Design and Analysis of Algorithms

# 03-07
# Divide – and – Conquer

## Merge Sort

**Imran Ihsan**
Assistant Professor, Department of Computer Science
Air University, Islamabad, Pakistan
www.imranihsan.com

# Example: Merge Sort

| 7 | 2 | 5 | 3 | 7 | 13 | 1 | 6 |
|---|---|---|---|---|---|---|---|

# Example: Merge Sort

| 7 | 2 | 5 | 3 | 7 | 13 | 1 | 6 |
|---|---|---|---|---|---|---|---|

Split the array into two halves

| 7 | 2 | 5 | 3 |
|---|---|---|---|

| 7 | 13 | 1 | 6 |
|---|---|---|---|

# Example: Merge Sort

| 7 | 2 | 5 | 3 | 7 | 13 | 1 | 6 |
|---|---|---|---|---|---|---|---|

Split the array into two halves

| 7 | 2 | 5 | 3 |
|---|---|---|---|

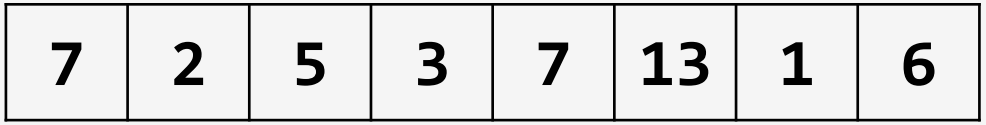
| 7 | 13 | 1 | 6 |
|---|---|---|---|

Sort the halves recursively

| 2 | 3 | 5 | 7 |
|---|---|---|---|

| 1 | 6 | 7 | 13 |
|---|---|---|---|

# Example: Merge Sort




Split the array into two halves


Sort the halves recursively


Merge the sorted halves into one array

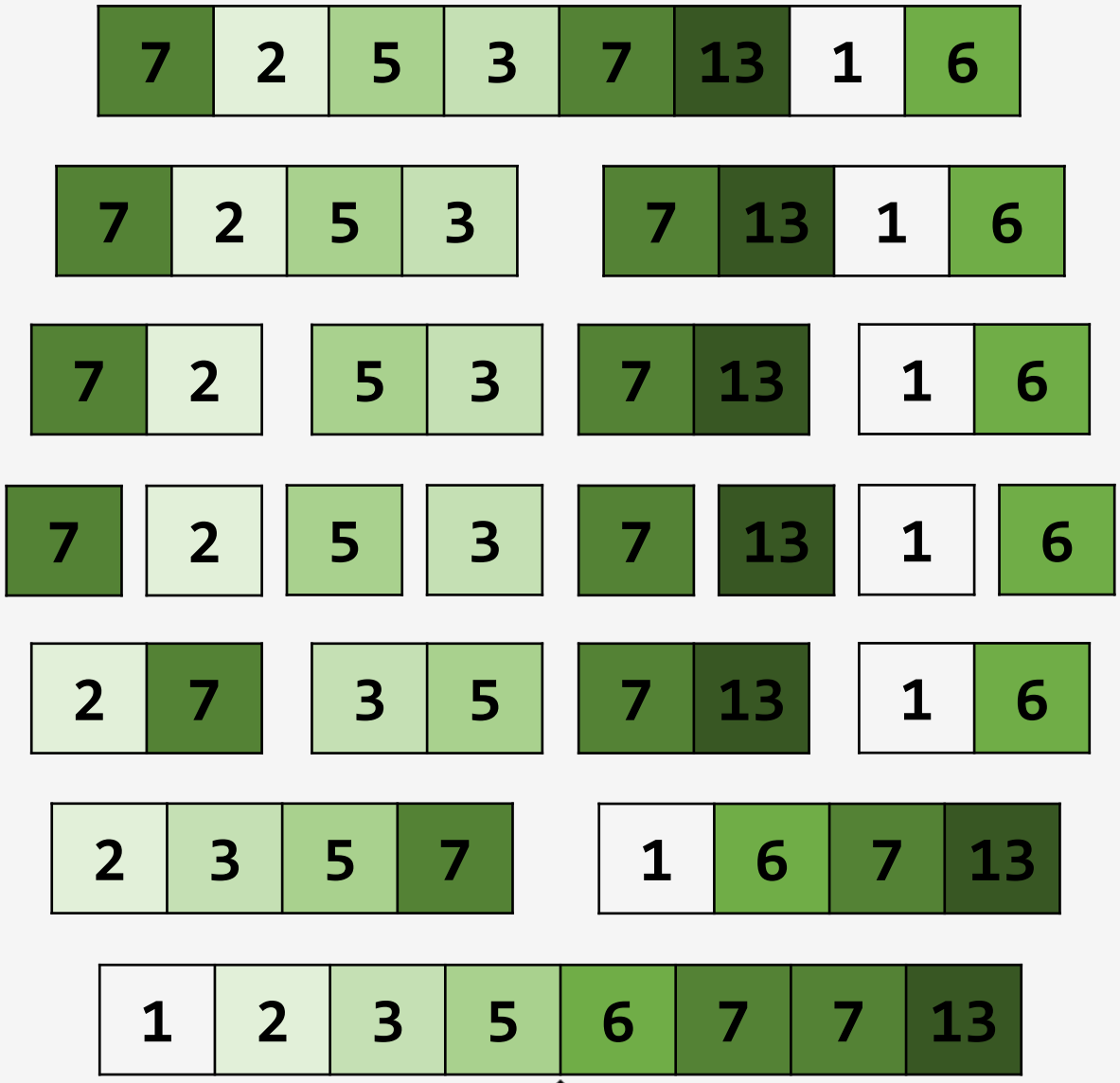
# Merge Sort

```
MergeSort(A[1 . . . n])
      if n = 1:
         return A
      m ← ⌊n/2⌋
      B ← MergeSort(A[1 . . .m])
      C ← MergeSort(A[m + 1 . . . n])
      A’ ← Merge(B, C)
      return A’
```

# Merging Two Sorted Arrays

```
Merge(B[1 . . . p], C[1 . . . q])
      {B and C are sorted}
      D ← empty array of size p + q
      while B and C are both non-empty:
         b ← the first element of B
         c ← the first element of C
         if b ≤ c:
            move b from B to the end of D
         else:
            move c from C to the end of D
      move the rest of B and C to the end of D
      return D
```

# Example: Merge Sort



# Merge Sort

Lemma

The running time of MergeSort(A[1 . . . n]) is O(nlogn).

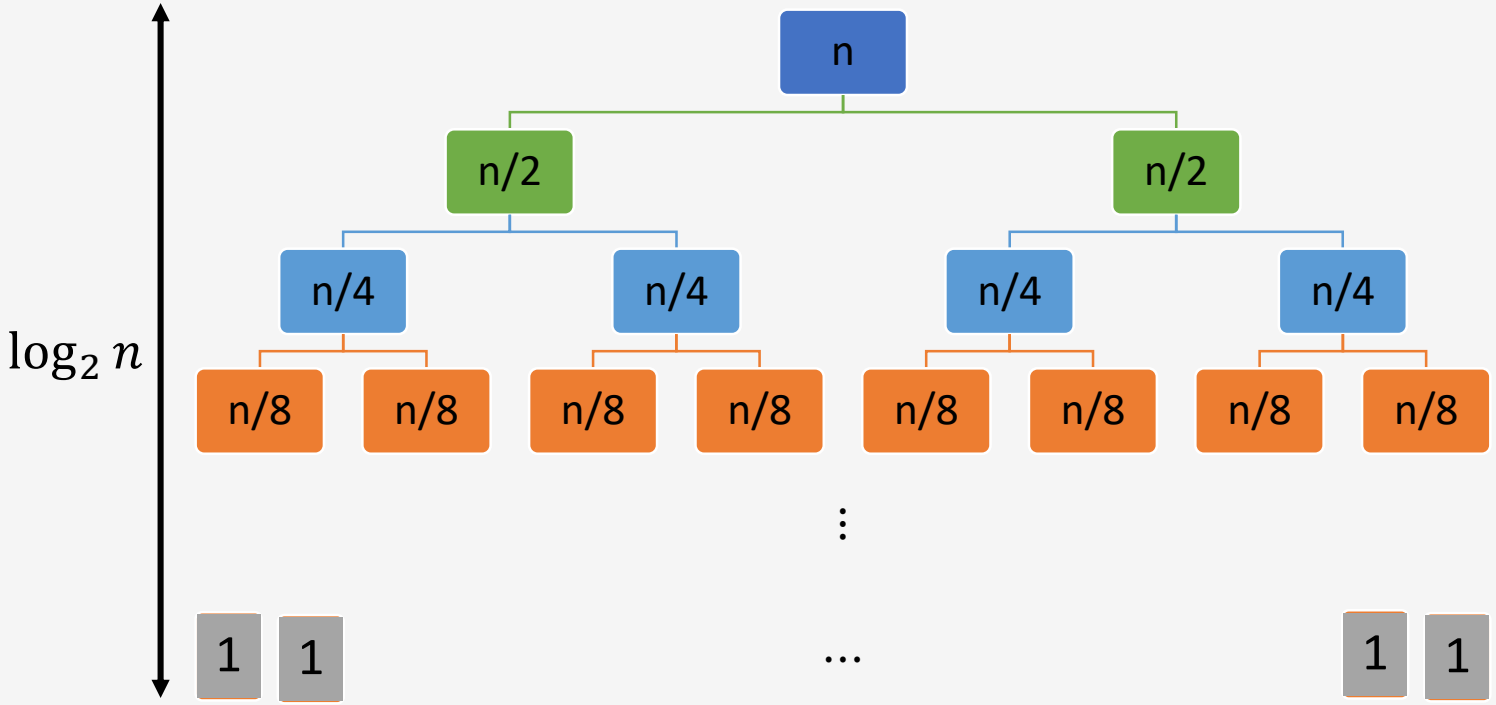
Proof

The running time of merging B and C is O(n).

Hence the running time of MergeSort(A[1 . . . n]) satisfies a recurrence

$$T(n) \leq 2T(n/2) + O(n)$$

# Recursion Steps




$$cn$$
$$+$$
$$2c\frac{n}{2} = cn$$
$$+$$
$$4c\frac{n}{4} = cn$$
$$+$$
$$\vdots$$

$$Total = cn \log_2 n$$

# Lower Bound for Comparison-Based Sorting

A comparison-based sorting algorithm sorts objects by comparing pairs of them.

Example: Selection Sort, Merge Sort

# Lower Bound for Comparison-Based Sorting

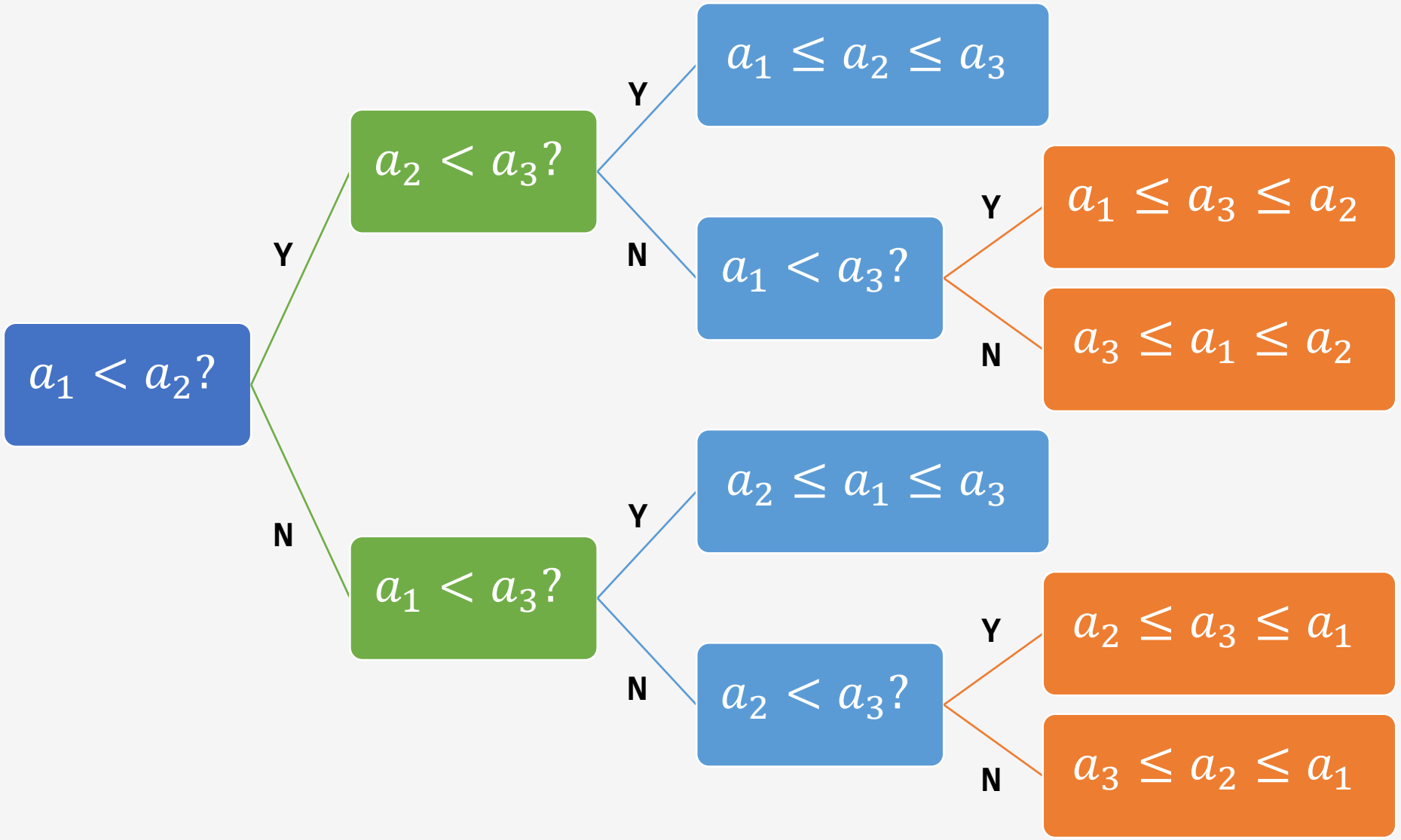
Lemma

Any comparison-based sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case to sort $n$ objects.

In other words

For any comparison-based sorting algorithm,
there exists an array $A[1 \ldots n]$ such that the
algorithm performs at least $\Omega(n \log n)$ comparisons to sort $A$.

# Decision Tree



# Estimating Tree Depth

the number of leaves $\ell$ in the tree must be at least $n!$
(the total number of permutations)

the worst-case running time of the algorithm
(the number of comparisons made) is at least the depth $d$

$d \geq \log_2 \ell$ (or, equivalently, $2^d \geq \ell$)

thus, the running time is at least

$$\log_2(n!) = \Omega(n \log n)$$

Lemma

$$\log_2(n!) = \Omega(n \log n)$$

Proof

$$\log_2(n!) = \log_2(1 \cdot 2 \cdot \cdots \cdot n)$$

$$\log_2(n!) = \log_2 1 + \log_2 2 + \cdots + \log_2 n$$

$$\log_2(n!) \geq \log_2 \frac{n}{2} + \cdots + \log_2 n$$

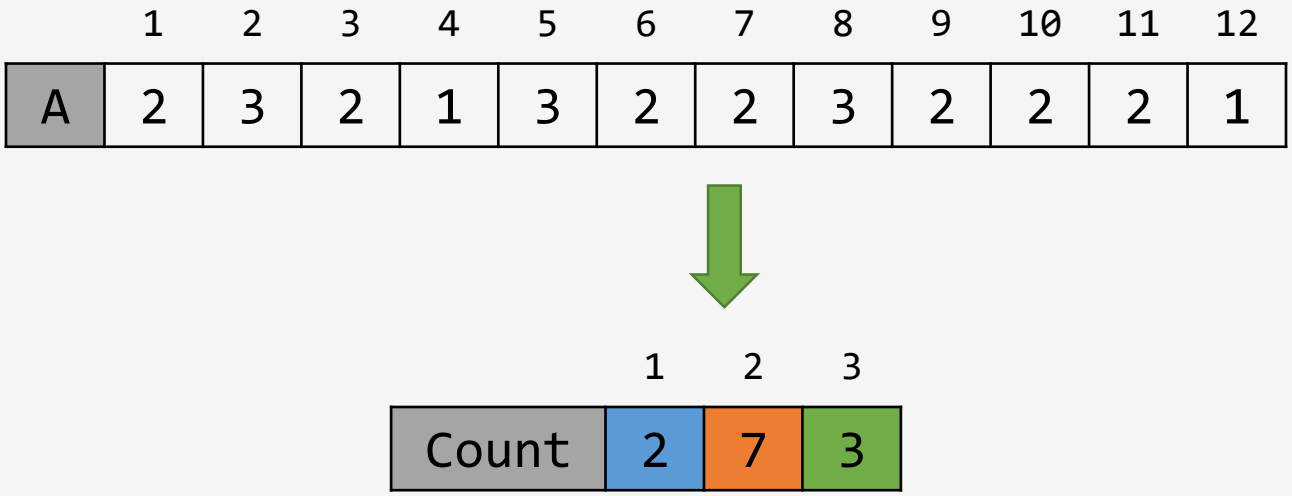
$$\log_2(n!) \geq \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)$$

# Non-Comparison-Based Sorting Algorithms

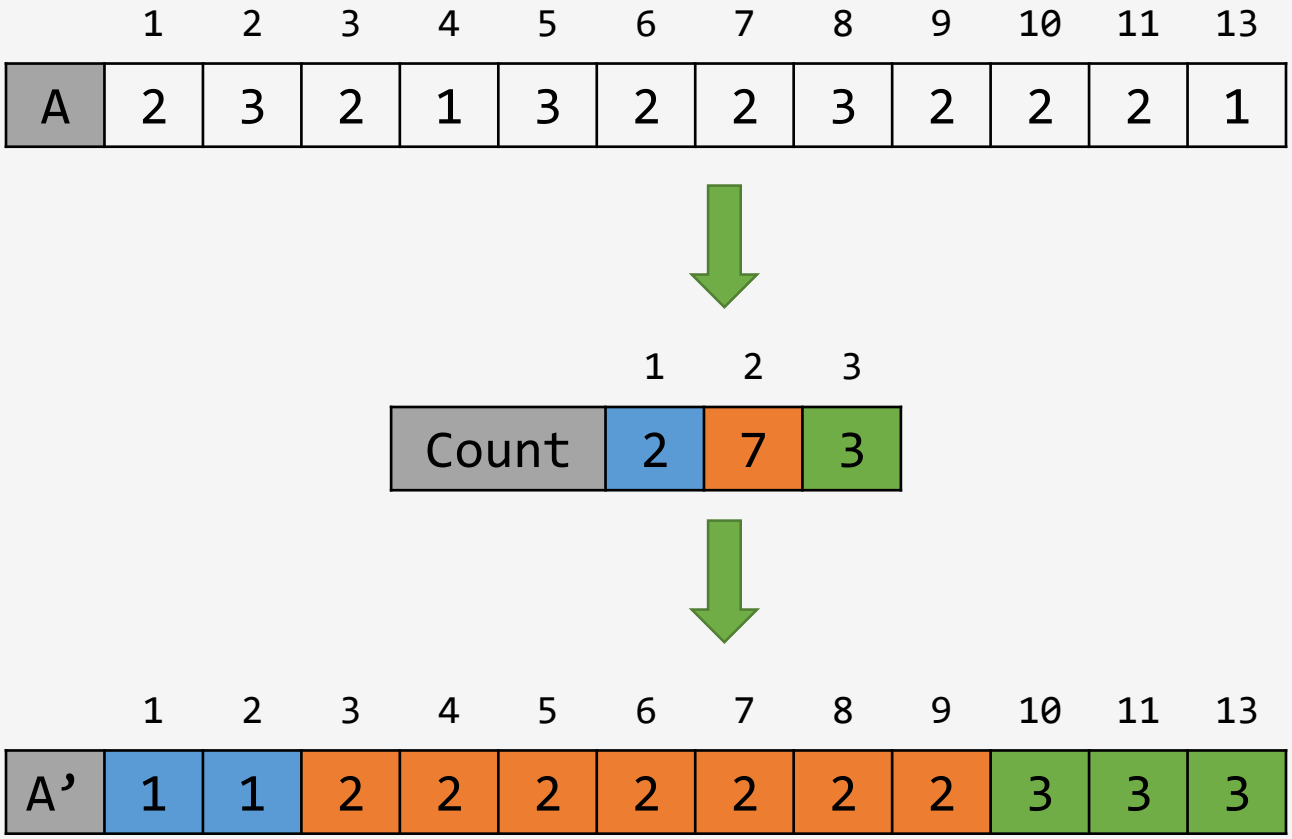
# Example: Sorting Small Integers

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 2 | 3 | 2 | 1 | 3 | 2 | 2 | 3 | 2 | 2 | 2 | 1 |

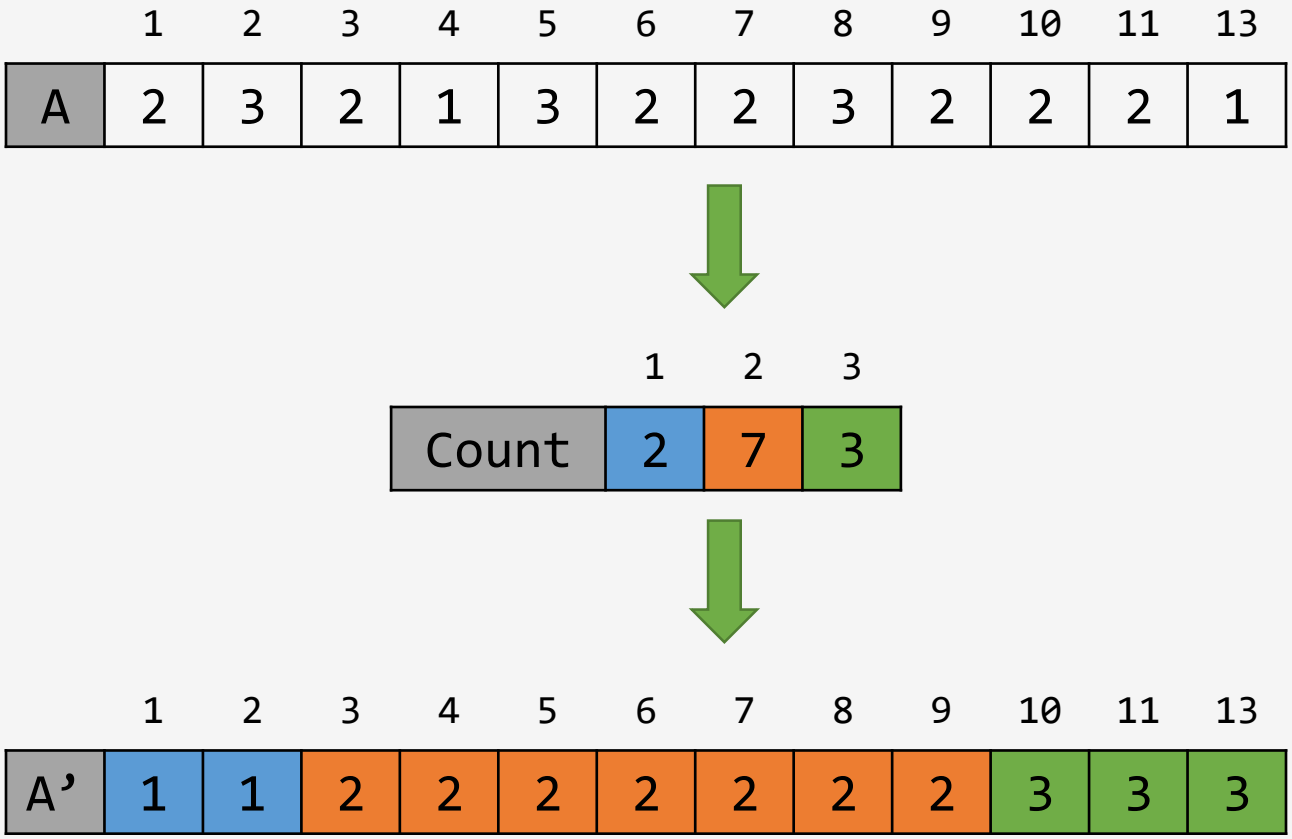
# Example: Sorting Small Integers



# Example: Sorting Small Integers



# Example: Sorting Small Integers




we have sorted these numbers without actually comparing them!

# Counting Sort: Ideas

Assume that all elements of $A[1 \ldots n]$ are integers from $1$ to $M$.

By a single scan of the array $A$, count the number of occurrences of each $1 \leq k \leq M$ in the array $A$ and store it in $Count[k]$.

Using this information, fill in the sorted array $A'$.

# Count Sort

```
CountSort(A[1 . . . n])
        Count[1 . . .M] ← [0, . . . , 0]
        for i from 1 to n:
            Count[A[i]] ← Count[A[i]] + 1
            {k appears Count[k] times in A}
        Pos[1 . . .M] ← [0, . . . , 0]
        Pos[1] ← 1
        for j from 2 to M:
            Pos[j] ← Pos[j – 1] + Count[j – 1]
        {k will occupy range [Pos[k]...Pos[k + 1] – 1]}
        for i from 1 to n:
            A’[Pos[A[i]]] ← A[i]
            Pos[A[i]] ← Pos[A[i]] + 1
```

# Count Sort

Lemma

$$Provided\ that\ all\ elements\ of\ A[1 \ldots n] are$$
$$in\ time\ O(n\ +\ M).\ integers\ from\ 1\ to\ M, CountSort(A)\ sorts\ A$$

Proof

$$If\ M\ =\ O(n), then\ the\ running\ time\ is\ O(n).$$

# Summary

Merge sort uses the divide-and-conquer strategy to sort an n-element array in time O(n log n).

No comparison-based algorithm can do this (asymptotically) faster.

One can do faster if something is known about the input array in advance (e.g., it contains small integers).