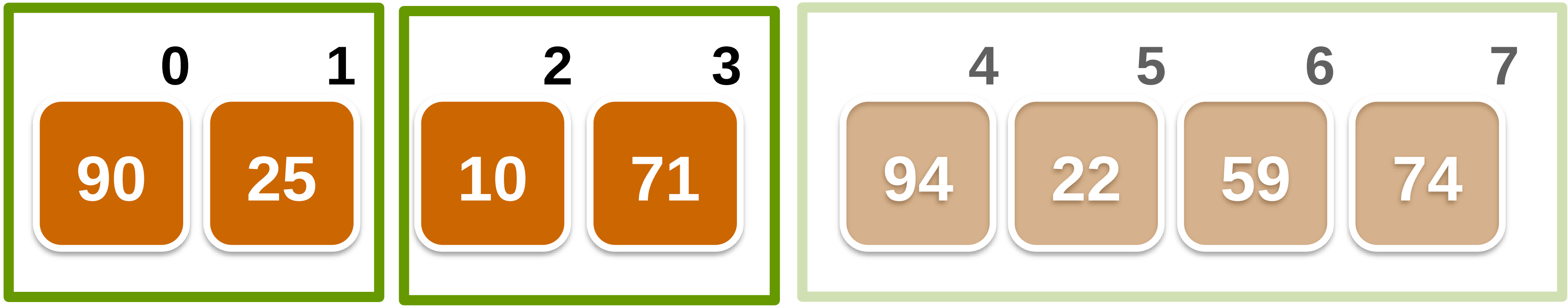
Sort in ascending order

0	1	2	3
90	25	10	71

4	5	6	7
94	22	59	74

Mergesort

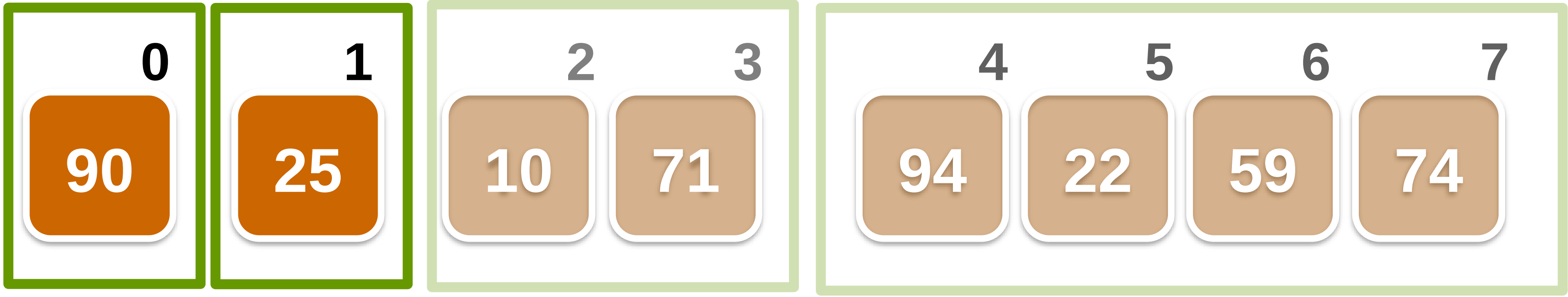
Sort in ascending order



0	1	2	3	4	5	6	7
90	25	10	71	94	22	59	74

Mergesort

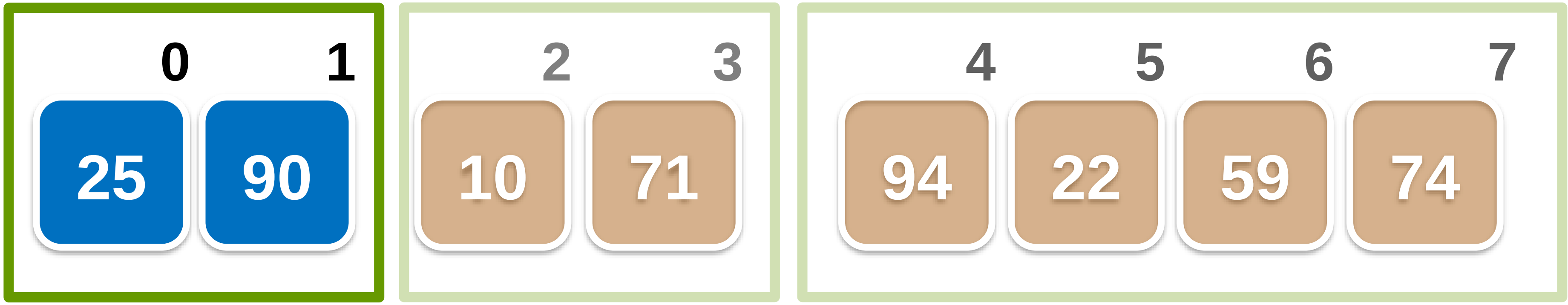
Sort in ascending order



0	1	2	3	4	5	6	7
90	25	10	71	94	22	59	74

Mergesort

Sort in ascending order

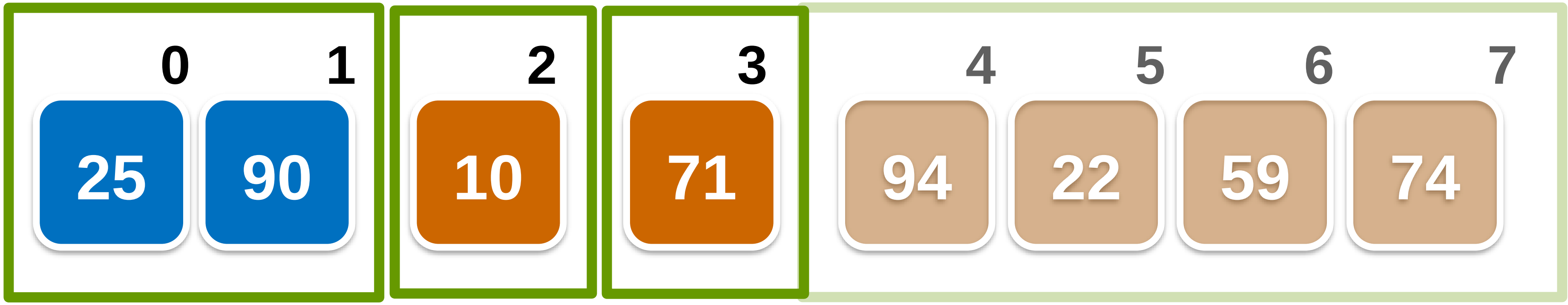


0	1	2	3	4	5	6	7
25	90	10	71	94	22	59	74

1 key comparison in merging

Mergesort

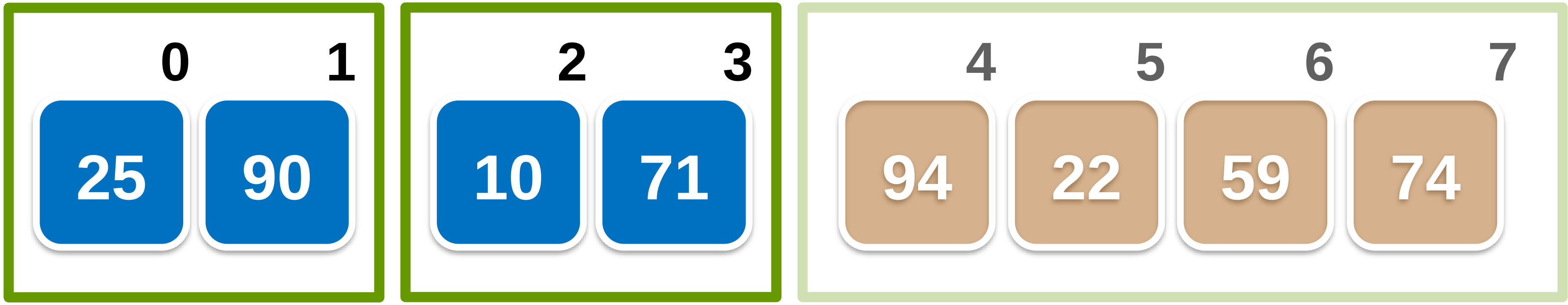
Sort in ascending order



0	1	2	3	4	5	6	7
25	90	10	71	94	22	59	74

Mergesort

Sort in ascending order

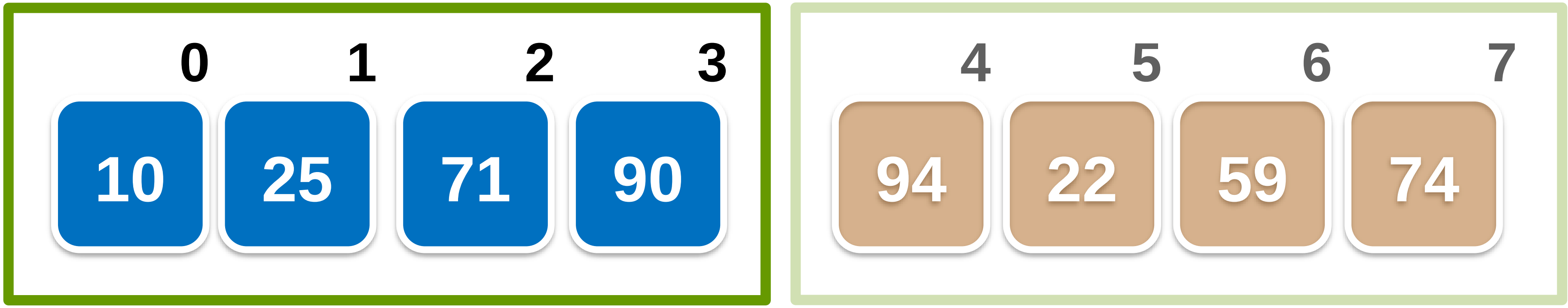


0	1	2	3	4	5	6	7
25	90	10	71	94	22	59	74

1 key comparison in merging

Mergesort

Sort in ascending order

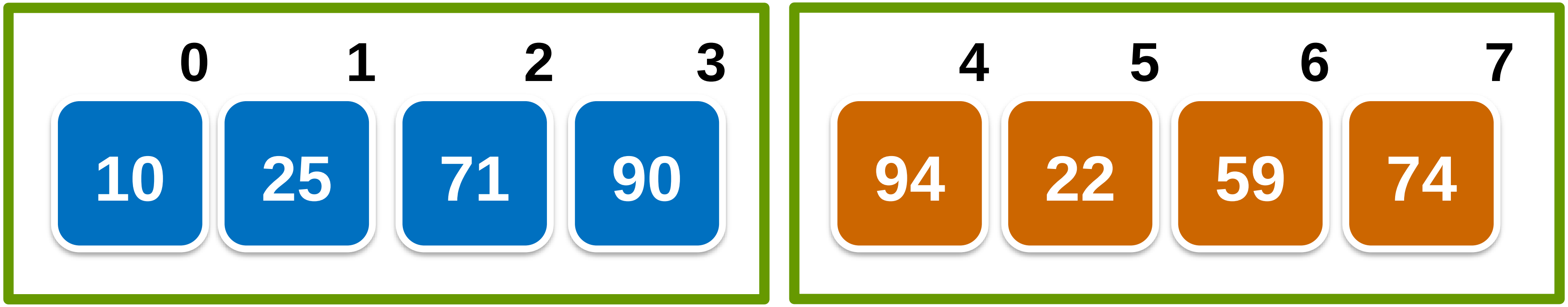


0	1	2	3	4	5	6	7
10	25	71	90	94	22	59	74

3 key comparison in merging

Mergesort

Sort in ascending order

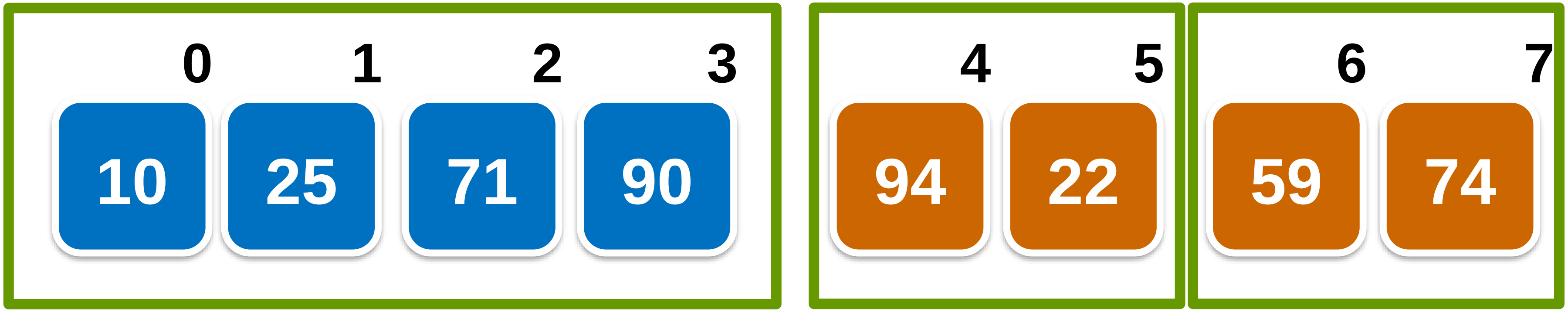


0	1	2	3	4	5	6	7
10	25	71	90	94	22	59	74

3 key comparison in merging

Mergesort

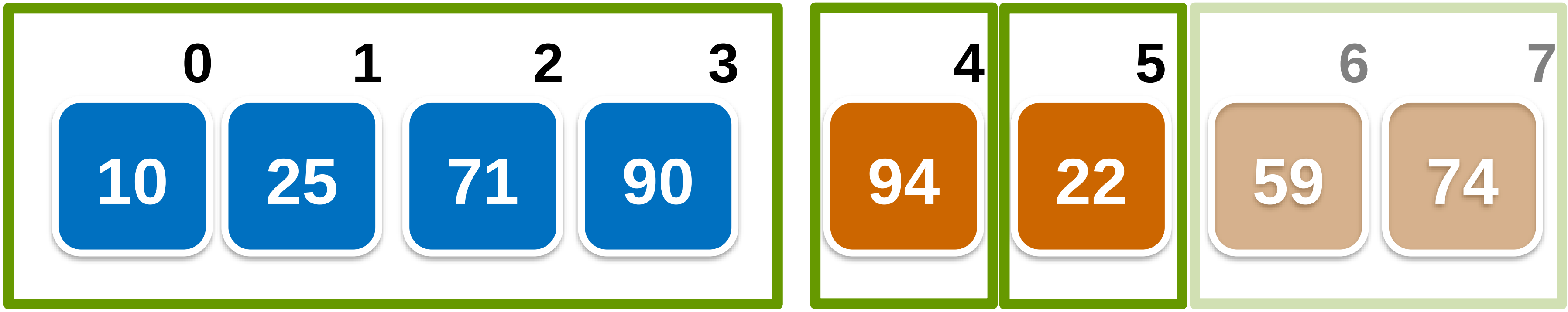
Sort in ascending order



0	1	2	3	4	5	6	7
10	25	71	90	94	22	59	74

Mergesort

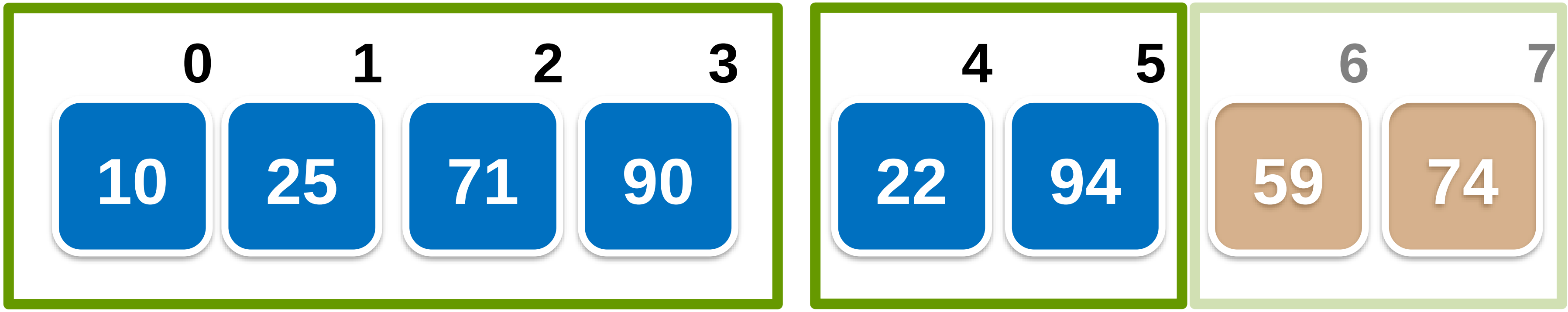
Sort in ascending order



0	1	2	3	4	5	6	7
10	25	71	90	94	22	59	74

Mergesort

Sort in ascending order

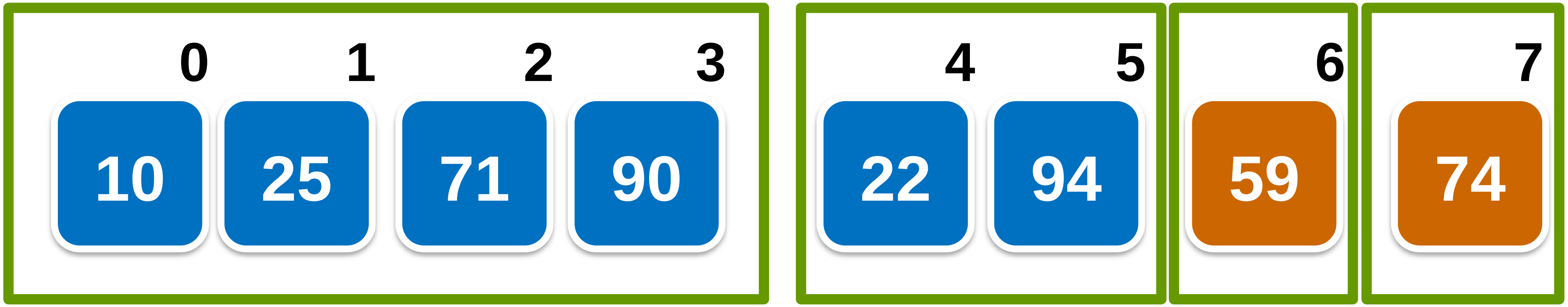


0	1	2	3	4	5	6	7
10	25	71	90	22	94	59	74

1 key comparison in merging

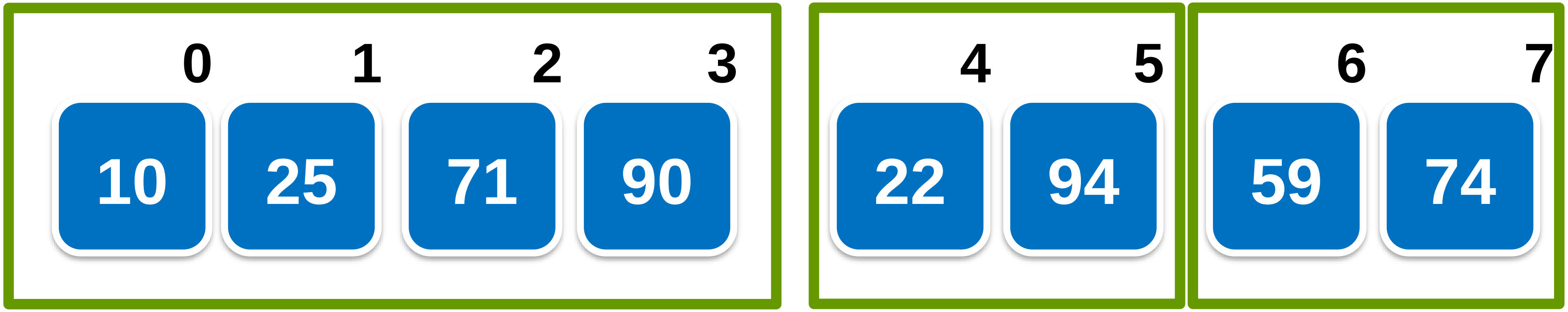
Mergesort

Sort in ascending order



Mergesort

Sort in ascending order

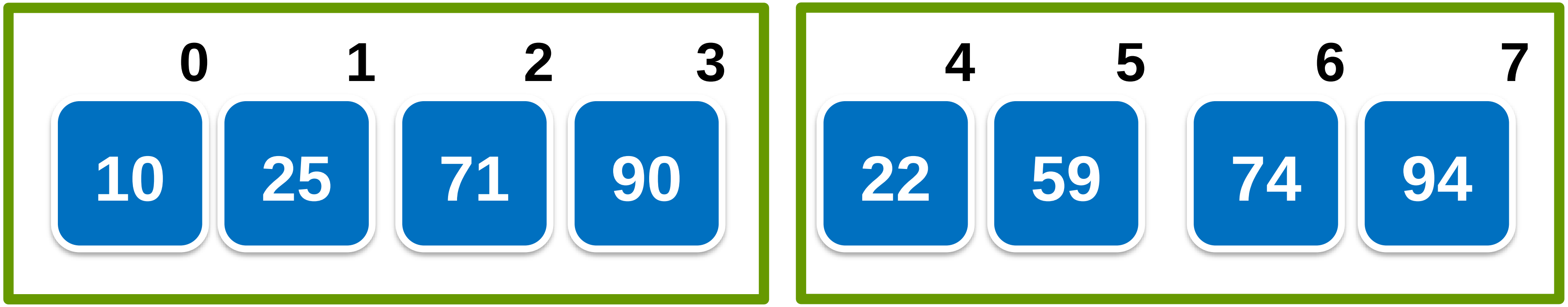


0	1	2	3	4	5	6	7
10	25	71	90	22	94	59	74

1 key comparison in merging

Mergesort

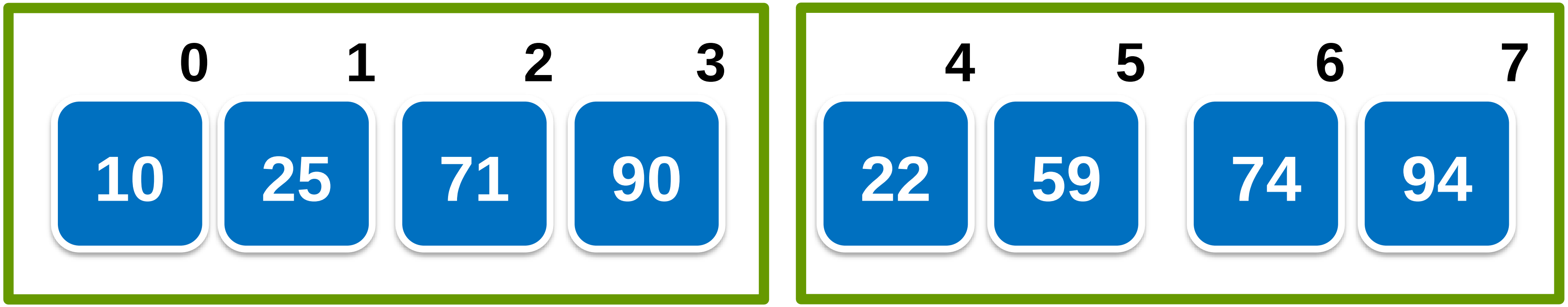
Sort in ascending order



0	1	2	3	4	5	6	7
10	25	71	90	22	59	74	94

3 key comparison in merging

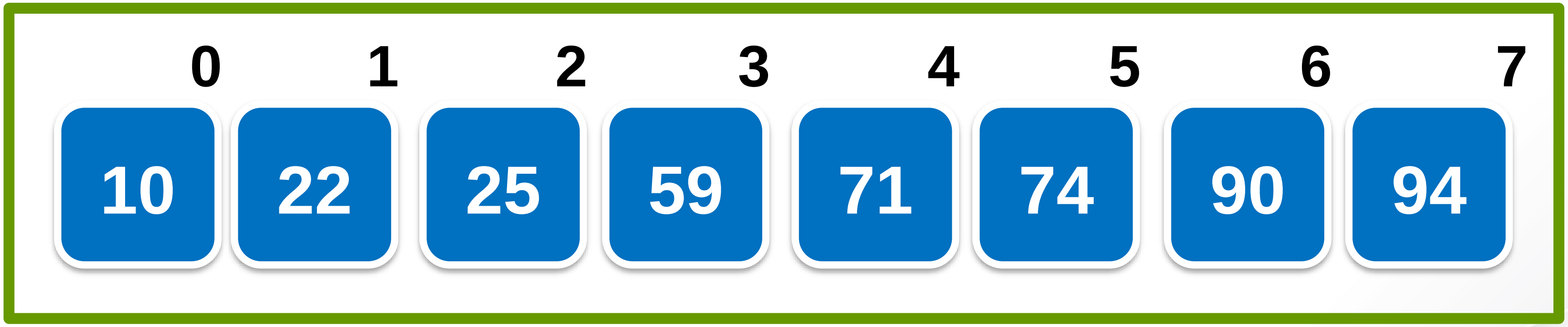
Mergesort



0	1	2	3
10	25	71	90

4	5	6	7
22	59	74	94

Sorted in ascending order



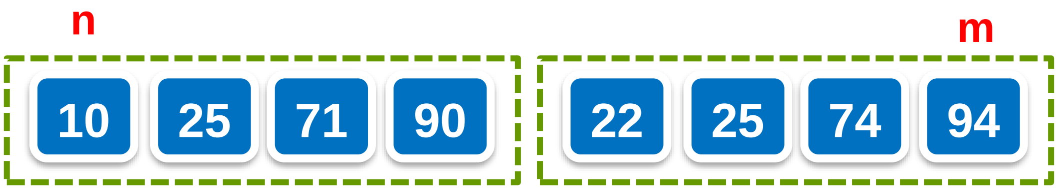
0	1	2	3	4	5	6	7
10	22	25	59	71	74	90	94

7 key comparisons in merging

Merge (Pseudo Code)

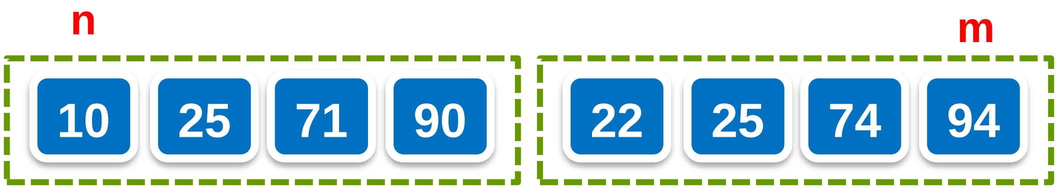
Merge (Pseudo Code)

```
void merge(int n, int m) {
    if (m-n <= 0) return;
    divide the list into 2 halves;  // both halves are sorted
    while (both halves are not empty) {
        compare the 1st elements of the 2 halves;  // 1 comparison
        if (1st element of 1st half is smaller)
            1st element of 1st half joins the end of the merged list;
        else if (1st element of 2nd half is smaller)
            move the 1st element of 2nd half to the end of the merged list;
```



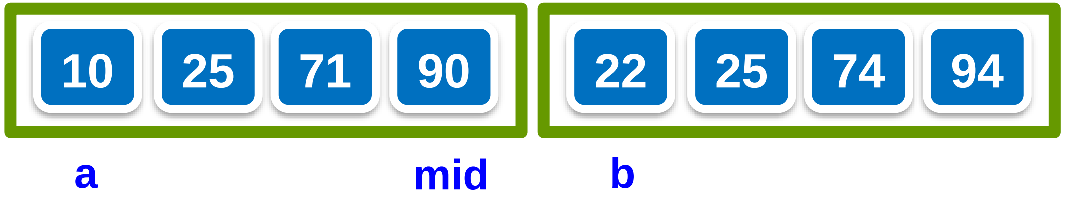
Merge (Pseudo Code)

```
void merge(int n, int m) {
        if (m-n <= 0) return;
        divide the list into 2 halves;  // both halves are sorted
        while (both halves are not empty) {
                compare the 1st elements of the 2 halves;  // 1 comparison
                if (1st element of 1st half is smaller)
                        1st element of 1st half joins the end of the merged list;
                else if (1st element of 2nd half is smaller)
                        move the 1st element of 2nd half to the end of the
                        merged list;
```



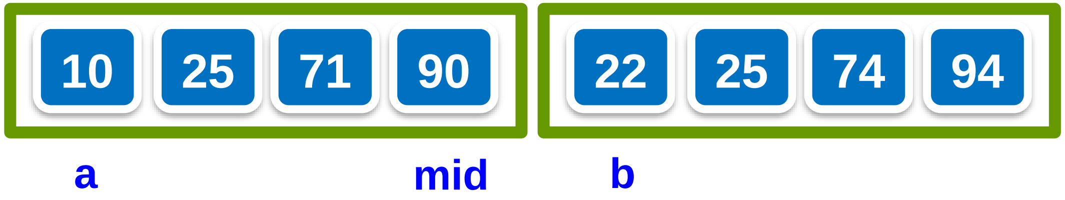
Merge (Pseudo Code)

```
void merge(int n, int m) {
        if (m-n <= 0) return;
        divide the list into 2 halves;  // both halves are sorted
        while (both halves are not empty) {
                compare the 1st elements of the 2 halves;  // 1 comparison
                if (1st element of 1st half is smaller)
                        1st element of 1st half joins the end of the merged list;
                else if (1st element of 2nd half is smaller)
                        move the 1st element of 2nd half to the end of the
                        merged list;
```



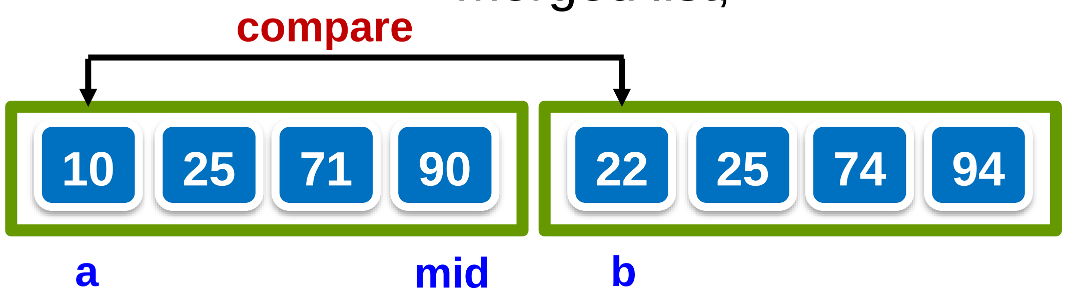
Merge (Pseudo Code)

```
void merge(int n, int m) {
        if (m-n <= 0) return;
        divide the list into 2 halves;  // both halves are sorted
        while (both halves are not empty) {
                compare the 1st elements of the 2 halves;  // 1 comparison
                if (1st element of 1st half is smaller)
                        1st element of 1st half joins the end of the merged list;
                else if (1st element of 2nd half is smaller)
                        move the 1st element of 2nd half to the end of the
                        merged list;
```



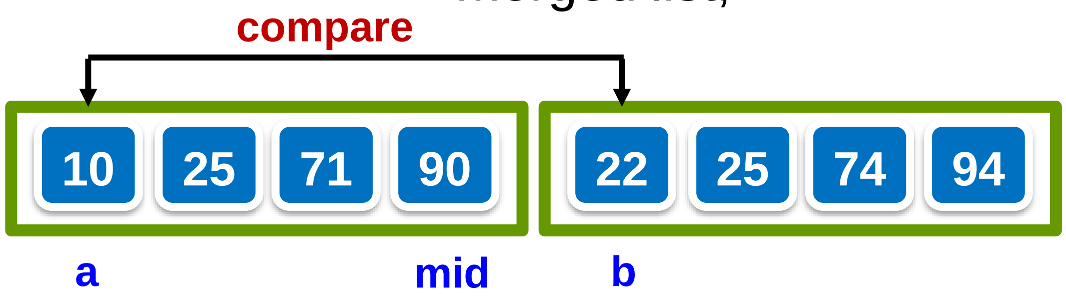
Merge (Pseudo Code)

```
void merge(int n, int m) {
    if (m-n <= 0) return;
    divide the list into 2 halves;  // both halves are sorted
    while (both halves are not empty) {
        compare the 1st elements of the 2 halves;  // 1 comparison
        if (1st element of 1st half is smaller)
            1st element of 1st half joins the end of the merged list;
        else if (1st element of 2nd half is smaller)
            move the 1st element of 2nd half to the end of the merged list;
```



Merge (Pseudo Code)

```
void merge(int n, int m) {
    if (m-n <= 0) return;
    divide the list into 2 halves;  // both halves are sorted
    while (both halves are not empty) {
        compare the 1st elements of the 2 halves;  // 1 comparison
        if (1st element of 1st half is smaller)
            1st element of 1st half joins the end of the merged list;
        else if (1st element of 2nd half is smaller)
            move the 1st element of 2nd half to the end of the
            merged list;
```



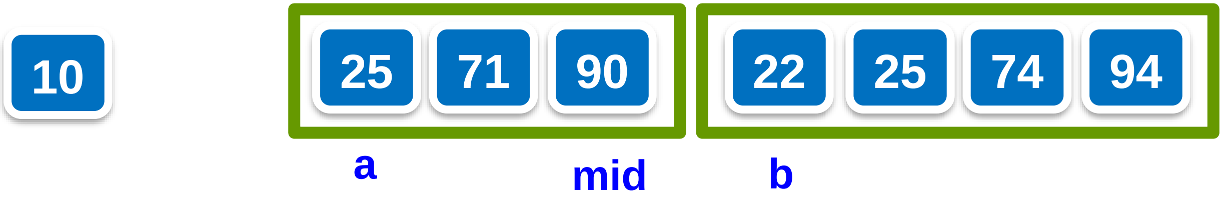
compare

10 25 71 90 | 22 25 74 94

a mid b

Merge (Pseudo Code)

```
void merge(int n, int m) {
        if (m-n <= 0) return;
        divide the list into 2 halves;  // both halves are sorted
        while (both halves are not empty) {
                compare the 1st elements of the 2 halves;  // 1 comparison
                if (1st element of 1st half is smaller)
                        1st element of 1st half joins the end of the merged list;
                else if (1st element of 2nd half is smaller)
                        move the 1st element of 2nd half to the end of the
                        merged list;
```



10

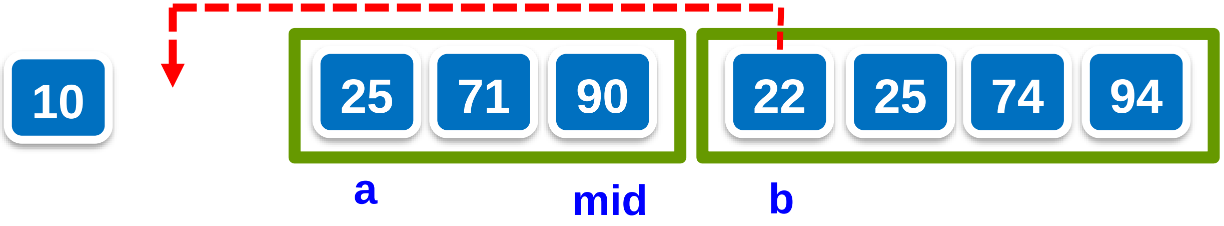
25 71 90

22 25 74 94

a mid b

Merge (Pseudo Code)

```
void merge(int n, int m) {
        if (m-n <= 0) return;
        divide the list into 2 halves;  // both halves are sorted
        while (both halves are not empty) {
                compare the 1st elements of the 2 halves;  // 1 comparison
                if (1st element of 1st half is smaller)
                        1st element of 1st half joins the end of the merged list;
                else if (1st element of 2nd half is smaller)
                        move the 1st element of 2nd half to the end of the
                        merged list;
```

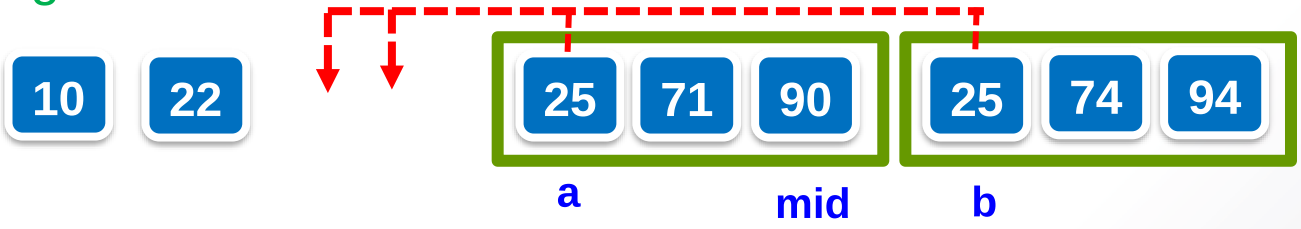


Merge (Pseudo Code)

```
else {   // the 1st elements of the 2 halves are equal
            if (they are the last elements)   break;
            1st element of 1st half joins end of the merged list;
            move the 1st element of 2nd half to the end of the merged list;
     }
   } // end of while loop;
} // end of merge
```

Merge (Pseudo Code)

```
else {   // the 1st elements of the 2 halves are equal
                    if (they are the last elements)   break;
                    1st element of 1st half joins end of the merged list;
                     move the 1st element of 2nd half to the end of the
    merged list;
        }
    } // end of while loop;
} // end of merge
```



Merge (Pseudo Code)

```
else {   // the 1st elements of the 2 halves are equal
          if (they are the last elements)   break;
          1st element of 1st half joins end of the merged list;
           move the 1st element of 2nd half to the end of the
    merged list;
      }
   } // end of while loop;
} // end of merge
```

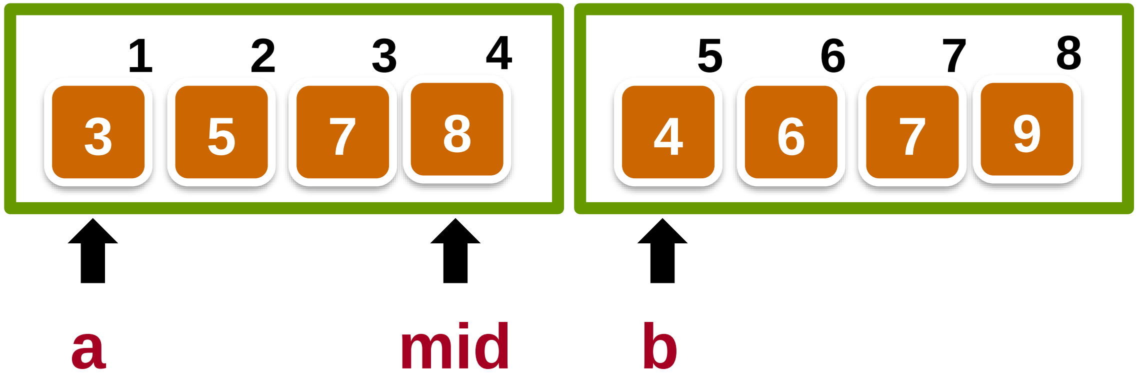
Challenge:

How to do it without auxiliary storage for the merged list?

Merge (Case Scenarios)

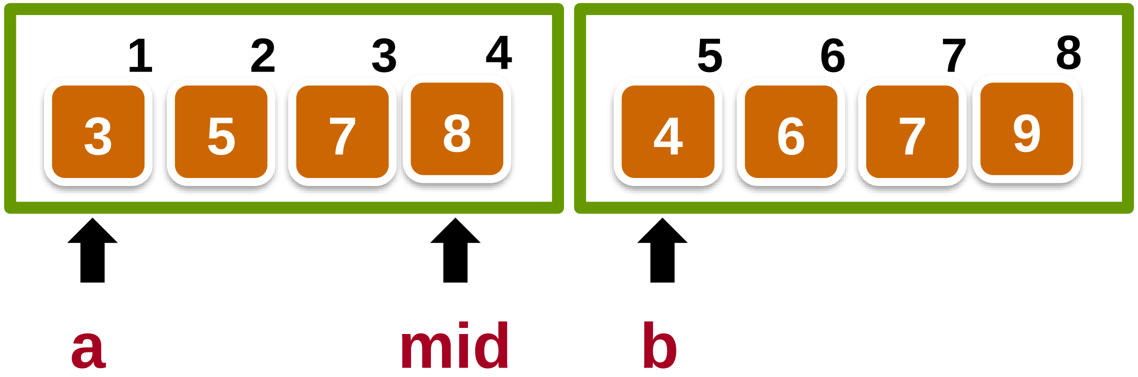
Merge (Case Scenarios)

Case 1: 1st element of 1st half is smaller

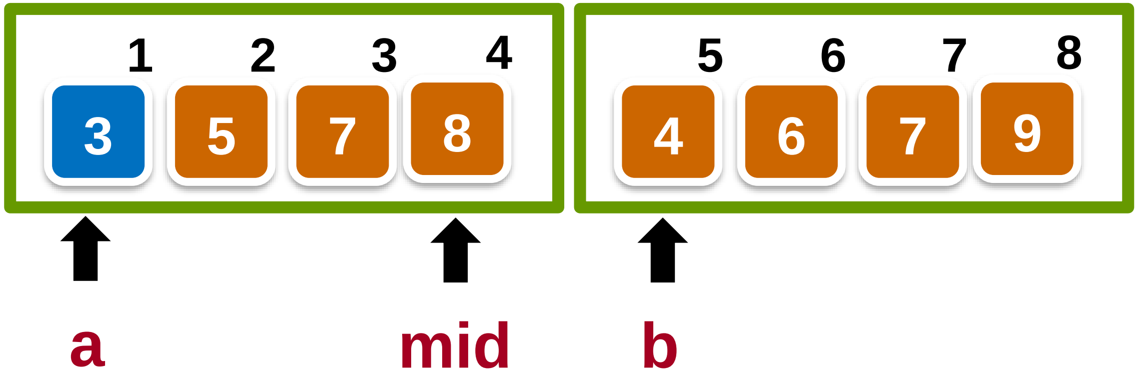


Merge (Case Scenarios)

Case 1: 1st element of 1st half is smaller



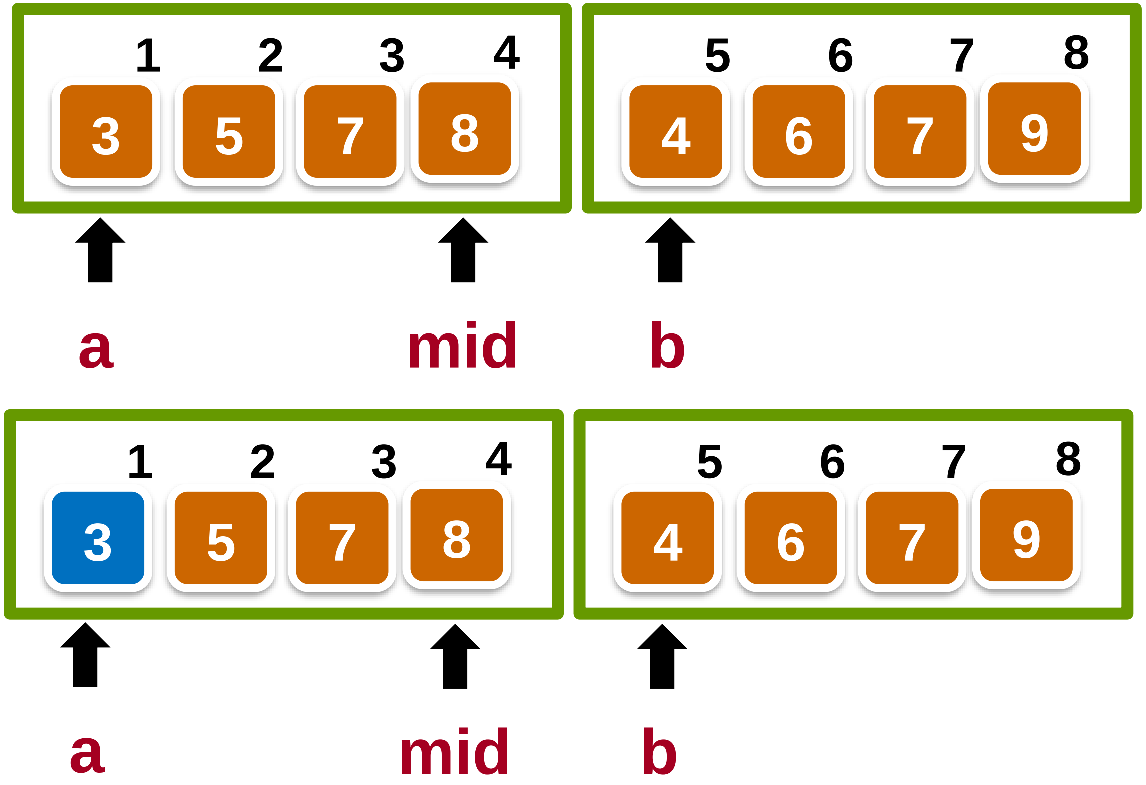
1	2	3	4	5	6	7	8
3	5	7	8	4	6	7	9
a			mid	b			



1	2	3	4	5	6	7	8
3	5	7	8	4	6	7	9
a			mid	b			

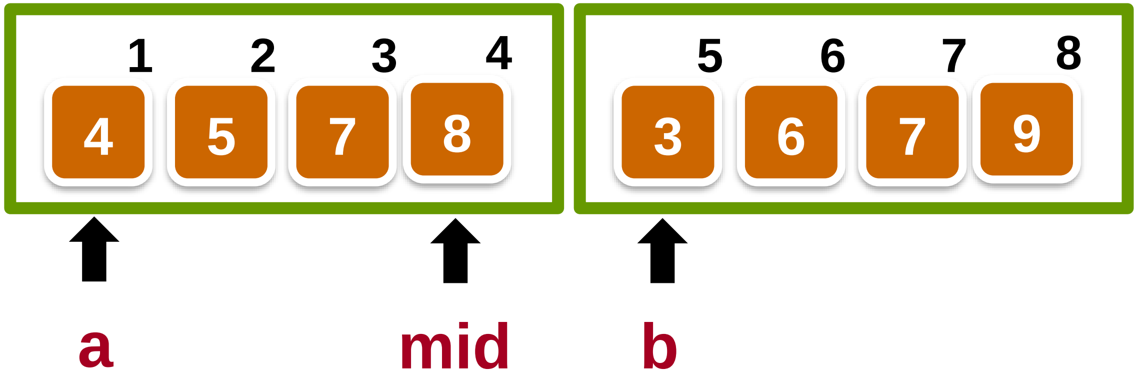
Merge (Case Scenarios)

Case 1: 1st element of 1st half is smaller



Merge (Case Scenarios)

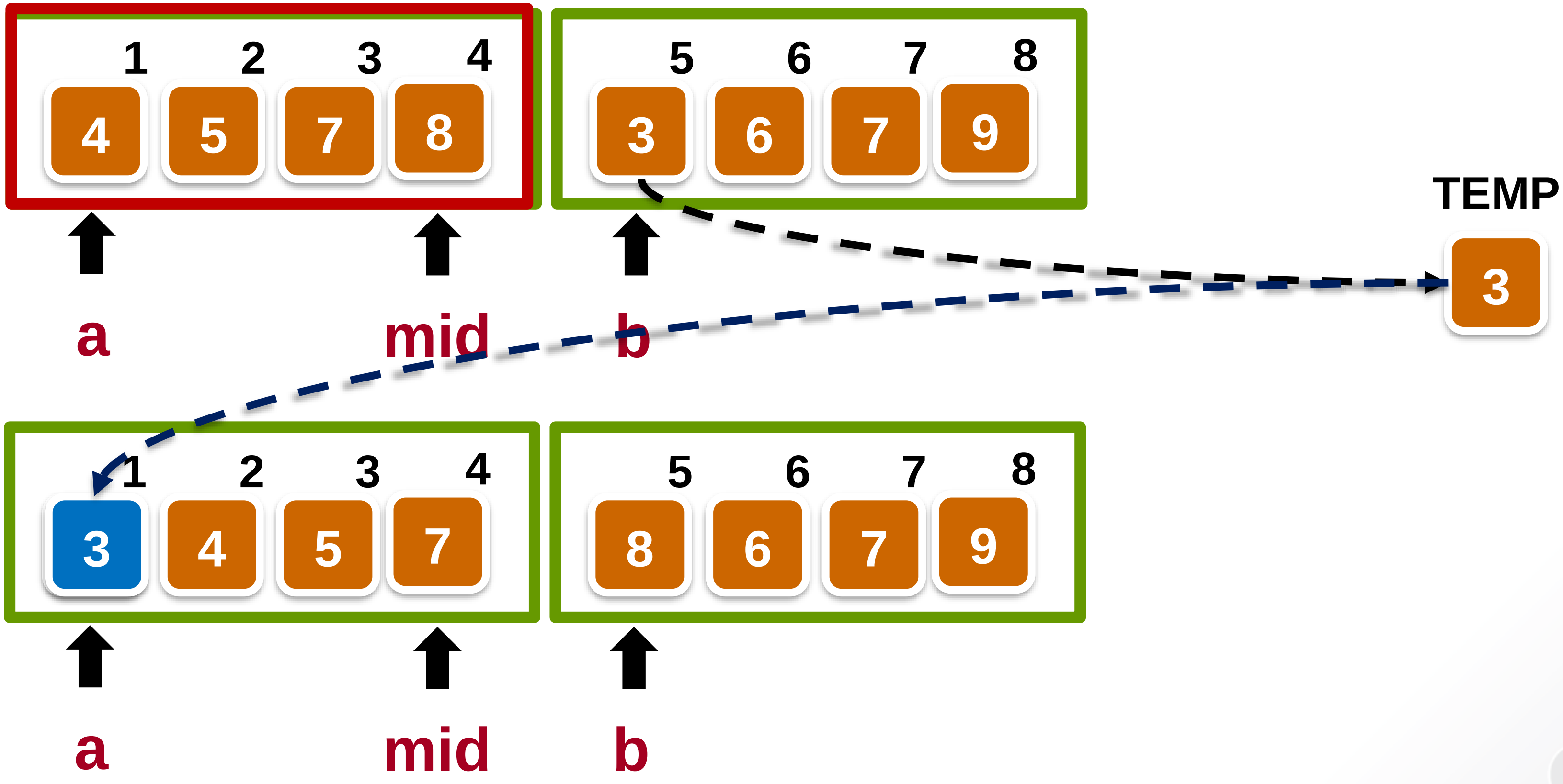
Case 2: 1st element of 2nd half is smaller



1	2	3	4	5	6	7	8
4	5	7	8	3	6	7	9

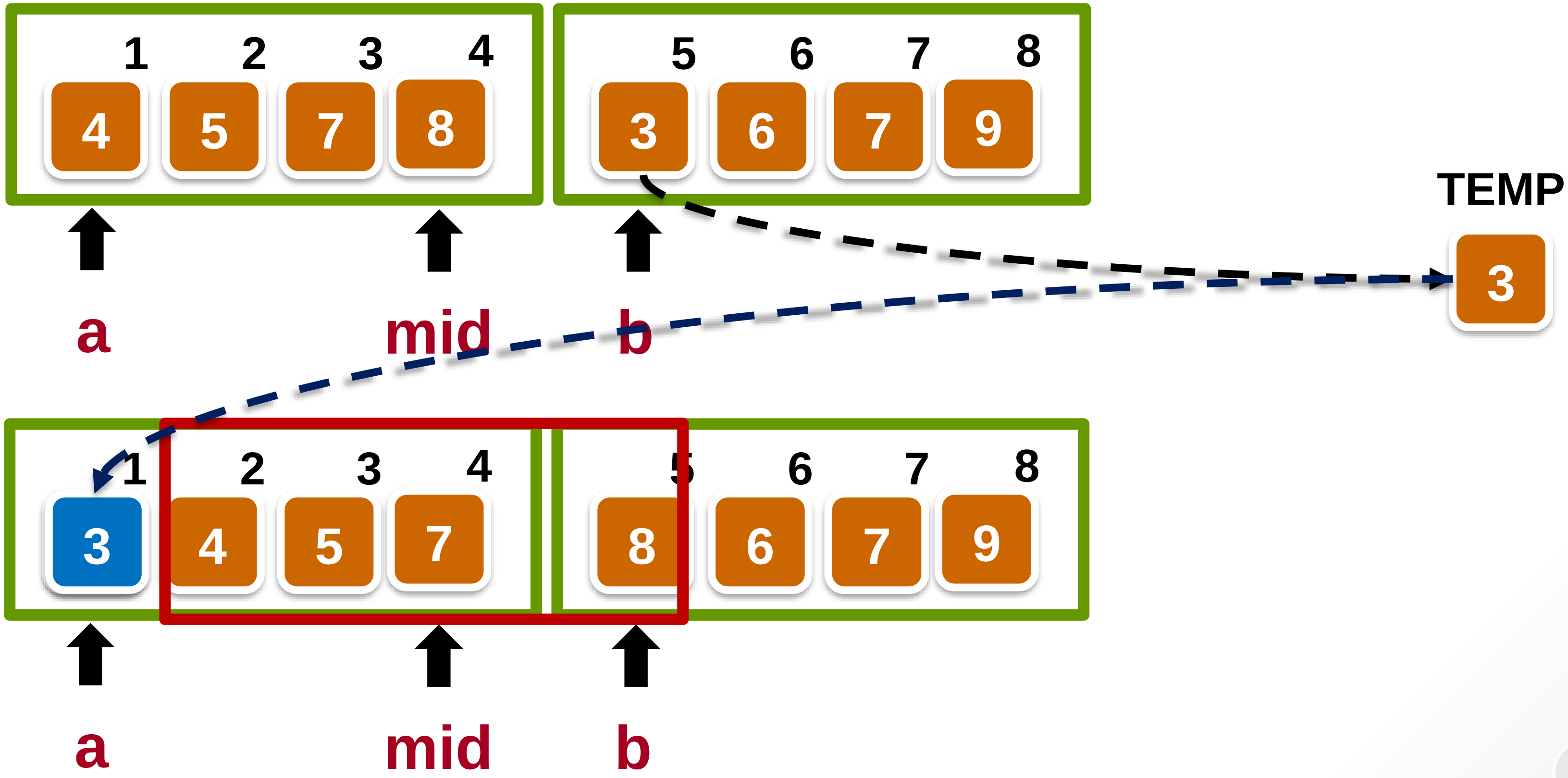
Merge (Case Scenarios)

Case 2: 1st element of 2nd half is smaller



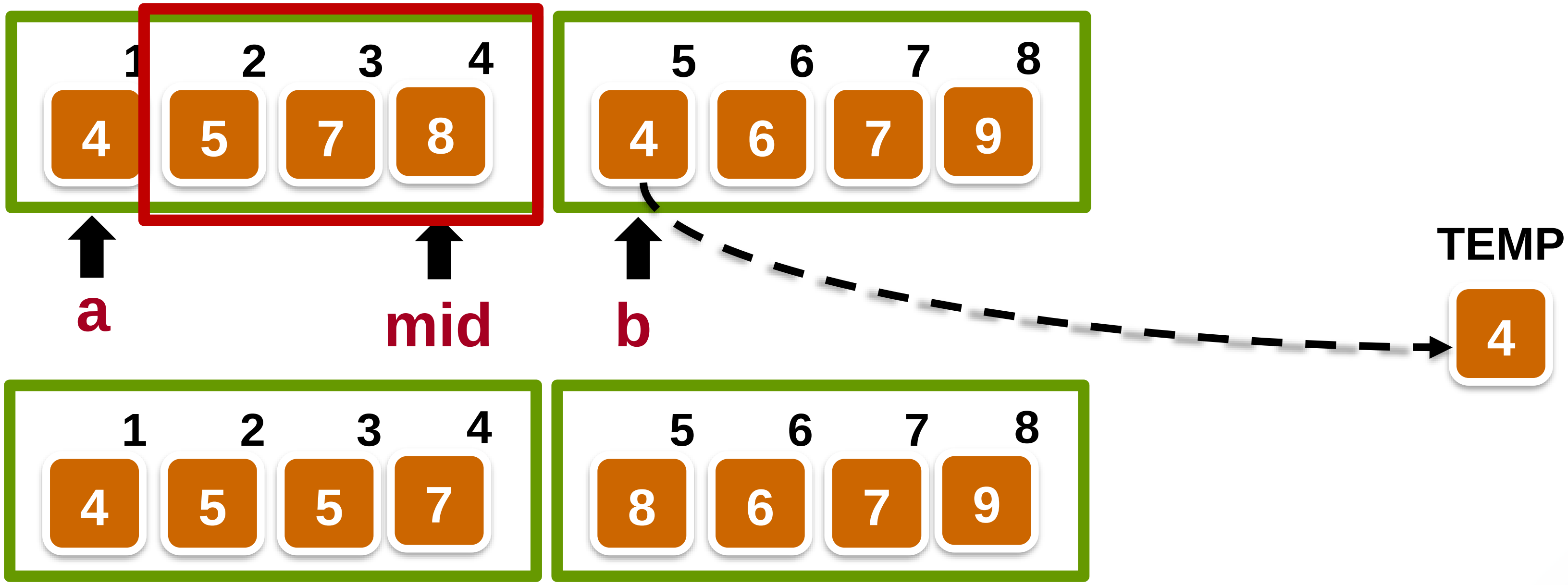
Merge (Case Scenarios)

Case 2: 1st element of 2nd half is smaller



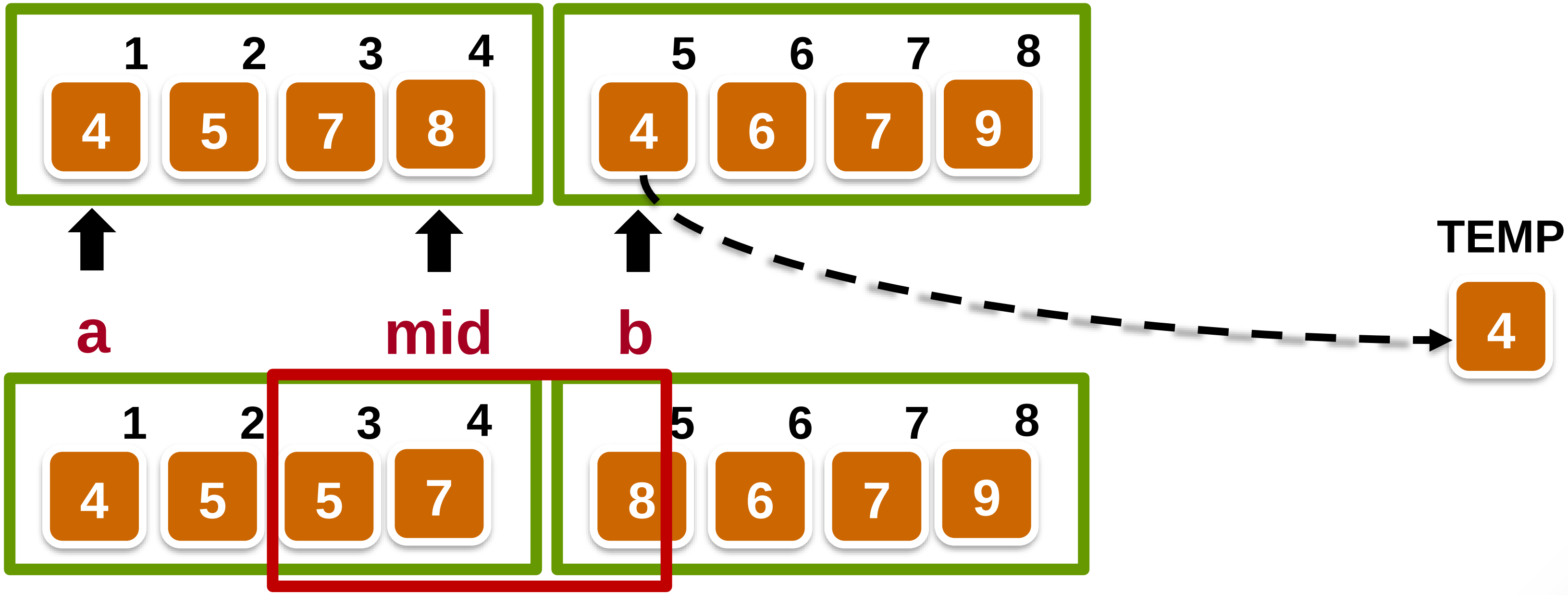
Merge (Case Scenarios)

Case 3: 1st element of 2nd half is equal



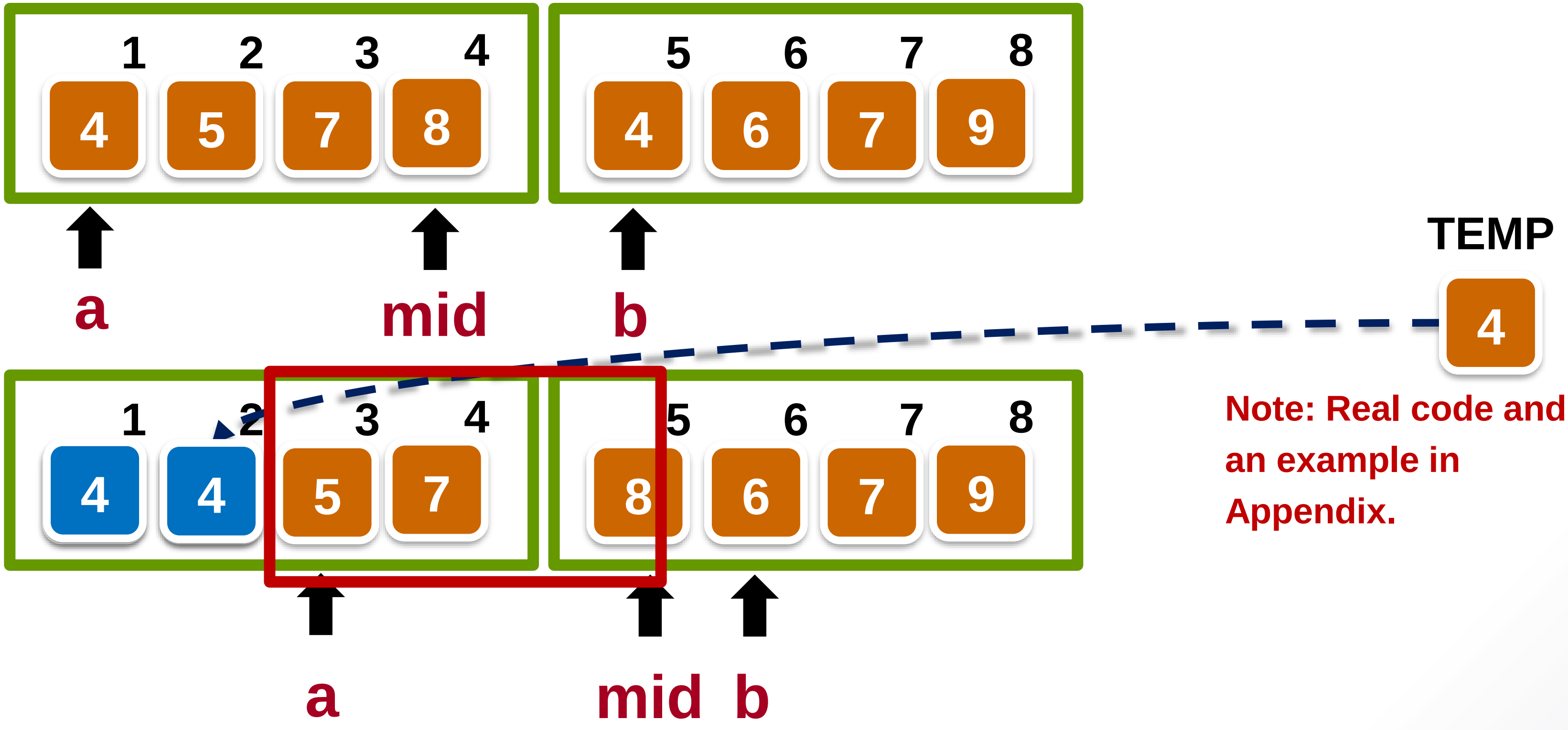
Merge (Case Scenarios)

Case 3: **1st element of 2nd half is equal**



Merge (Case Scenarios)

Case 3: 1st element of 2nd half is equal



Note: Real code and an example in Appendix.

Can we just copy the first 4 twice?

Mergesort Algorithm (Recap)

Mergesort Algorithm (Recap)

- Since merging is performed directly on the original array, swapping and shifting are needed
- mergesort() partitions a contiguous array of elements between index n and m into two subarrays

```
void mergesort(int n, int m)
{    int mid = (n+m)/2;
     if (m-n <= 0)
         return;
     else if (m-n > 1) {
         mergesort(n, mid);
         mergesort(mid+1, m);
     }
     merge(n, m);
}
```

Mergesort Algorithm (Recap)

- Since merging is performed directly on the original array, swapping and shifting are needed
- mergesort() partitions a contiguous array of elements between index n and m into two subarrays
- Recursively partitions until m-n<=0, then merge the resulting two subarrays

```
void mergesort(int n, int m)
{   int mid = (n+m)/2;
    if (m-n <= 0)
        return;
    else if (m-n > 1)
     .....
}
```

Mergesort Algorithm (Recap)

- Since merging is performed directly on the original array, swapping and shifting are needed
- mergesort() partitions a contiguous array of elements between index n and m into two subarrays
- Recursively partitions until m-n<=0, then merge the resulting two subarrays
- merge() function merges two sub-arrays of elements between index n and 'mid', and between 'mid+1' and m

```
void mergesort(int n, int m)
{    int mid = (n+m)/2;
     if (m-n <= 0)
     ……..
     merge(n, m);
}
```

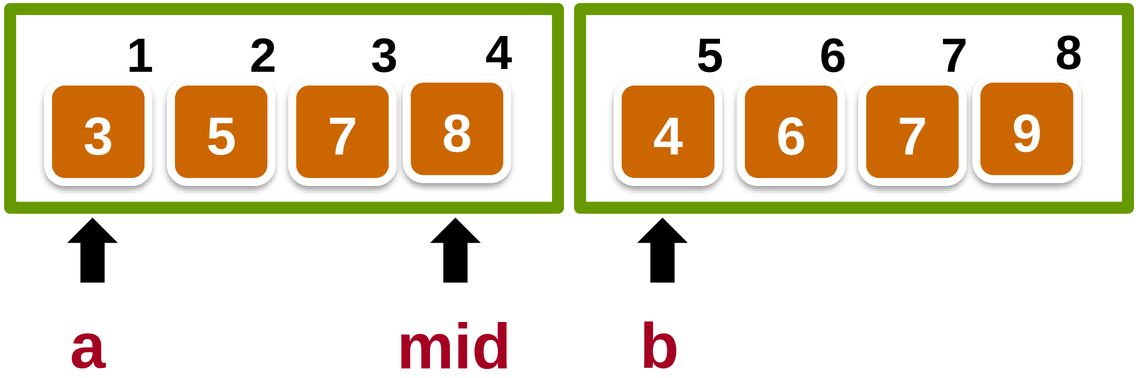
Mergesort Algorithm (Recap)

- Since merging is performed directly on the original array, swapping and shifting are needed
- mergesort() partitions a contiguous array of elements between index n and m into two subarrays
- Recursively partitions until m-n<=0, then merge the resulting two subarrays
- merge() function merges two sub-arrays of elements between index n and ‘mid’, and between ‘mid+1’ and m
- During merging, one element from each subarray is compared and the smaller one is inserted into new list

```
void mergesort(int n, int m)
{   …..
    merge(n, m);
}
```

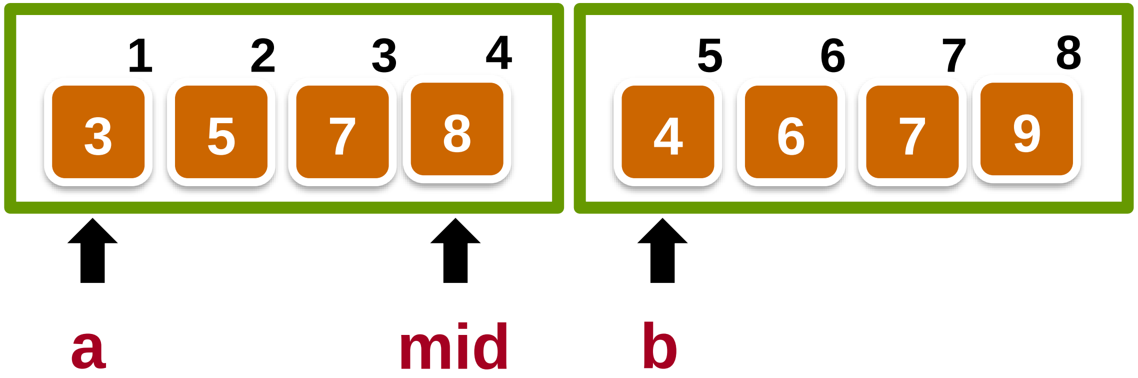
Mergesort Algorithm (Recap)

- Left subarray runs from n to ‘mid’ with a as running index; right subarray runs from mid+1 to m with b as running index



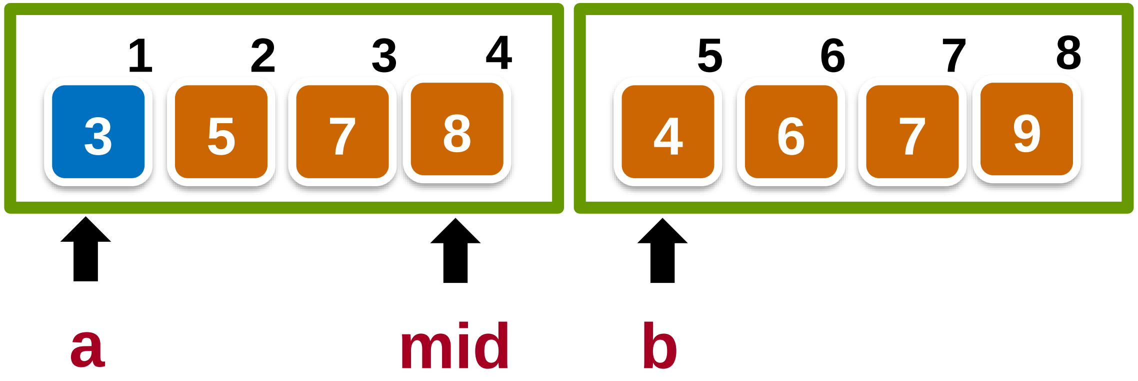
Mergesort Algorithm (Recap)

- Left subarray runs from n to ‘mid’ with a as running index; right subarray runs from mid+1 to m with b as running index
- slot[a] is the head element of left subarray, slot[b] is the head element of right subarray



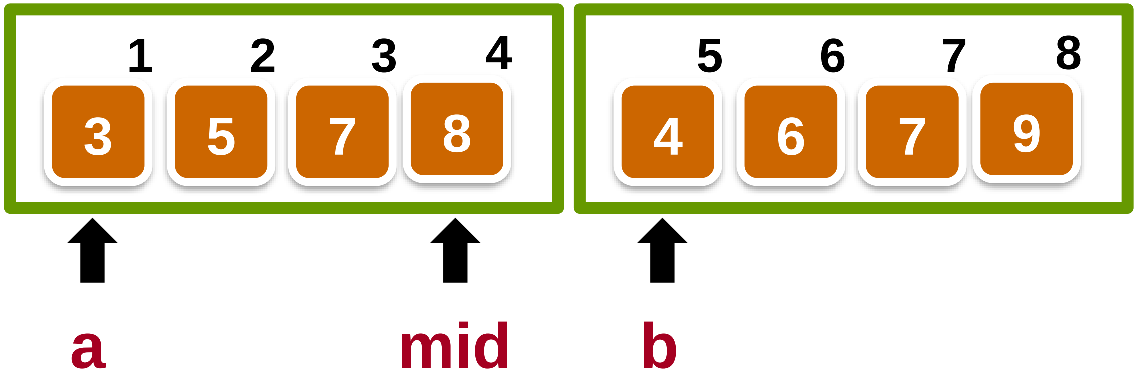
Mergesort Algorithm (Recap)

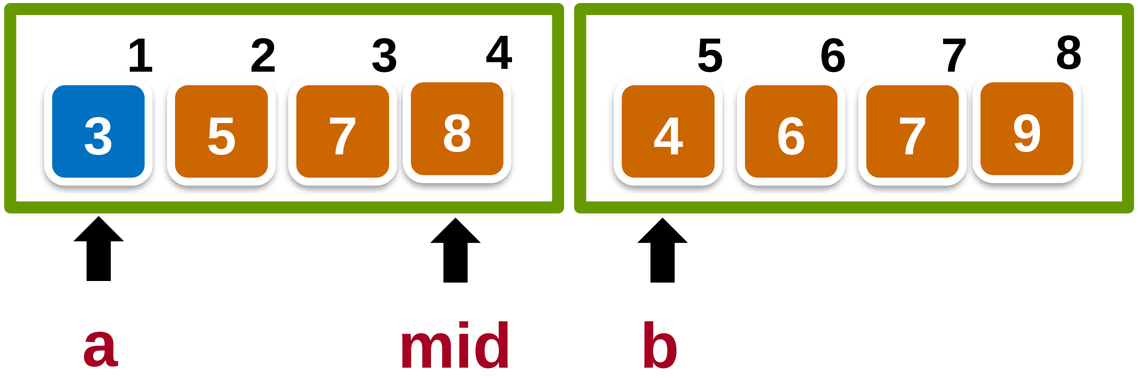
- Left subarray runs from n to 'mid' with a as running index; right subarray runs from mid+1 to m with b as running index
- slot[a] is the head element of left subarray, slot[b] is the head element of right subarray
- During merging, both left and right subarrays shrink towards the right to make space for the newly merged array



Mergesort Algorithm (Recap)

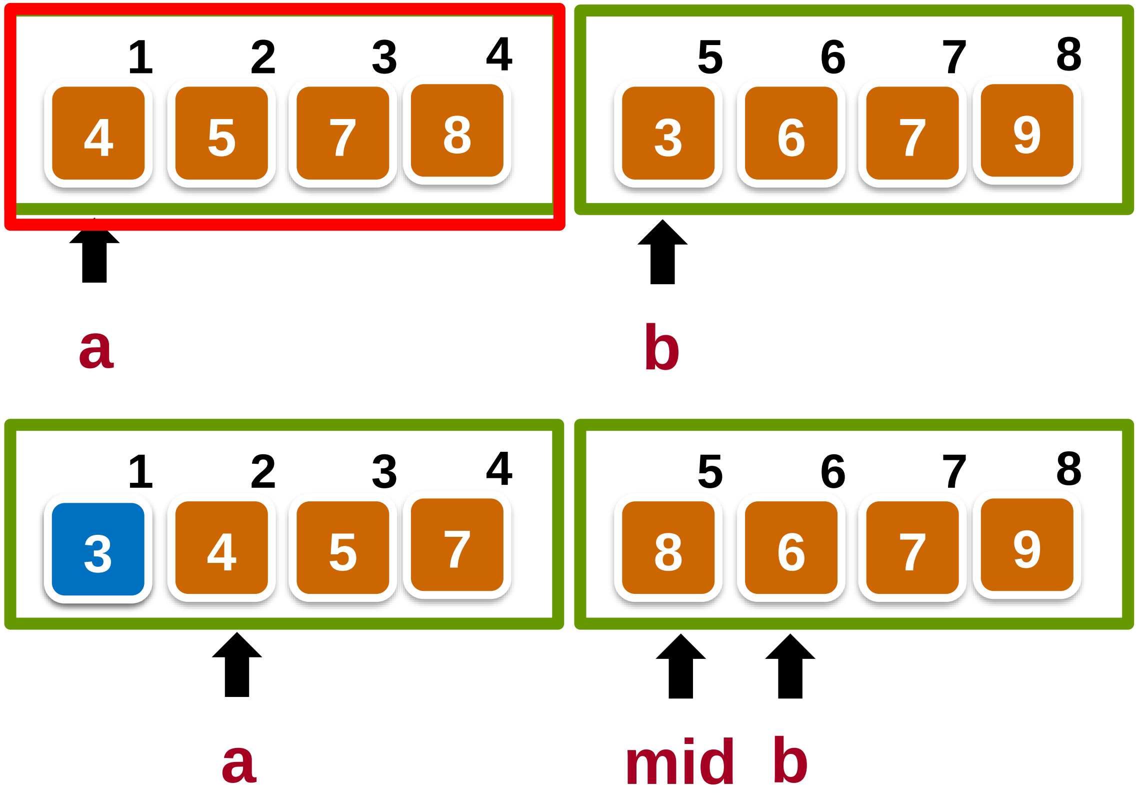
Case 1: if slot[a] < slot[b], there is nothing much to do since smaller element already in correct position (with regard to the merged array)





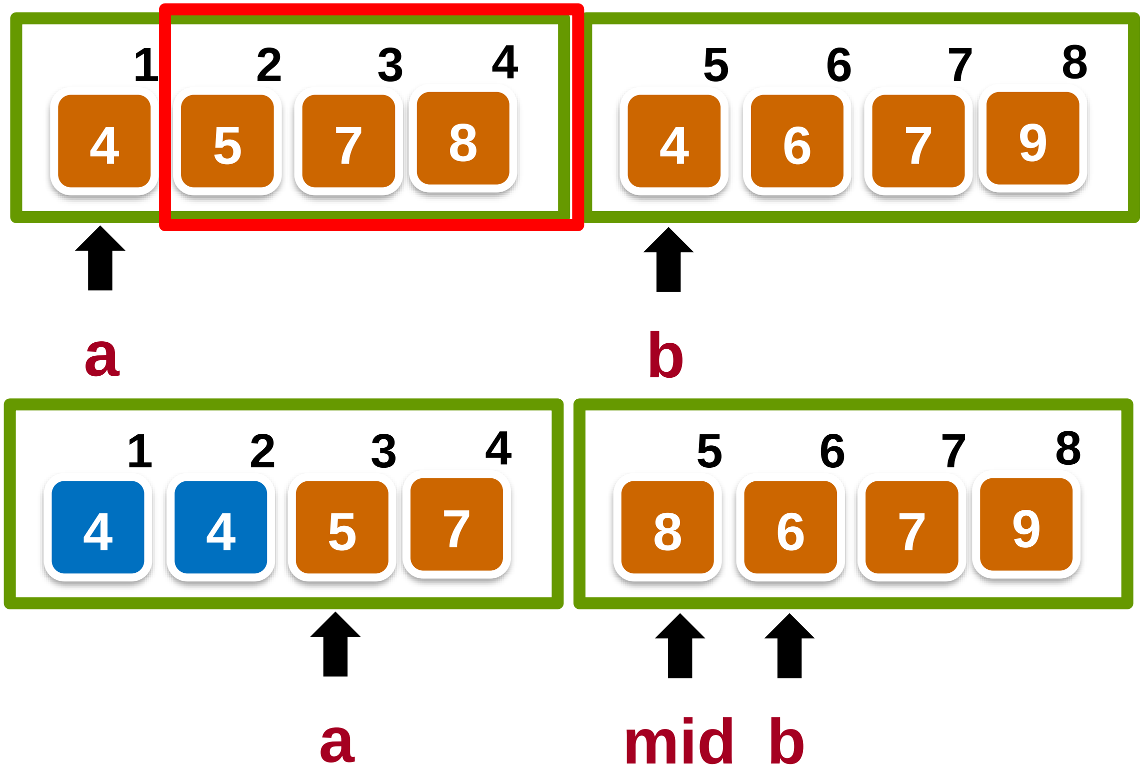
Mergesort Algorithm (Recap)

Case 2: if slot[a] > slot[b], then Right-shift (by one) elements of left subarray from index a to 'mid' and insert element at slot[b] into slot[a]



Mergesort Algorithm (Recap)

Case 3: if slot[a] == slot[b], then slot[a] is in the correct position. So, move slot[b] next to beside slot[a], by Right-shifting and swapping



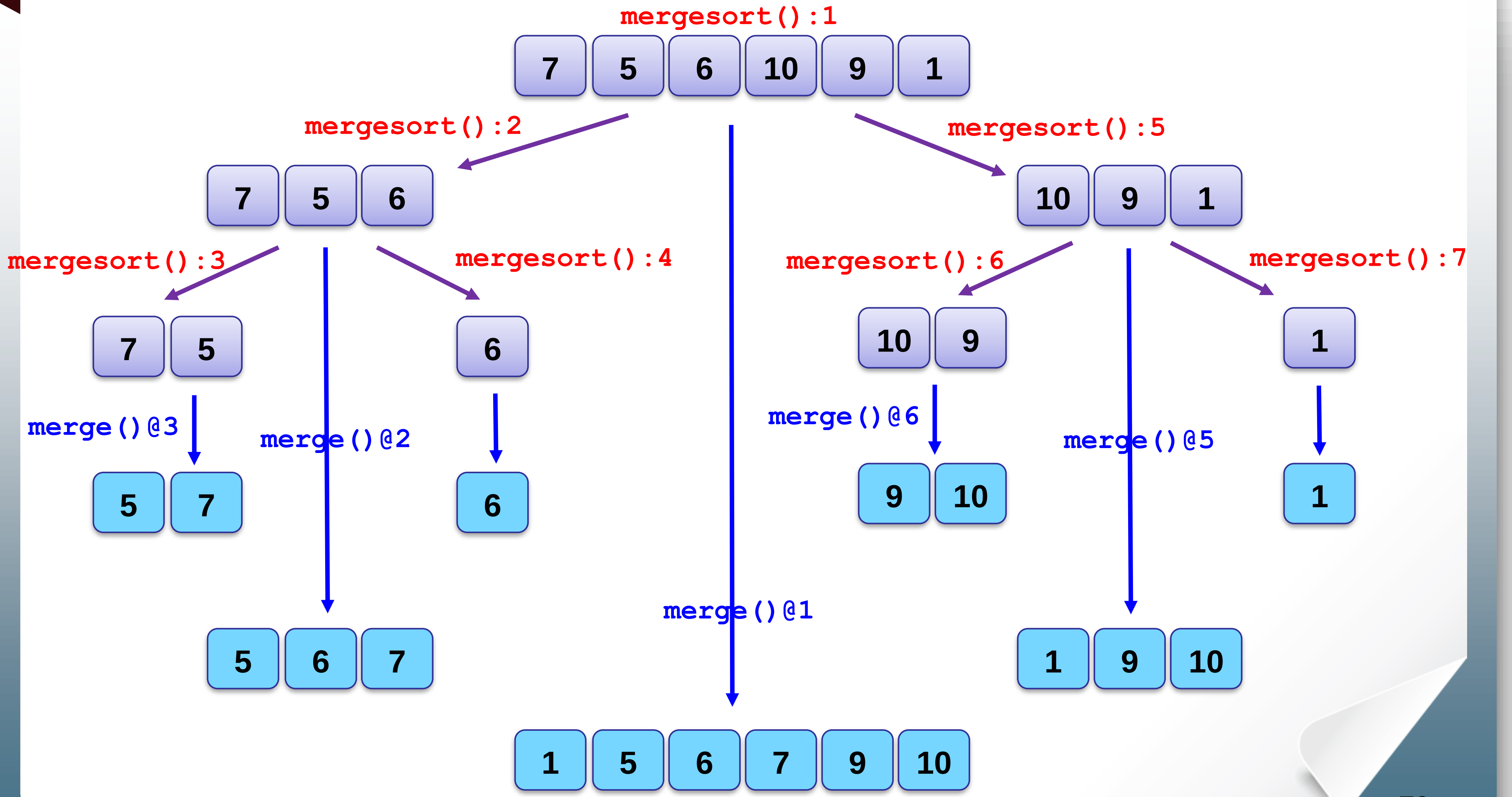
1 2 3 4 | 5 6 7 8

4 5 7 8 | 4 6 7 9

1 2 3 4 | 5 6 7 8

4 4 5 7 | 8 6 7 9

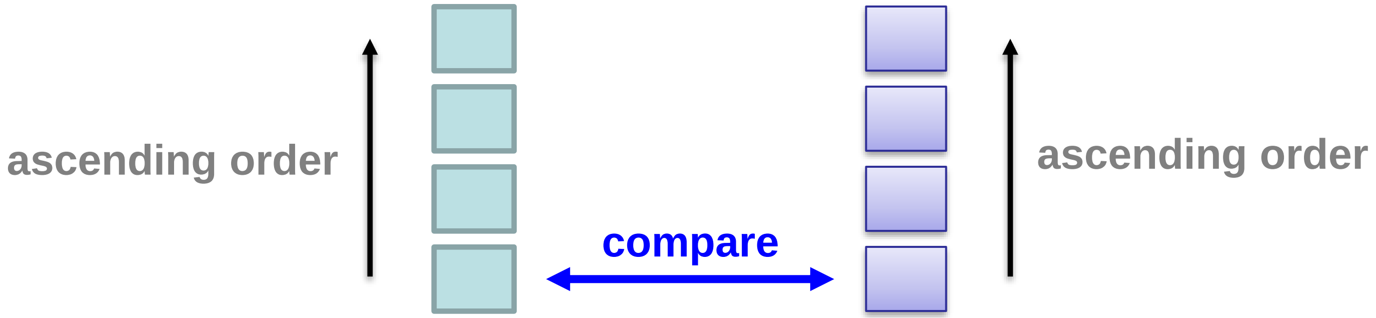
Call Graph of Mergesort



Complexity of Mergesort

Complexity of **merge()**

- After each comparison of keys from the two sub-lists, at least one element is moved to the new merged list and never compared again



- After the last key comparison, at least two elements will be moved into the merged list
- Thus, to merge two sub-lists of n elements in total, the number of key comparisons needed is **at most (worst case)** $n - 1$
  - How about best case?

Complexity of Mergesort

```
void mergesort(int s, int e) // s=start, e=end
{   int mid = (s+e)/2;
    if (e-s <= 0) return;            →   W(1) = 0
    else if (e-s > 1) {
        mergesort(s, mid);           →   W(n/2)
        mergesort(mid+1, e);         →   W(n/2)
    }
    merge(s, e);                     →   Worst case: n-1
}
```

$\}$ W(n)

Complexity of Mergesort

Mergesort performance (assume $n = 2^k$)

k = lg n

Worst case :

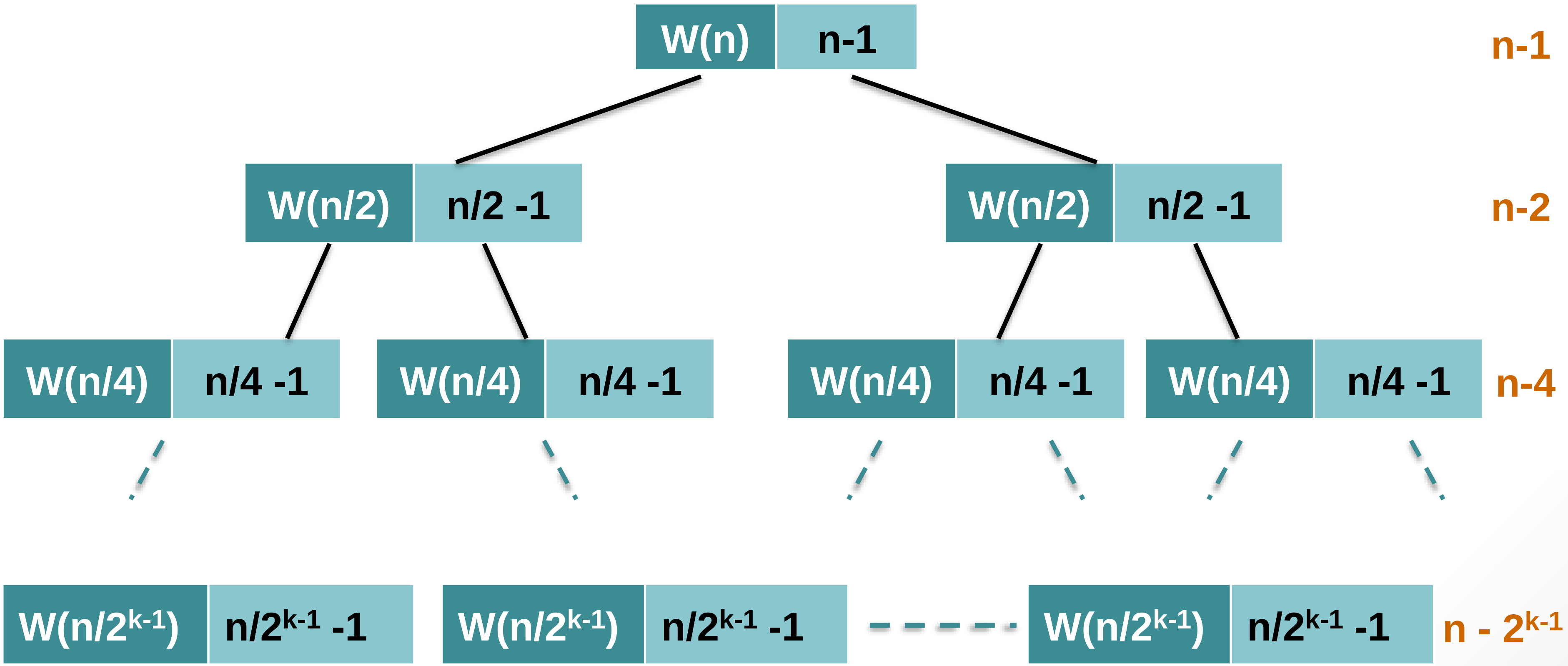
$W(1) = 0,$

$W(n) = W(n/2) + W(n/2) + n-1$ Or

$$
\begin{aligned}
W(2^k) &= 2W(2^{k-1}) + 2^k - 1 \\
&= 2(2W(2^{k-2}) + 2^{k-1} - 1) + 2^k - 1 \\
&= 2^2W(2^{k-2}) + 2^k - 2 + 2^k - 1 \\
&= 2^2(2W(2^{k-3}) + 2^{k-2} - 1) + 2^k - 2 + 2^k - 1 \\
&= 2^3W(2^{k-3}) + 2^k - 2^2 + 2^k - 2 + 2^k - 1 \\
&\ \ldots \\
&= 2^kW(2^{k-k}) + k2^k - (1 + 2 + 4 + \ldots + 2^{k-1}) \\
&= k2^k - (2^k - 1) \\
&= n \lg n - (n - 1) \\
&= O(n \lg n)
\end{aligned}
$$

Geometric series

Visually :Recursion Tree



Total: $O(n\lg n)$

W(2) = 2W(1) + 1 = 1

Height of tree is $k = O(\lg n)$

Evaluation of Mergesort

☺ **Strengths:**

- ☞ Simple and good runtime behavior
- ☞ Easy to implement when using linked list

☹ **Weaknesses:**

- ☞ Difficult to implement for contiguous data storage such as array without auxiliary storage (requires data movements during merging)

Summary

- Mergesort uses the **Divide and Conquer** approach.
  - It recursively divide a list into two halves of approximately equal sizes, until the sub-list is too small (no more than two elements).
  - Then, it recursively merges two sorted sub-lists into one sorted list.

- The worst-case running time for **merging** two sorted lists of total size n is $n - 1$ key comparisons.

- The running time of Mergesort is $O(n\lg n)$.

SC2001/CE2101/CZ2101: Algorithm Design and Analysis

Appendix

(Merge operation in Mergesort)

Instructor: Assoc. Prof. ZHANG Hanwang

Courtesy of Dr. Ke Yiping, Kelly’s slides

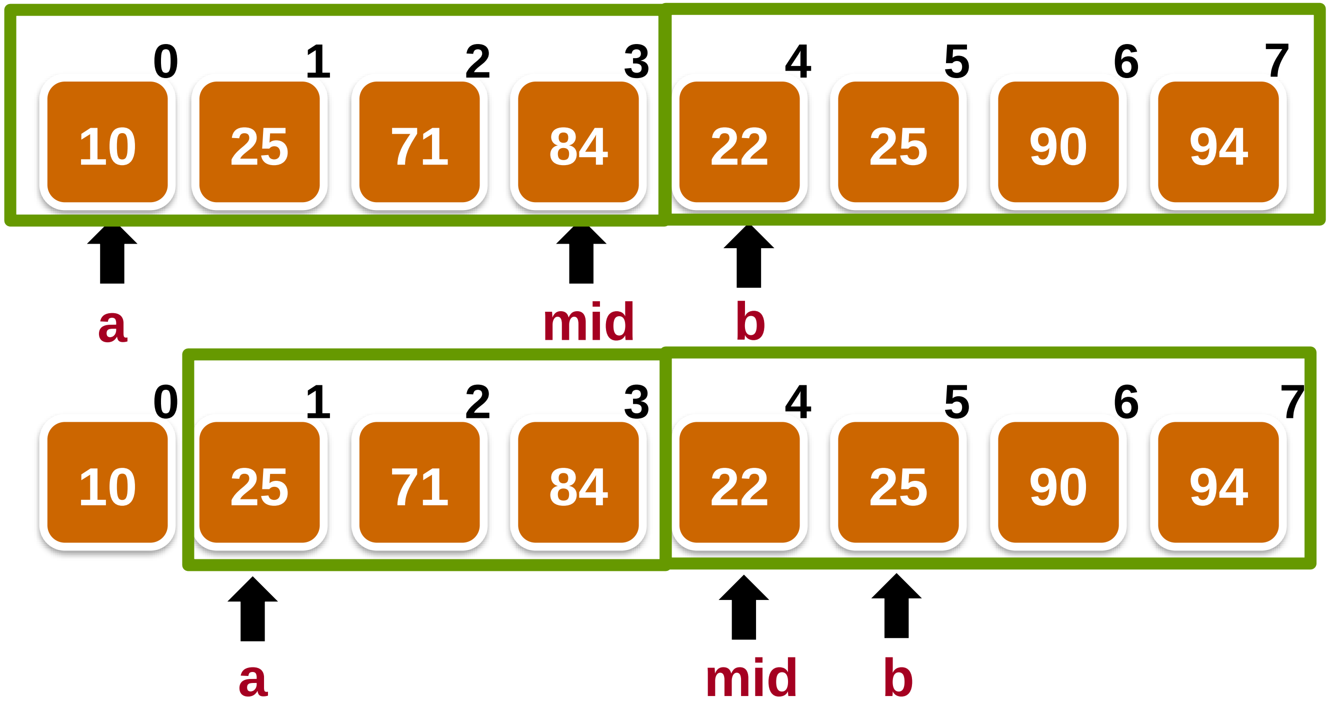
Merge Function

```
void merge(int n, int m)
  {
    int mid = (n+m)/2;
    int a = n, b = mid+1, i, tmp;
    if (m-n <= 0) return;
    while (a <= mid && b <= m) {
      cmp = compare(slot[a], slot[b]);
      if (cmp > 0) { //slot[a] > slot[b]
        tmp = slot[b++];
        for (i = ++mid; i > a; i--)
          slot[i] = slot[i-1];
```

Merge Function

```
                slot[a++] = tmp;
        } else if (cmp < 0) //slot[a] < slot[b]
                a++;
        else {   //slot[a] == slot[b]
                if (a == mid && b == m)
                    break;
                tmp = slot[b++];
                a++;
                for (i = ++mid; i > a; i--)
                    slot[i] = slot[i-1];
                slot[a++] = tmp;
        }
    } // end of while loop;
} // end of merge
```

Merge Operation



Parameters for merge:

n:0, m: 7

mid = (0+7)/2 = 3;

a = n; **b** = mid+1;

Comparison:

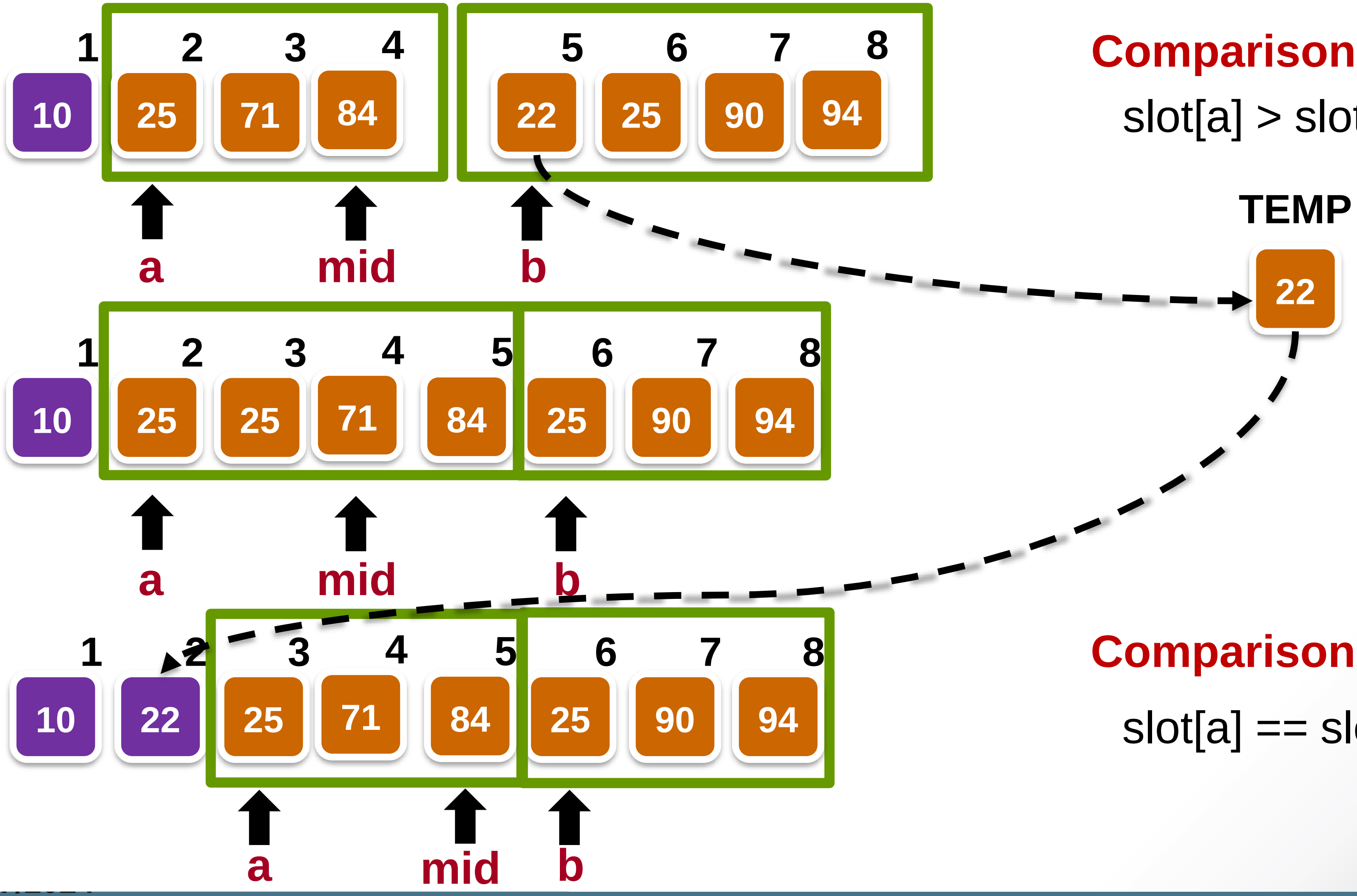
slot[a] < slot[b]

a : the 1st element of the 1st half

mid : the last element of the 1st half

b : the 1st element of the 2nd half

Merge Operation



1	2	3	4	5	6	7	8
10	25	71	84	22	25	90	94

1	2	3	4	5	6	7	8
10	25	25	71	84	25	90	94

1	2	3	4	5	6	7	8
10	22	25	71	84	25	90	94

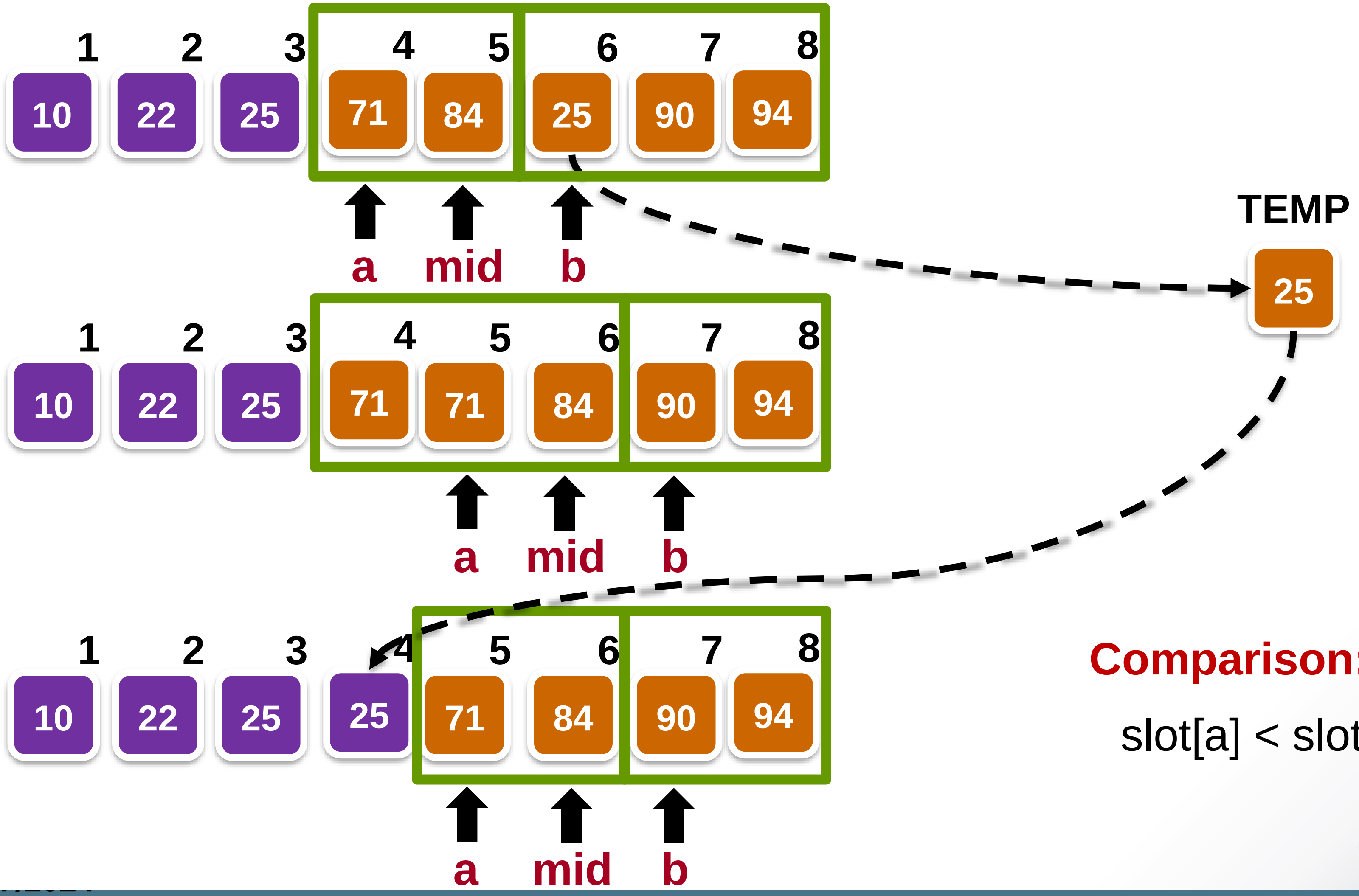
Comparison:

slot[a] > slot[b]

Comparison:

slot[a] == slot[b]

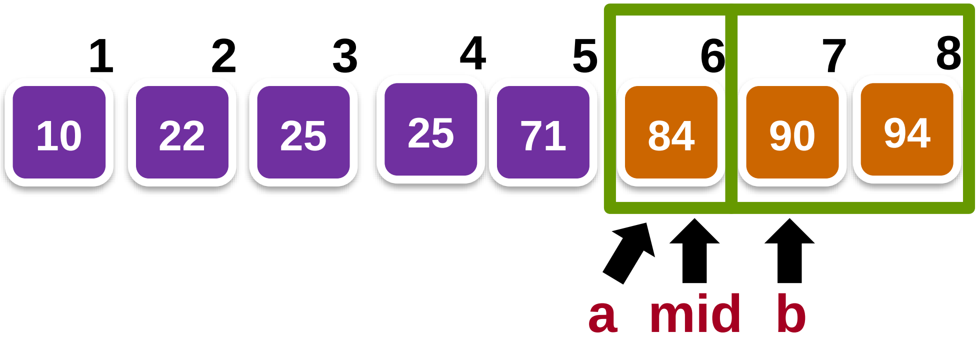
Merge Operation



Comparison:

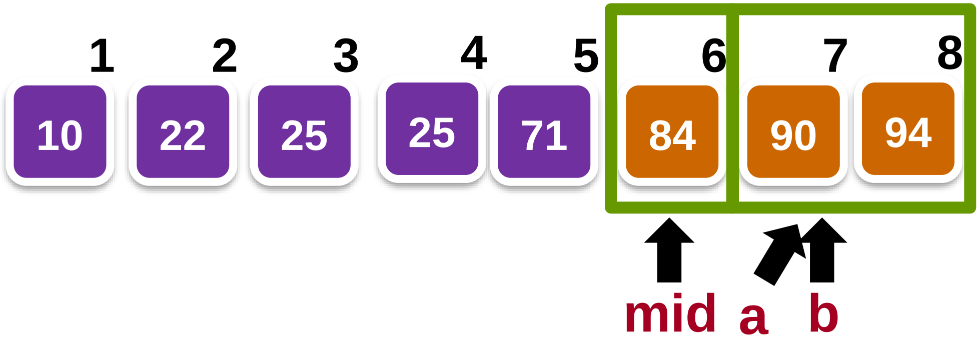
slot[a] < slot[b]

Merge Operation



Comparison:

slot[a] < slot[b]



1st half empty

Merge operation completed