Searching algorithms

Searching algorithms

Check for an element or retrieve an element from any data structure where it is stored

Sequential Search
(Linear Search**)**

Interval Search
(Binary Search**)**

Linear Search

Problem: Given an array $arr[]$ of n elements, write a function to search a given element x in $arr[]$.

Examples :
Input : $arr[] = \{10, 20, 80, 30, 60, 50, 110, 100, 130, 170\}$
$x = 110;$
Output : 6
Element x is present at index 6

Input : $arr[] = \{10, 20, 80, 30, 60, 50, 110, 100, 130, 170\}$
$x = 175;$
Output : -1
Element x is not present in $arr[]$

Linear Search

A simple approach is to do a **linear search**, i.e

- Start from the leftmost element of $arr[]$ and one by one compare x with each element of $arr[]$
    - If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

Linear Search

```
def linear_search(arr, x):

    for i in range(0, len(arr)):
        if (arr[i] == x):
            return i
    return -1
```

```
arr = [2, 3, 4, 10, 40]
x = 10

result = linear_search(arr, x)
if(result == -1):
    print("Element is not present in array")
else:
    print("Element is present at index", result)
```

Binary Search

Problem: Given an array $arr[]$ of n **sorted** elements, write a function to search a given element x in $arr[]$.

Binary Search Approach:

Binary Search is a searching algorithm used in a sorted array by repeatedly **dividing the search interval in half**.

Binary Search

The basic steps to perform Binary Search are:

- Begin with an interval covering the whole array.
- If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
- Otherwise, narrow it to the upper half.
- Repeatedly check until the value is found, or the interval is empty.

Binary Search

Binary Search Algorithm: We basically ignore half of the elements just after one comparison.

- Compare x with the middle element.
- If x matches with the middle element,
  we return the mid index.
- Else If x is greater than the mid element,
  then x can only lie in the right half subarray after the mid element.
  So we recur for the right half.
  - Else (x is smaller) recur for the left half.

Binary Search

```
def binary_search(arr, l, r, x):

    if r >= l:
        mid = l + (r - l) // 2

        if arr[mid] == x:
            return mid

        elif arr[mid] > x:
            return binary_search(arr, l, mid-1, x)

        else:
            return binary_search(arr, mid+1, r, x)

    else:
        return -1
```

$O(\log(n))$ time

Jump Search

Like Binary Search, Jump Search is a searching algorithm for sorted arrays

The basic idea :
- To check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

Example :
- Suppose we have an array $arr[]$ of size n and block (to be jumped) size m.
- Then, we search at the indexes $arr[0], arr[m], arr[2m] \ldots\ldots arr[km]$ and so on.
- Once we find the interval ($arr[km] < x < arr[(k+1)m]$), we perform a linear search operation from the index km to find the element x.

Jump Search

Let's consider the following array:

$$(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610)$$

Length of the array is 16.

Assume : $x = 55$ and jump step is 4

1. Jump from index 0 to index 4;
2. Jump from index 4 to index 8;
3. Jump from index 8 to index 12;
4. Since the element at index 12 is greater than 55 we will jump back a step to come to index 8.
5. Perform linear search from index 8 to get the element 55.

Jump Search

What is the optimal block size to be skipped?

- In the worst case, we have to do $\boldsymbol{n/m}$ jumps
- If the last checked value is greater than the element to be searched for, we perform $\boldsymbol{m - 1}$ comparisons more for linear search.
- Therefore, the total number of comparisons in the worst case will be $((n/m) + m - 1)$.
- The value of the function $((n/m) + m - 1)$ will be minimum when $m = \sqrt{n}$.
- Therefore, the **best step size** is $\boldsymbol{m = \sqrt{n}}$.

```
def jump_search(arr , x , n ):

    # Finding block size to be jumped
    step = math.sqrt(n)

    # Finding the block where the element is present (if it is present)
    prev = 0
    while arr[int(min(step, n)-1)] < x:
        prev = step
        step += math.sqrt(n)
        if prev >= n:
            return -1

    # Doing a linear search for x in block beginning with prev.
    while arr[int(prev)] < x:
        prev += 1

       # If we reached next block or end of array, element is not present.
        if prev == min(step, n):
            return -1

        # If element is found
        if arr[int(prev)] == x:
             return prev

    return -1
```

$O(\sqrt{n})$ time

Interpolation Search

The Interpolation Search :
- is an improvement over Binary Search for instances, where the values in a sorted array are **uniformly distributed**.

Binary Search always goes to the middle element to check.

Interpolation search **may go to different locations** according to the value of the key being searched.

For example, if the value of the key is closer to the last element, interpolation search is likely to start search toward the end side.

Interpolation Search

To find the position to be searched, it uses the following formula.

// The idea of the formula is to return higher value of pos
// when element to be searched is closer to $arr[hi]$. And
// smaller value when closer to $arr[lo]$

$$\boldsymbol{pos} = \boldsymbol{lo} + [\,(\boldsymbol{x} - \boldsymbol{arr}[\boldsymbol{lo}]) * (\boldsymbol{hi} - \boldsymbol{lo}) \,/\, (\boldsymbol{arr}[\boldsymbol{hi}] - \boldsymbol{arr}[\boldsymbol{Lo}])\,]$$

$arr[]$ ==> Array where elements need to be searched
x ==> Element to be searched
lo ==> Starting index in $arr[]$
hi ==> Ending index in $arr[]$

Interpolation Search

The formula for pos can be derived as follows.

Let's assume that the elements of the array are linearly distributed.

General equation of line : $y = m * x + c$.
y is the value in the array and x is its index.

Now putting value of lo, hi and x in the equation
$arr[hi] = m * hi + c$ ----(1)
$arr[lo] = m * lo + c$ ----(2)
$x = m * pos + c$ ----(3)

$m = (arr[hi] - arr[lo])/(hi - lo)$

subtracting eqn (2) from (3)
$x - arr[lo] = m * (pos - lo)$
$lo + (x - arr[lo])/m = pos$
$pos = lo + (x - arr[lo]) * (hi - lo)/(arr[hi] - arr[lo])$

Interpolation Search

Algorithm

Same as binary search except the partition logic.

1. In a loop, calculate the value of “pos” using the probe position formula.
2. If it is a match, return the index of the item, and exit.
3. If the item is less than $arr[pos]$, calculate the probe position of the left sub-array. Otherwise calculate the same in the right sub-array.
4. Repeat until a match is found or the sub-array reduces to zero.

Interpolation Search

```
def interpolation_search(arr, lo, hi, x):

    # Since array is sorted, an element present
    # in array must be in range defined by corner
    if (lo <= hi and x >= arr[lo] and x <= arr[hi]):

        # Probing the position with keeping
        # uniform distribution in mind.
        pos = lo + ((hi - lo) // (arr[hi] - arr[lo]) * (x - arr[lo]))

        # Condition of target found
        if arr[pos] == x:
            return pos

        # If x is larger, x is in right subarray
        if arr[pos] < x:
            return interpolation_search(arr, pos + 1, hi, x)

        # If x is smaller, x is in left subarray
        if arr[pos] > x:
            return interpolation_search(arr, lo, pos - 1, x)
    return -1
```

Exponential Search

Exponential search involves two steps:

- Find range where the element is present
- Do Binary Search in the found range.

How to find the range where element may be present?

The idea is to start with subarray size 1, compare its last element with x,
then try size 2, then 4 and so on until last element of a subarray is not greater.

Once we find an index i (after repeated doubling of i), we know that the element must be present between $i/2$ and i (Why $i/2$? because we could not find a greater value in previous iteration)

Exponential Search

```
def exponential_search(arr, n, x):
    # IF x is present at first
    # location itself
    if arr[0] == x:
        return 0

    # Find range for binary search j by repeated doubling
    i = 1
    while i < n and arr[i] <= x:
        i = i * 2

    # Call binary search for the found range
    return binary_search( arr, i // 2, min(i, n-1), x)
```

Exponential Search

Time Complexity : $O(Log\ n)$
Auxiliary Space : The above implementation of Binary Search is recursive and requires $O(Log\ n)$ space. With iterative Binary Search, we need only $O(1)$ space.

Applications of Exponential Search:

- Exponential Binary Search is particularly useful for unbounded searches, where the size of the array is infinite.

- It works better when the element to be searched is closer to the first element.

Sorting algorithms

Selection Sort

Selection Sort

The selection sort algorithm:

- repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array:

- The subarray which is already sorted.
- Remaining subarray which is unsorted.

In every iteration of selection sort,

- The minimum element from the unsorted subarray is picked and moved to the sorted subarray.

Selection Sort

$arr[] = 64\ 25\ 12\ 22\ 11$

// Find the minimum element in $arr[0 \ldots 4]$
// and place it at beginning

$11 \quad 25 \quad 12 \quad 22 \quad 64$

// Find the minimum element in $arr[1 \ldots 4]$
// and place it at beginning of $arr[1 \ldots 4]$

$11 \quad 12 \quad 25 \quad 22 \quad 64$

// Find the minimum element in $arr[2 \ldots 4]$
// and place it at beginning of $arr[2 \ldots 4]$

$11 \quad 12 \quad 22 \quad 25 \quad 64$

// Find the minimum element in $arr[3 \ldots 4]$
// and place it at beginning of $arr[3 \ldots 4]$

$11 \quad 12 \quad 22 \quad 25 \quad 64$

Selection Sort

Time Complexity: $O(n^2)$

Auxiliary Space: $O(1)$

Advantages:

- it never makes more than $O(n)$ swaps
- can be useful when memory write is a costly operation.

```
def selection_sort(A):

        # Traverse through all array elements
        for i in range(len(A)):

            # Find the minimum element in remaining unsorted array
            min_idx = i
            for j in range(i+1, len(A)):
                if A[min_idx] > A[j]:
                    min_idx = j

            # Swap the found minimum element with the first element
            A[i], A[min_idx] = A[min_idx], A[i]
        return A
```

Bubble Sort

Bubble Sort

Bubble Sort :
One of the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order

First Pass:
(5 1 4 2 8) –> (1 5 4 2 8), compares and swaps since 5 > 1.
(1 5 4 2 8) –> (1 4 5 2 8), Swap since 5 > 4
(1 4 5 2 8) –> (1 4 2 5 8), Swap since 5 > 2
(1 4 2 5 8) –> (1 4 2 5 8), already in order (8 > 5)

Second Pass:
(1 4 2 5 8) –> (1 4 2 5 8)
(1 4 2 5 8) –> (1 2 4 5 8), Swap since 4 > 2
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)

Third Pass:
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed.

The algorithm needs one whole pass without any swap to know it is sorted.

Bubble Sort

Time Complexity: $O(n^2)$

Auxiliary Space: $O(1)$

```
def bubble_sort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
      return arr
```

Bubble Sort

Optimized Implementation:

It can be optimized by stopping the algorithm if inner loop didn't cause any swap.

Worst and Average Case Time Complexity: $O(n^2)$.
Worst case occurs when array is reverse sorted.

Best Case Time Complexity: $O(n)$.
Best case occurs when array is already sorted.

```
def bubble_sort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        swapped = False

        # Last i elements are already
        #  in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to
            # n-i-1. Swap if the element
            # found is greater than the
            # next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True

        # IF no two elements were swapped
        # by inner loop, then break
        if swapped == False:
            break
```

Insertion Sort

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.

The array is virtually split into a sorted and an unsorted part.

Values from the unsorted part are picked and placed at the correct position in the sorted part.

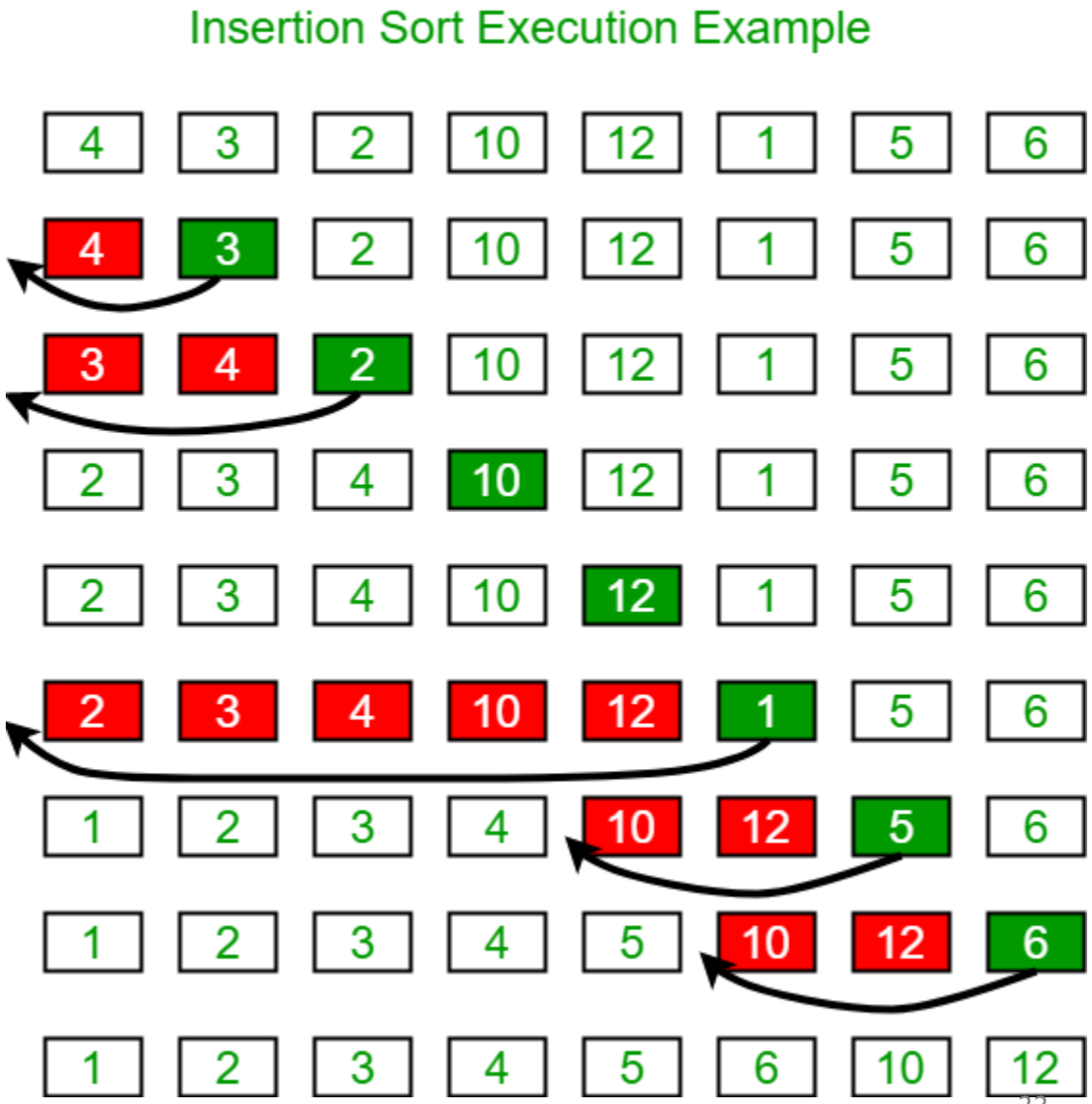
Insertion Sort

Algorithm
To sort an array of size n in ascending order:

1: Iterate over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.



Insertion Sort

Time Complexity: $O(n^2)$

Auxiliary Space: $O(1)$

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

```
def insertion_sort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        current = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and current < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = current
     return arr
```

Insertion Sort

Exercise:

Use Binary Search to find the proper location to insert the selected item at each iteration.

- In normal insertion sort, it takes $O(n)$ comparisons (at n^{th} iteration) in the worst case.
- We can reduce it to $O(\log n)$ by using binary search.

```
def binary_search(a, item, low, high):
    while (low <= high):
        mid = low + (high - low) // 2
        if (item == a[mid]):
            return mid + 1
        elif (item > a[mid]):
            low = mid + 1
        else:
            high = mid - 1
    return low
```

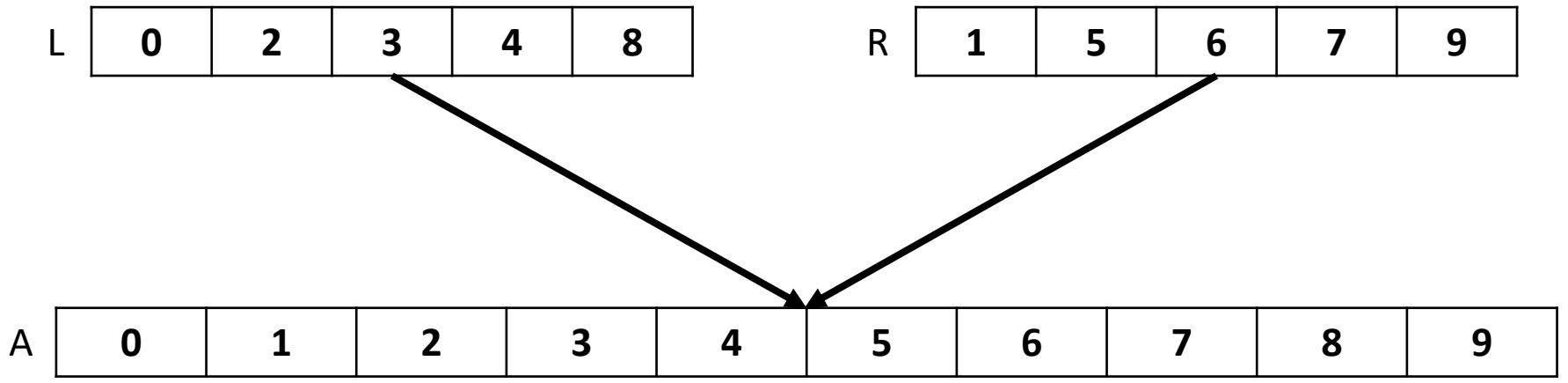
```
# Function to sort an array a[] of size 'n'
def insertion_sort(a, n):
    for i in range (1, n):
        j = i - 1
        selected = a[i]

        # find location where selected should be inserted
        loc = binary_search(a, selected, 0, j)

        # Move all elements after location to create space
        while (j >= loc):
            a[j + 1] = a[j]
            j-=1
        a[j + 1] = selected
    return a
```

Merge Sort

Merge Sort

Basic idea:

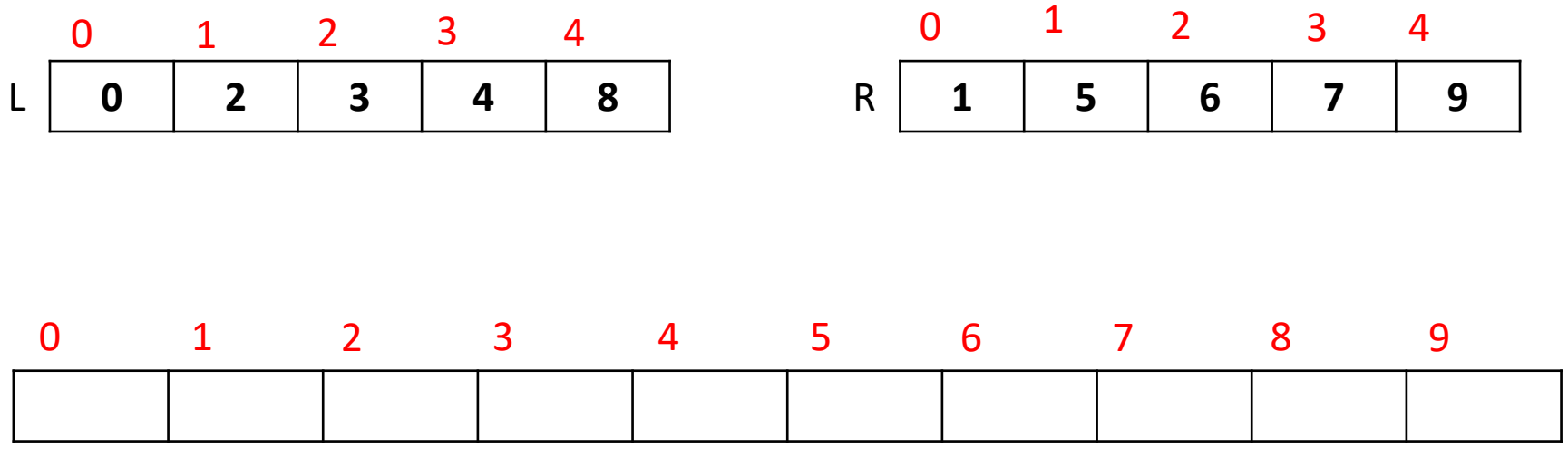
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

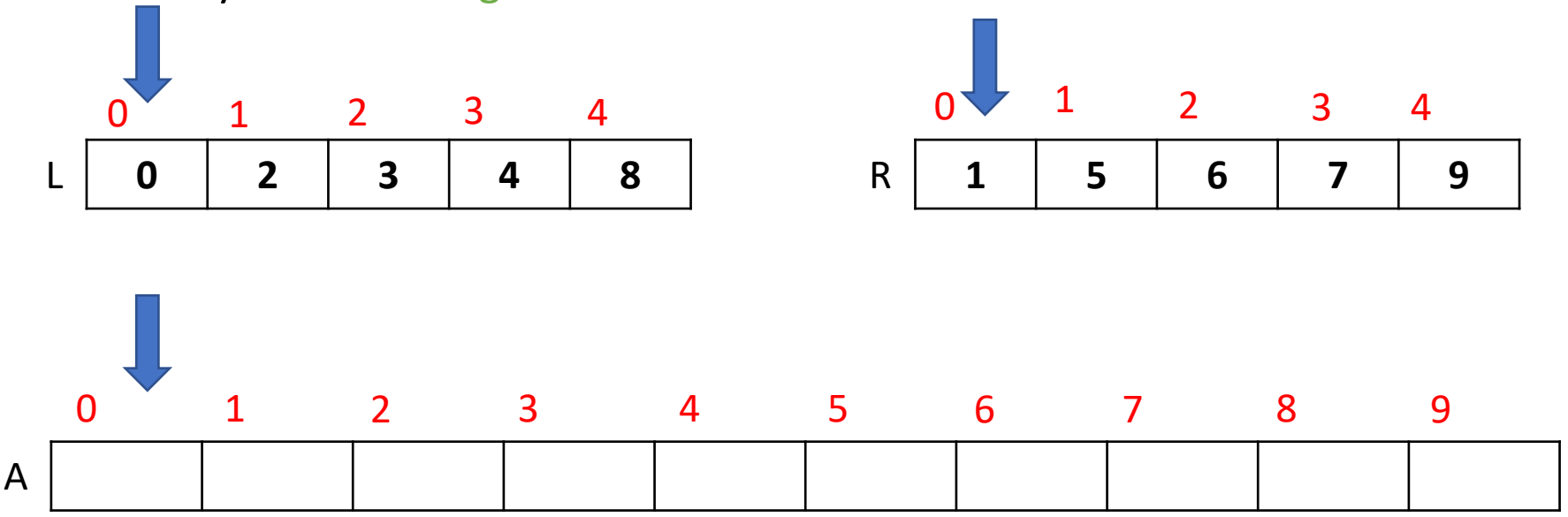
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

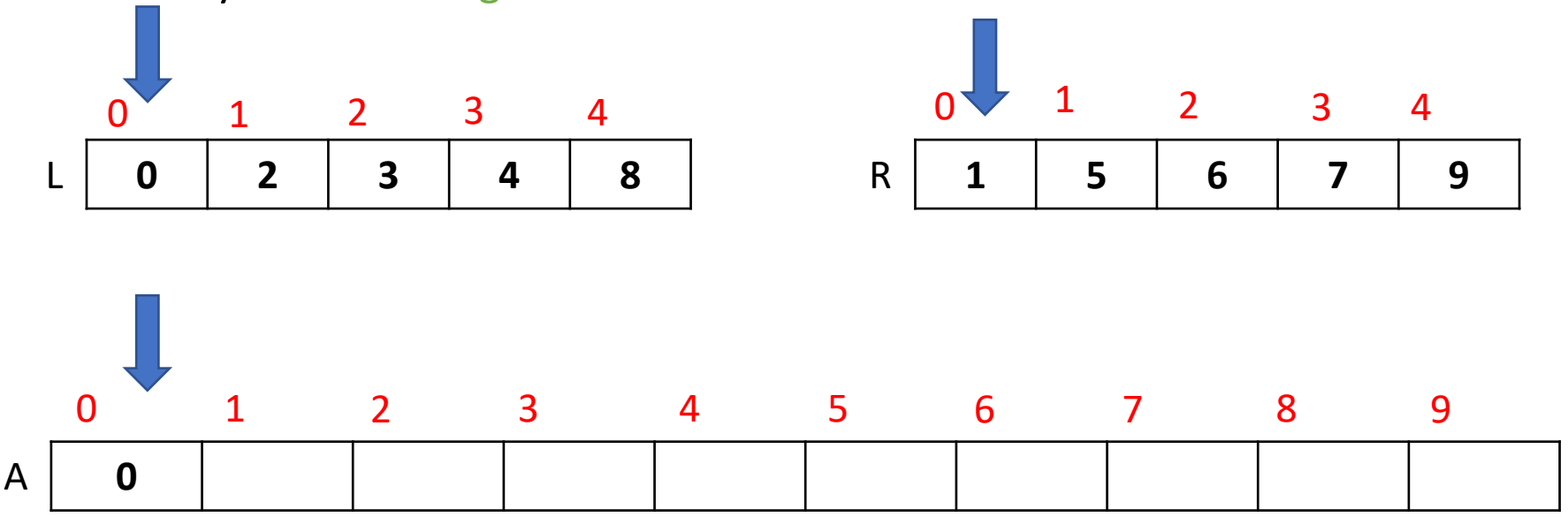
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

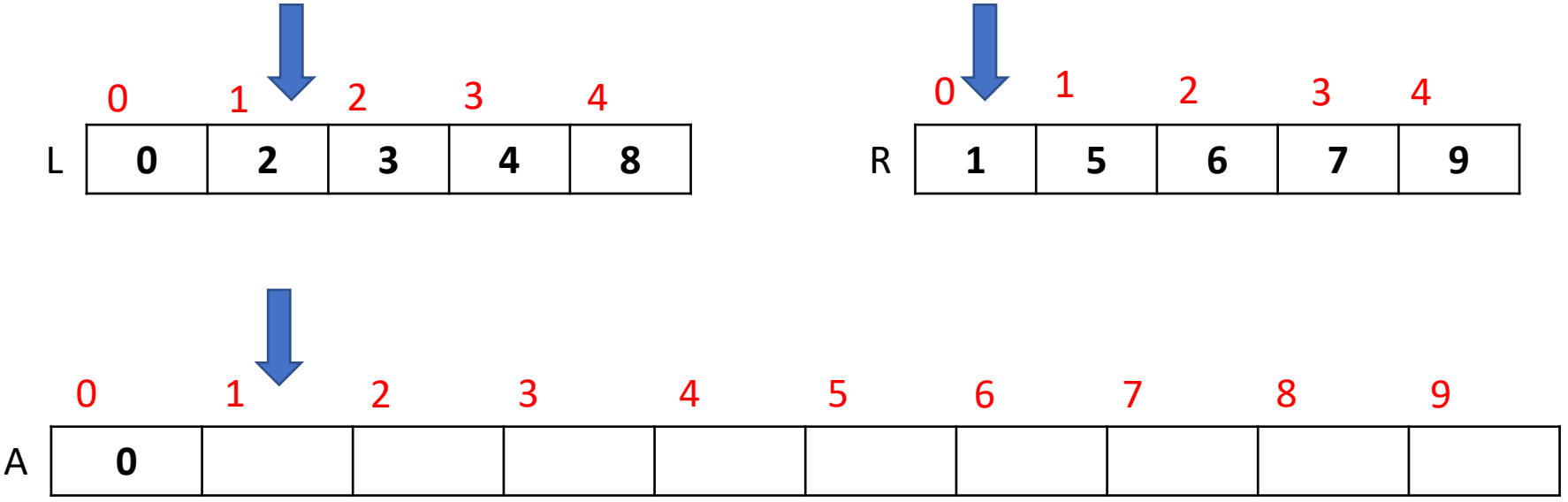
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

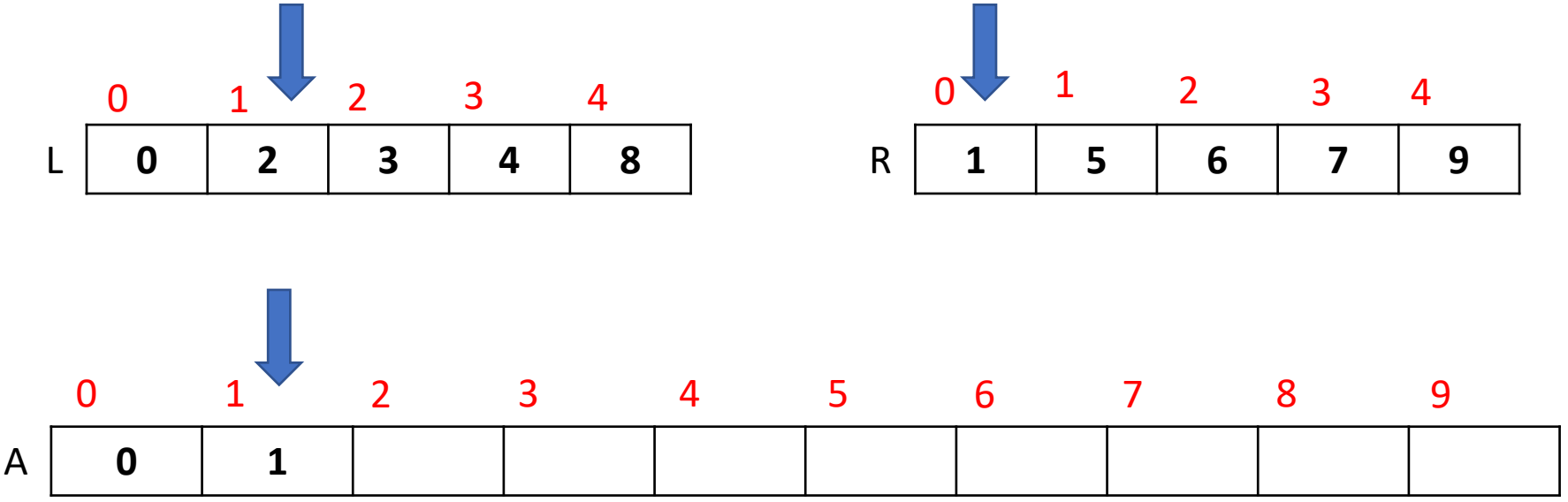
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

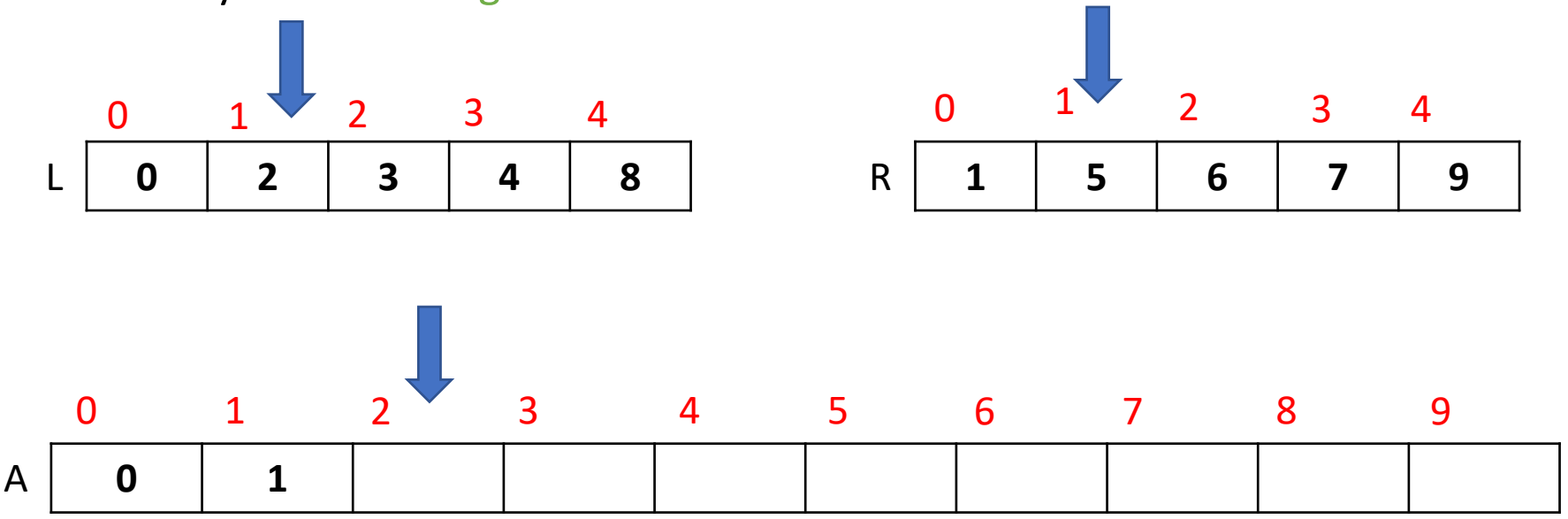
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

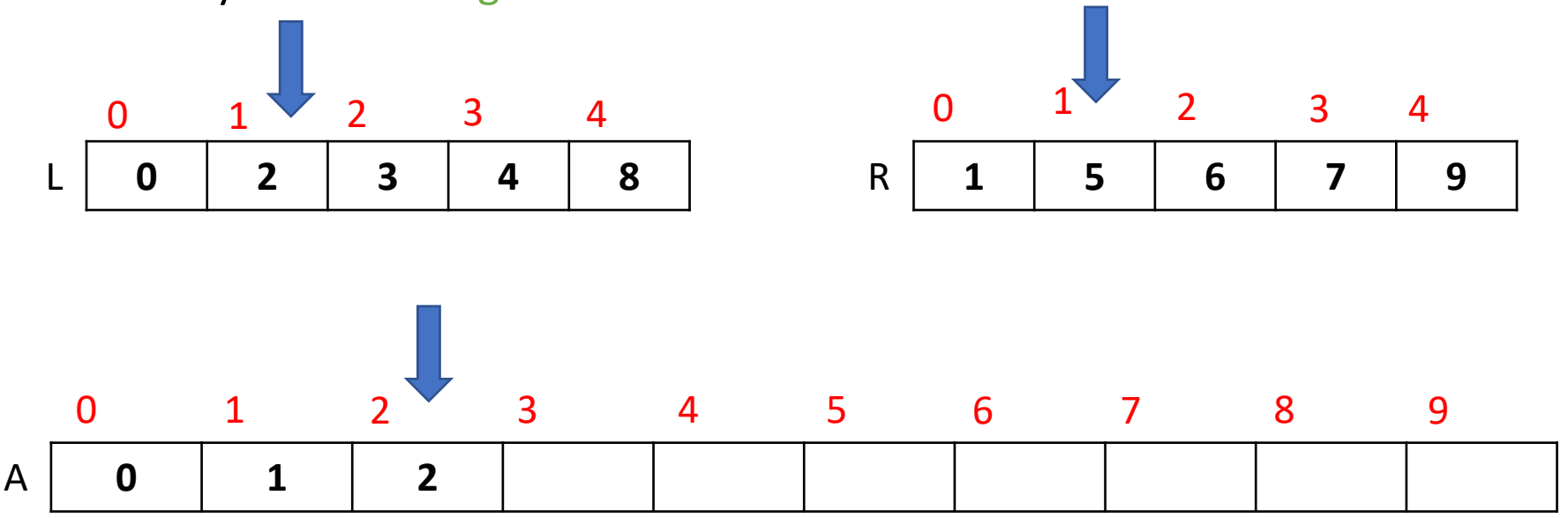
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

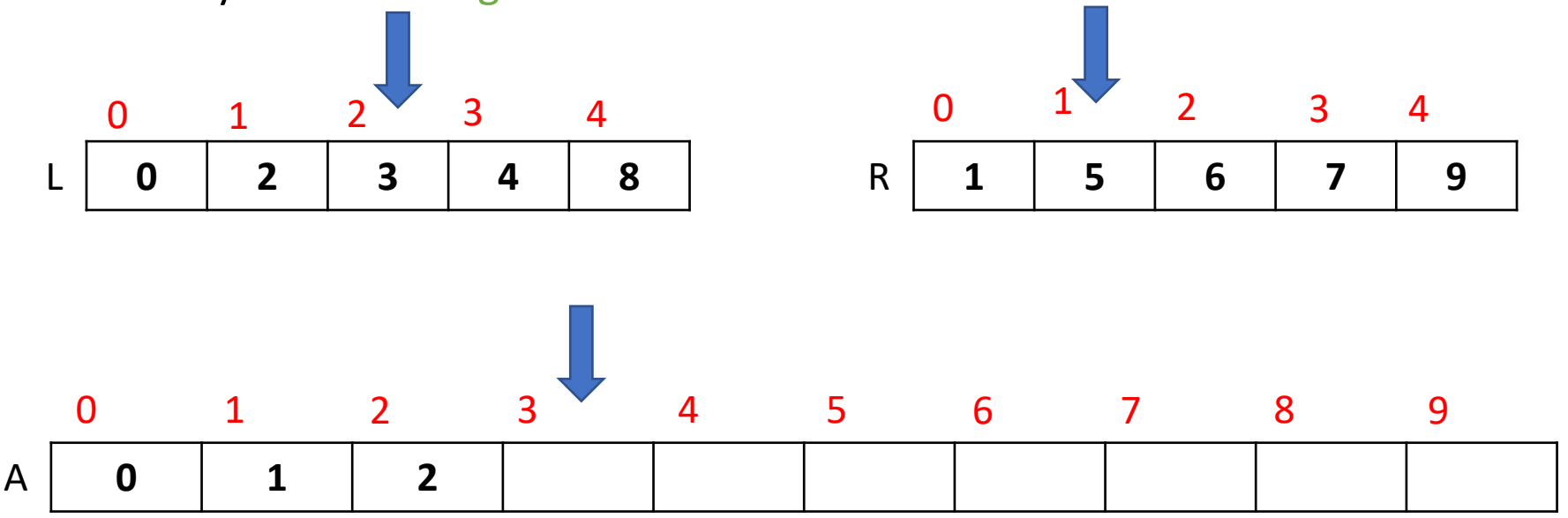
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

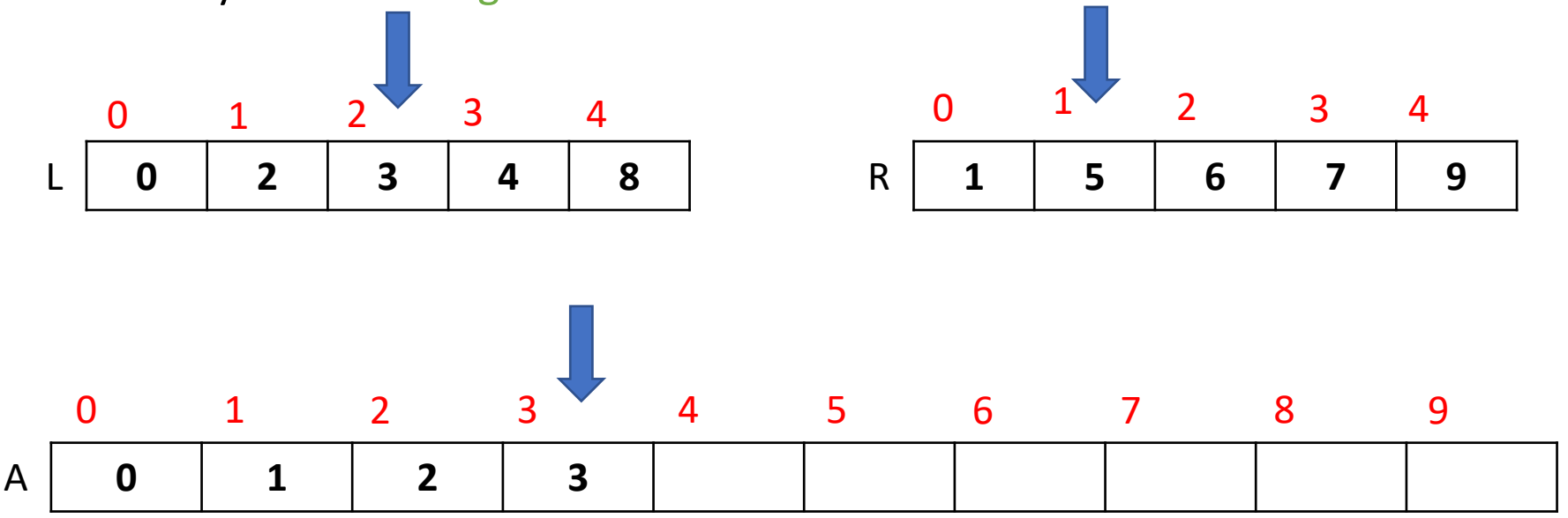
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

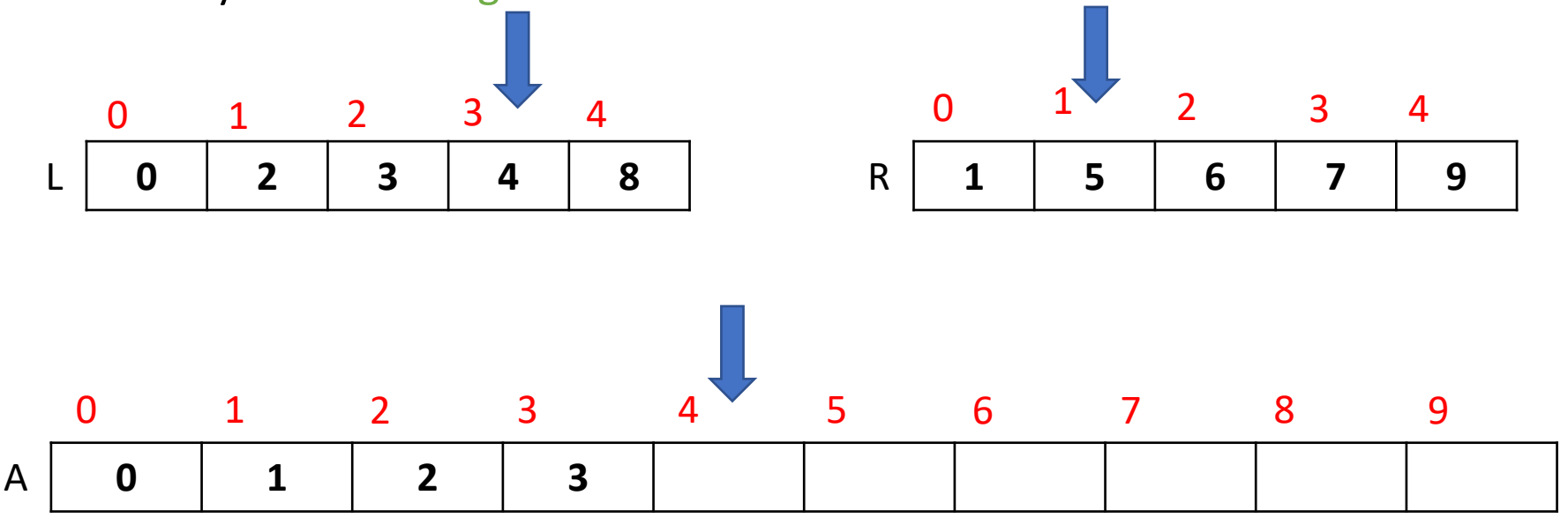
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

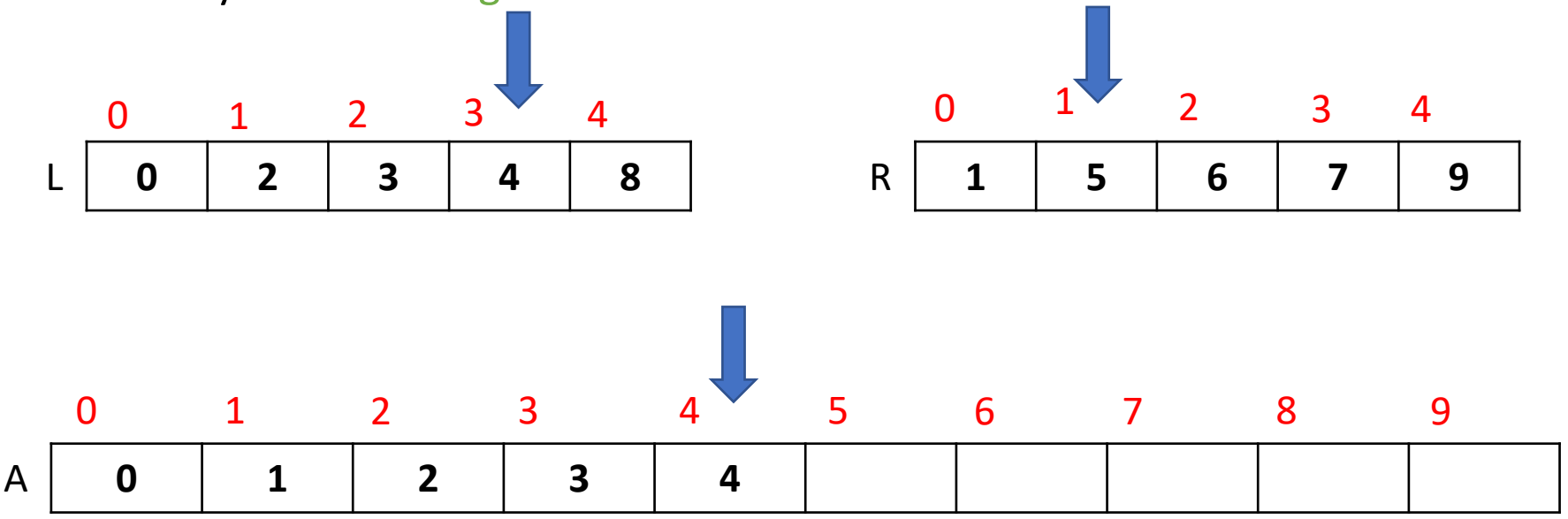
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

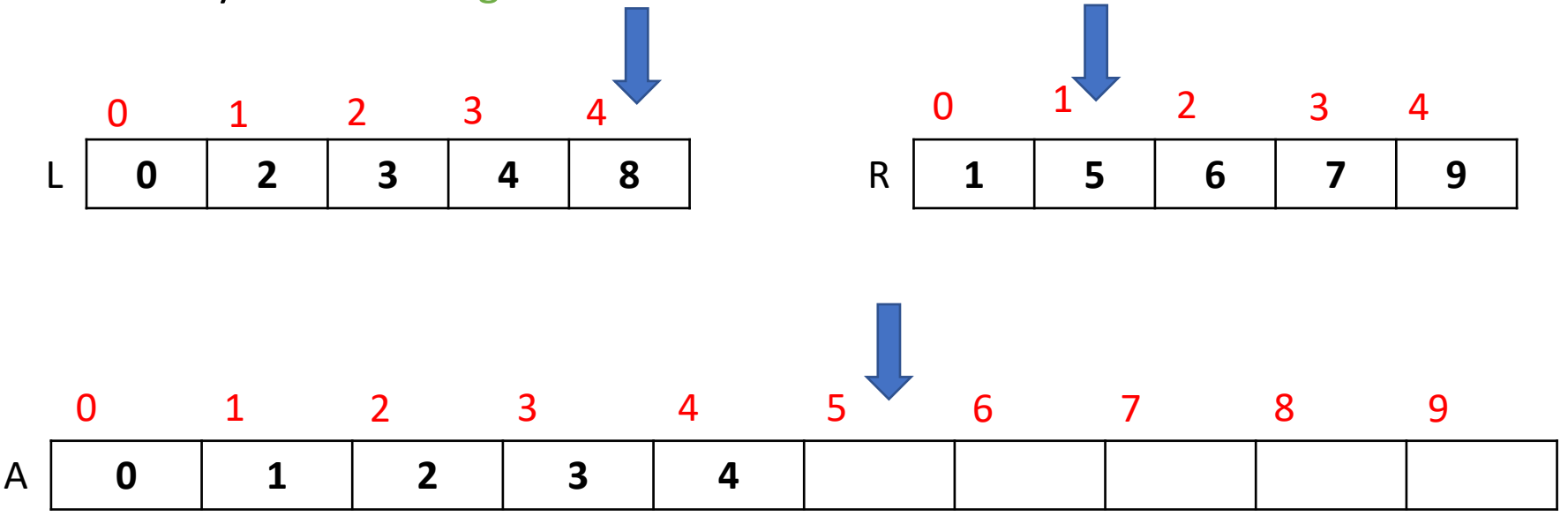
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

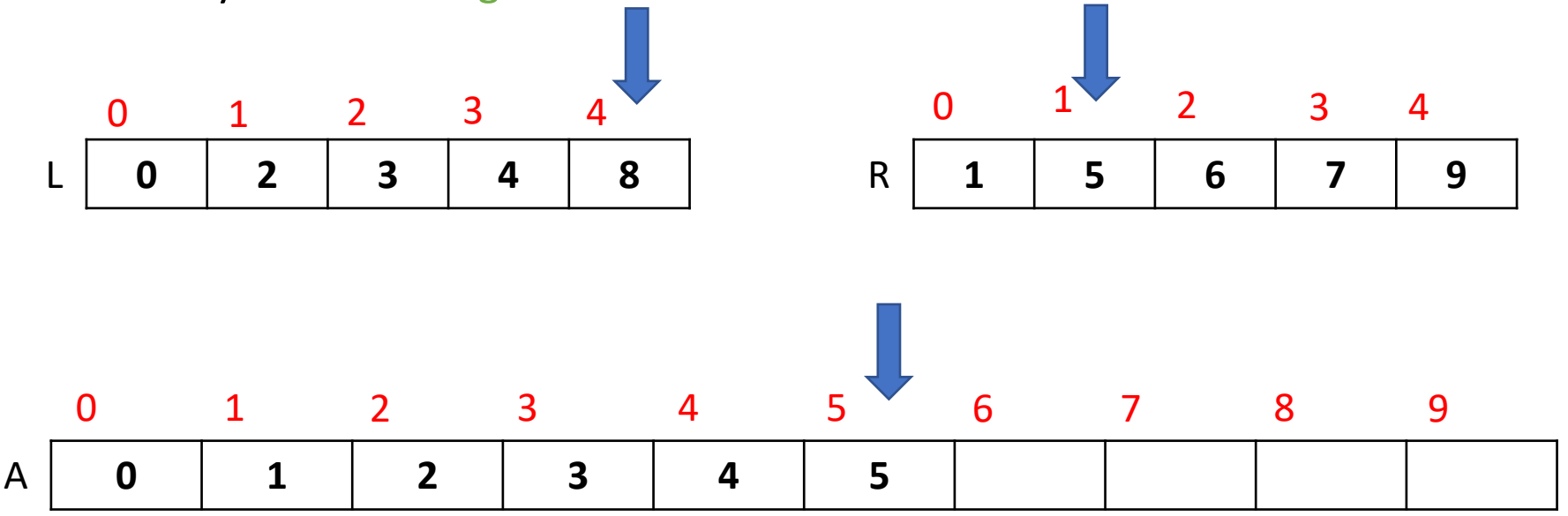
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

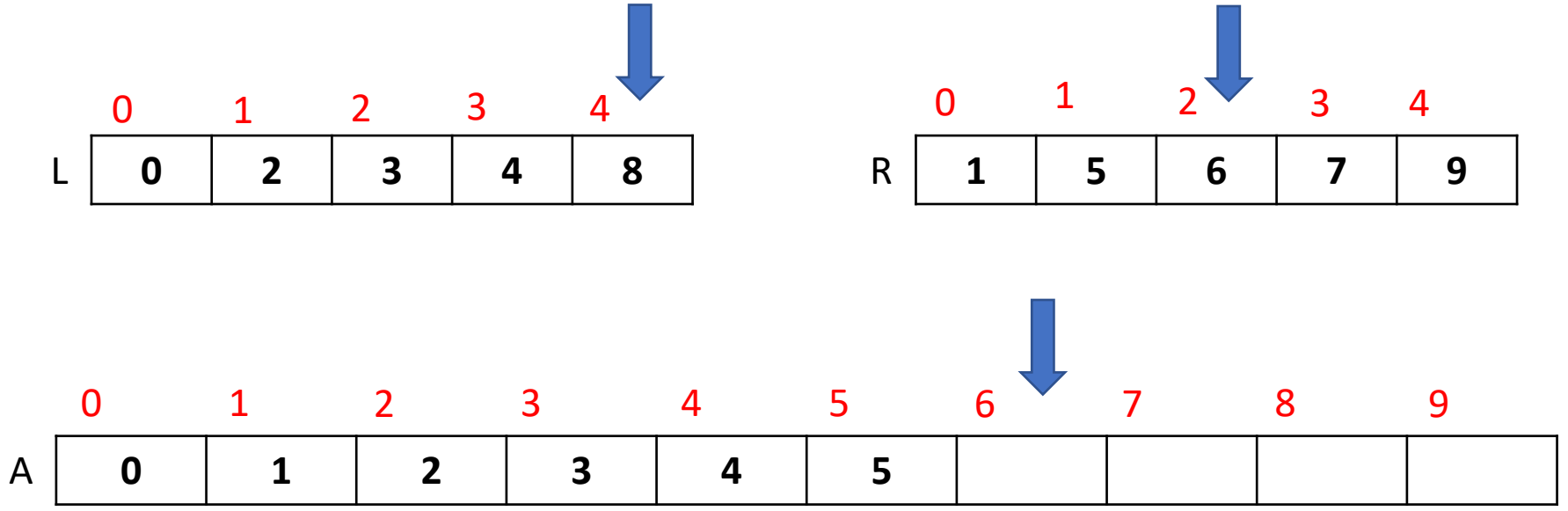
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

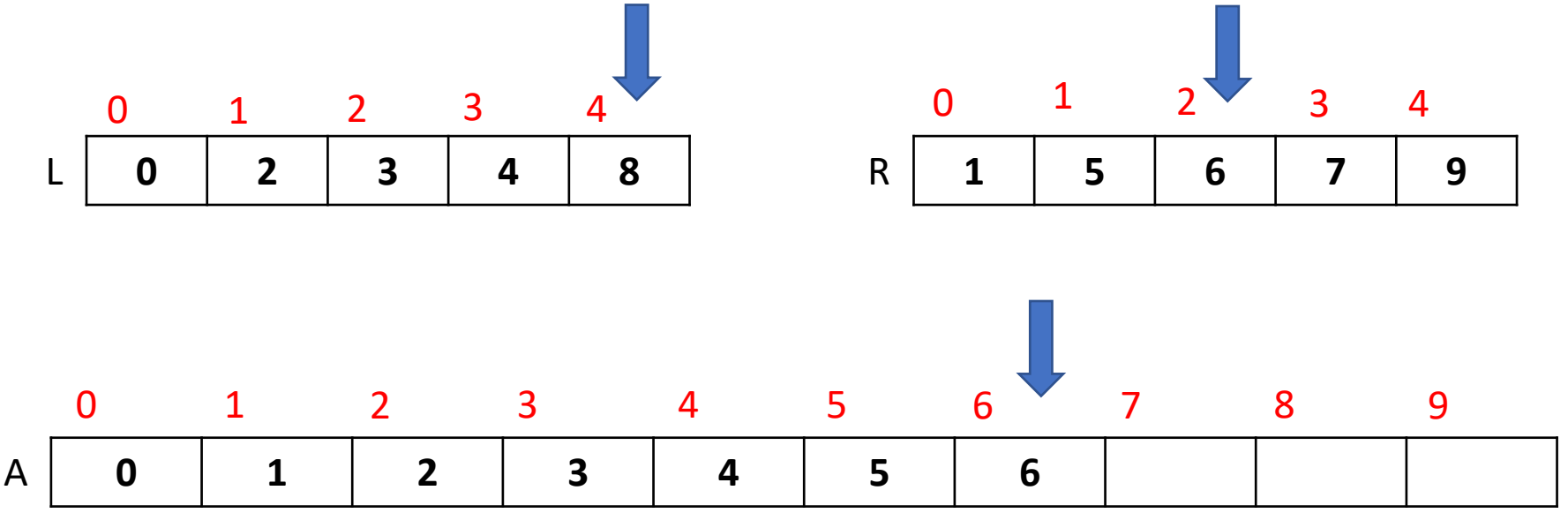
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

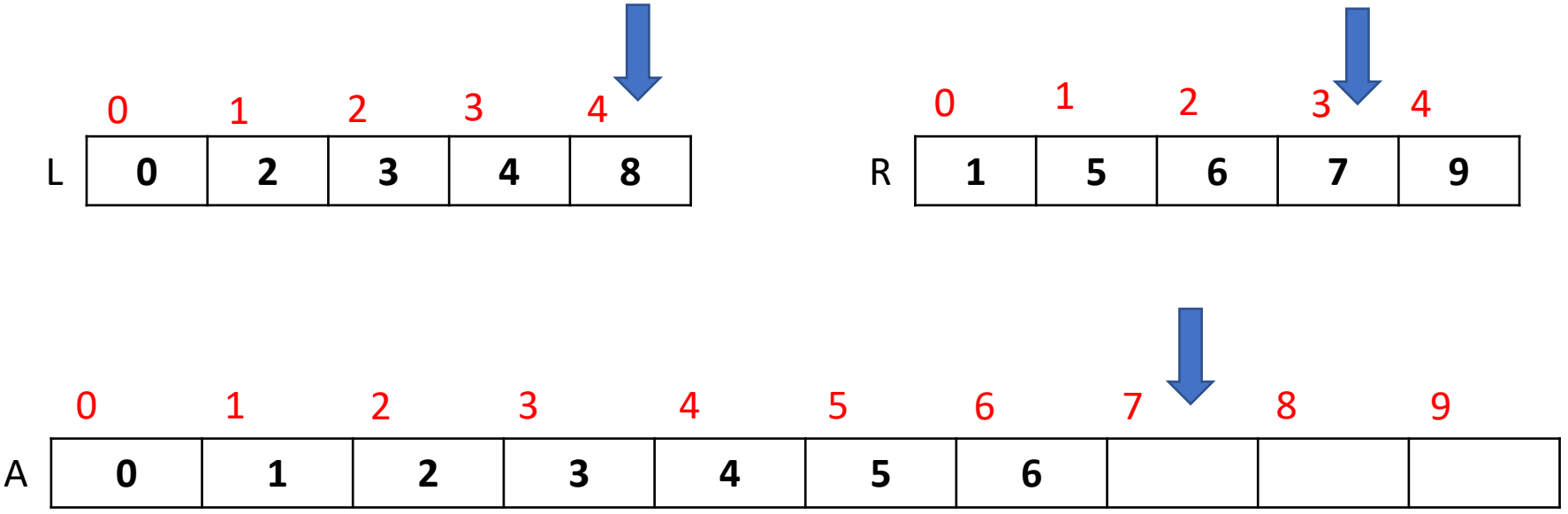
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

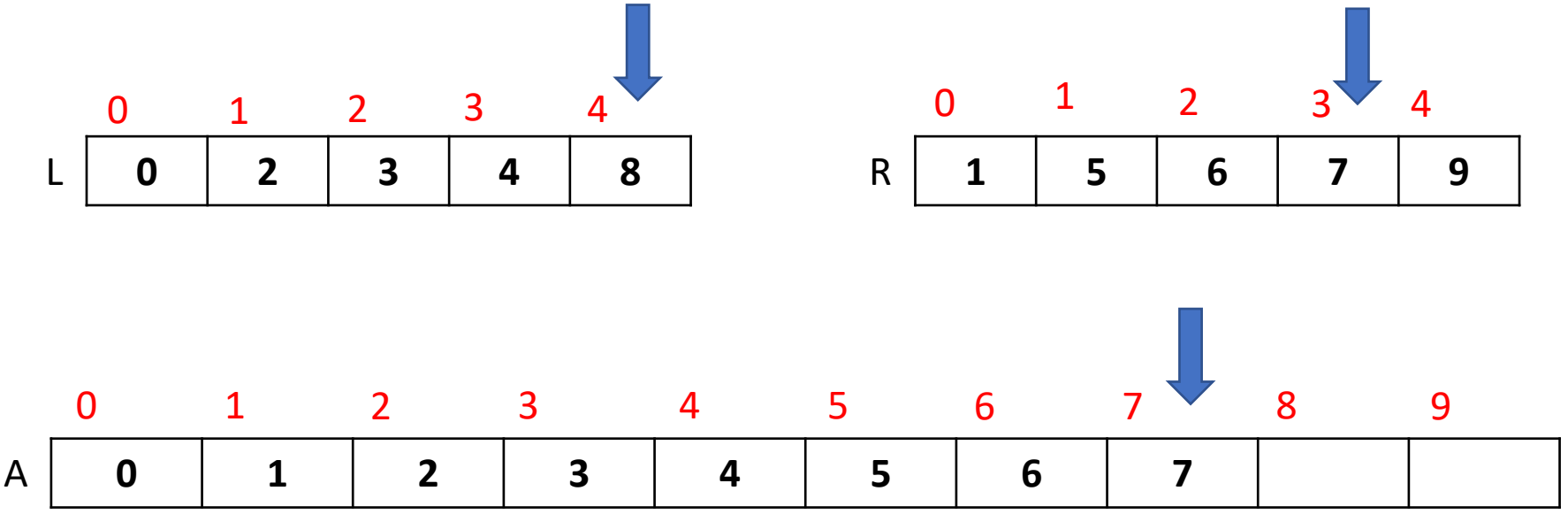
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

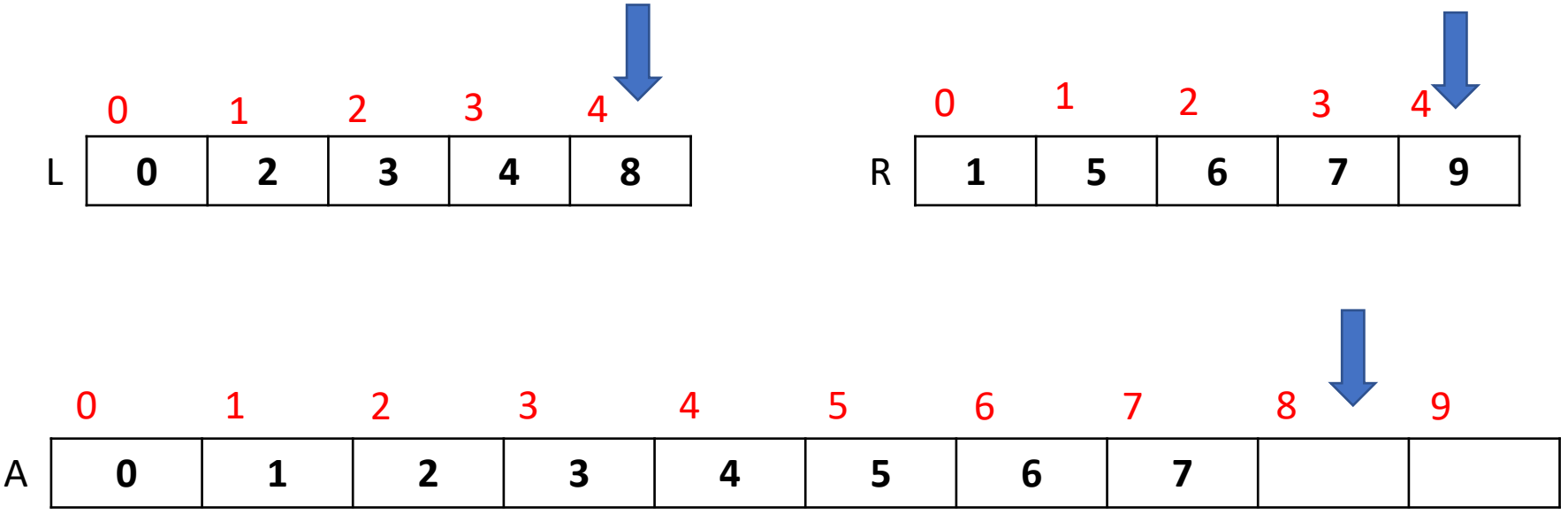
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

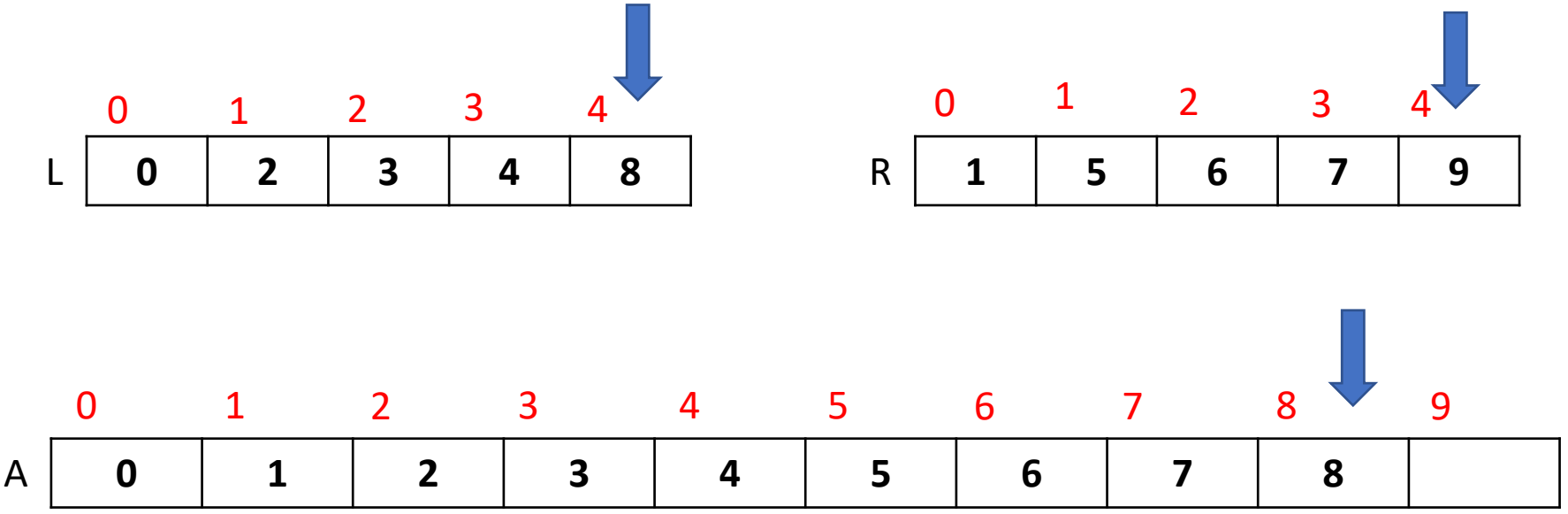
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

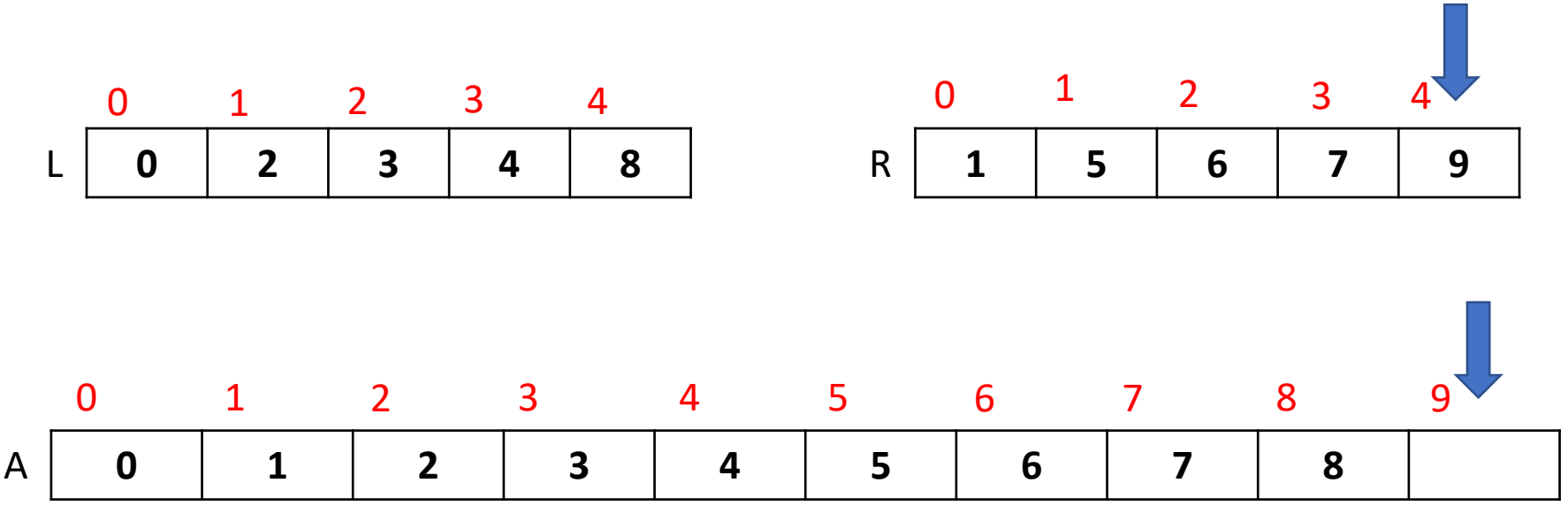
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

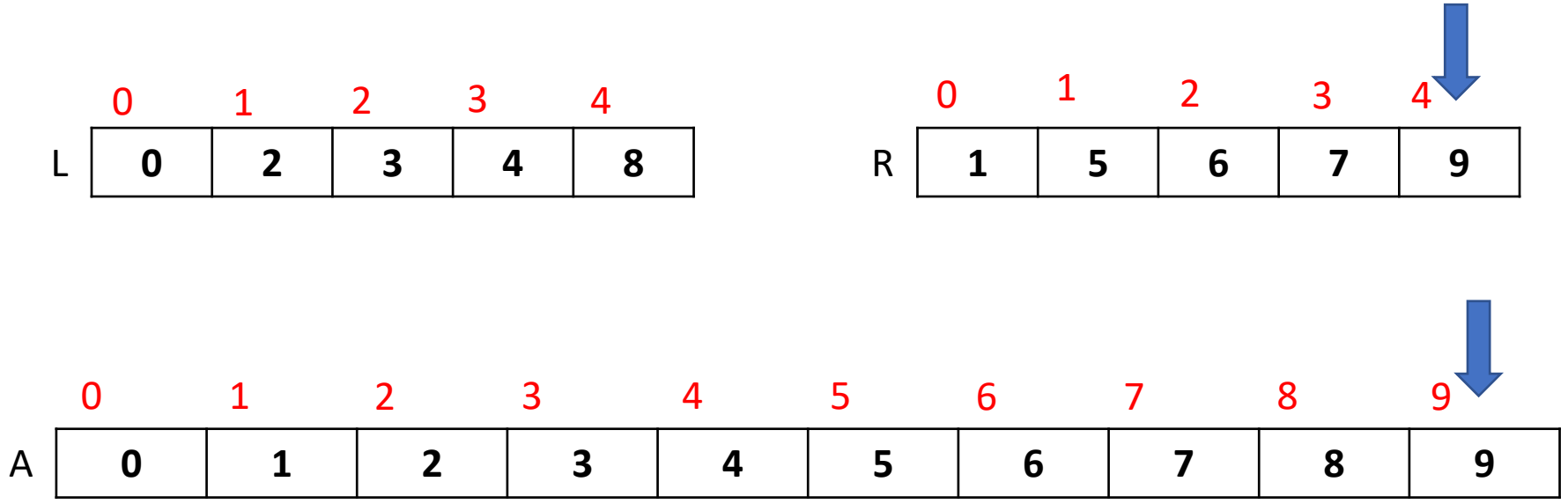
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

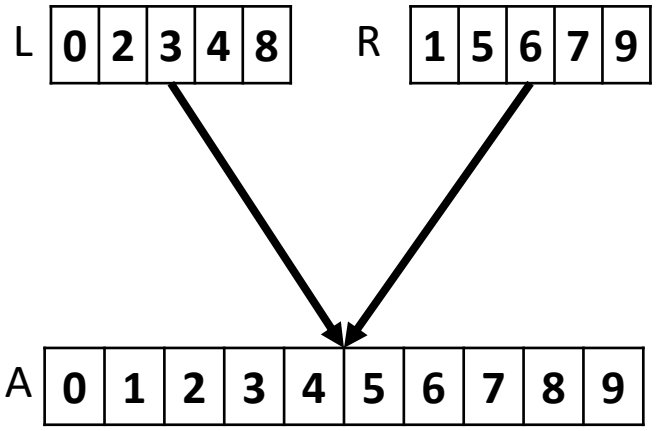
- Two sorted arrays can be merged in linear time



Merge Sort

Basic idea:

- Two sorted arrays can be merged in linear time



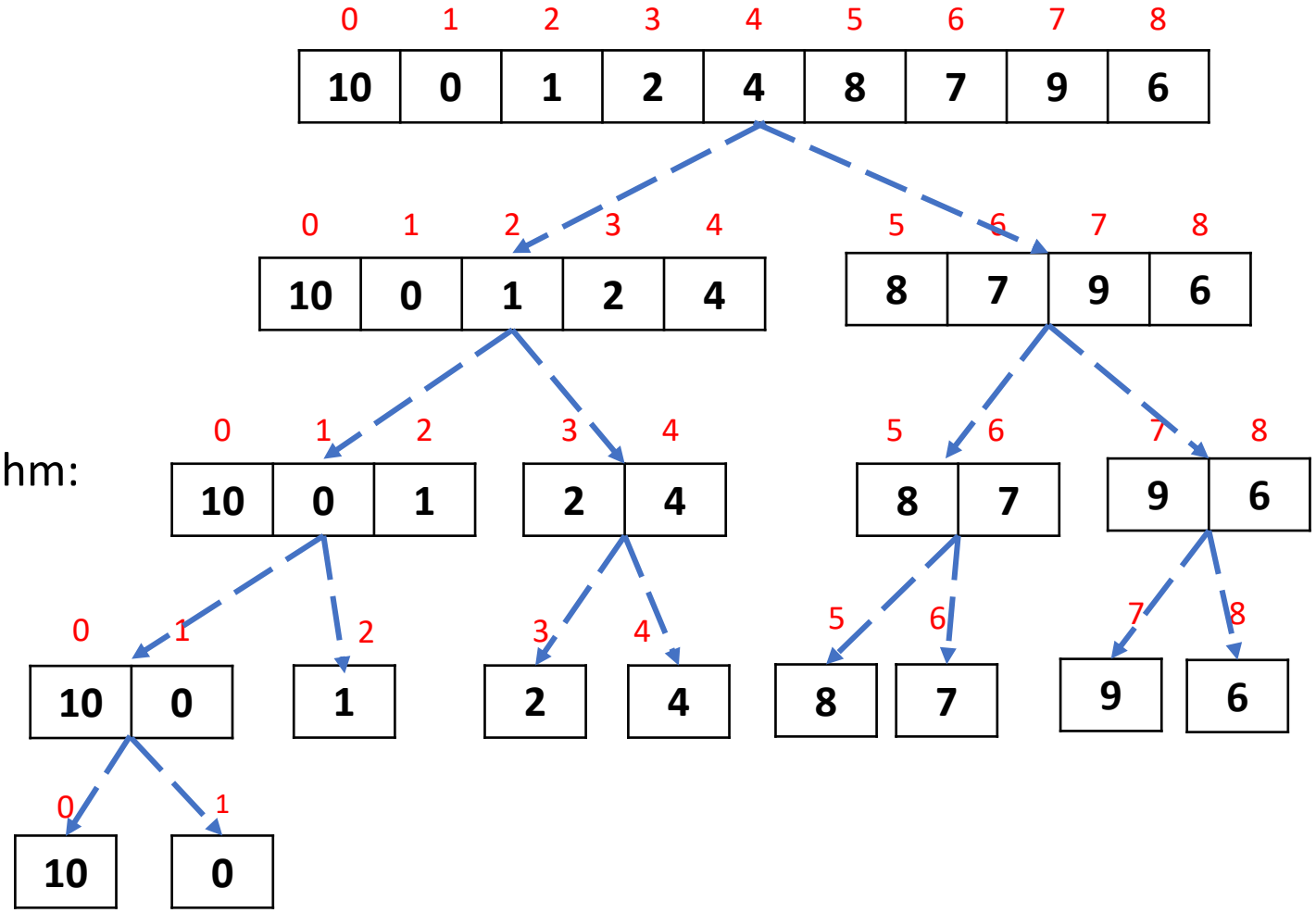

```
merge(A, L, R):
        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                A[k] = L[i]
                i += 1
            else:
                A[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            A[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            A[k] = R[j]
            j += 1
            k += 1
```

Merge Sort

Basic idea:

Merge Sort is a Divide and Conquer algorithm:

- It divides the input array into two halves
- calls itself for the two halves
- merges the two sorted halves

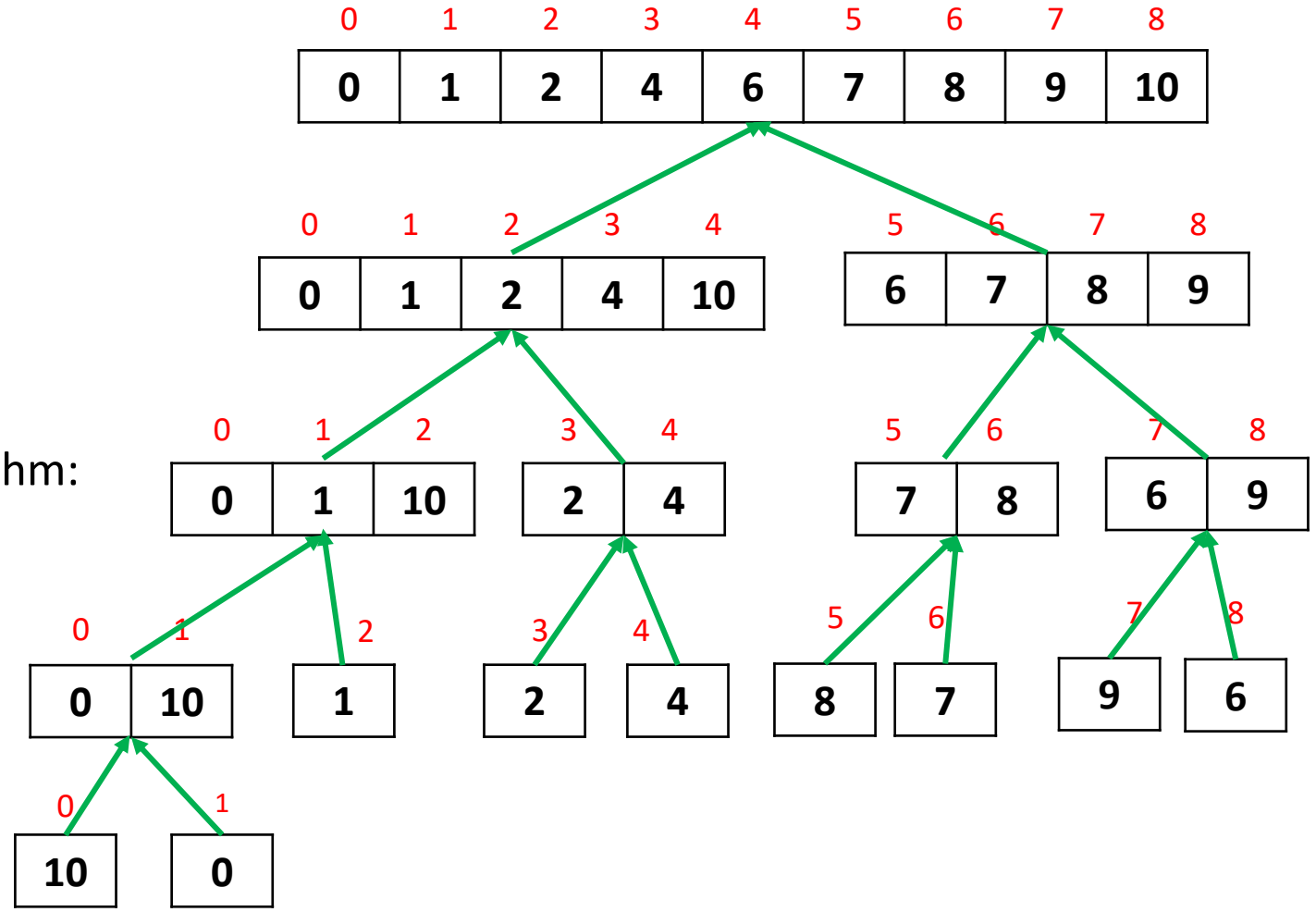


Merge Sort

Basic idea:

Merge Sort is a Divide and Conquer algorithm:

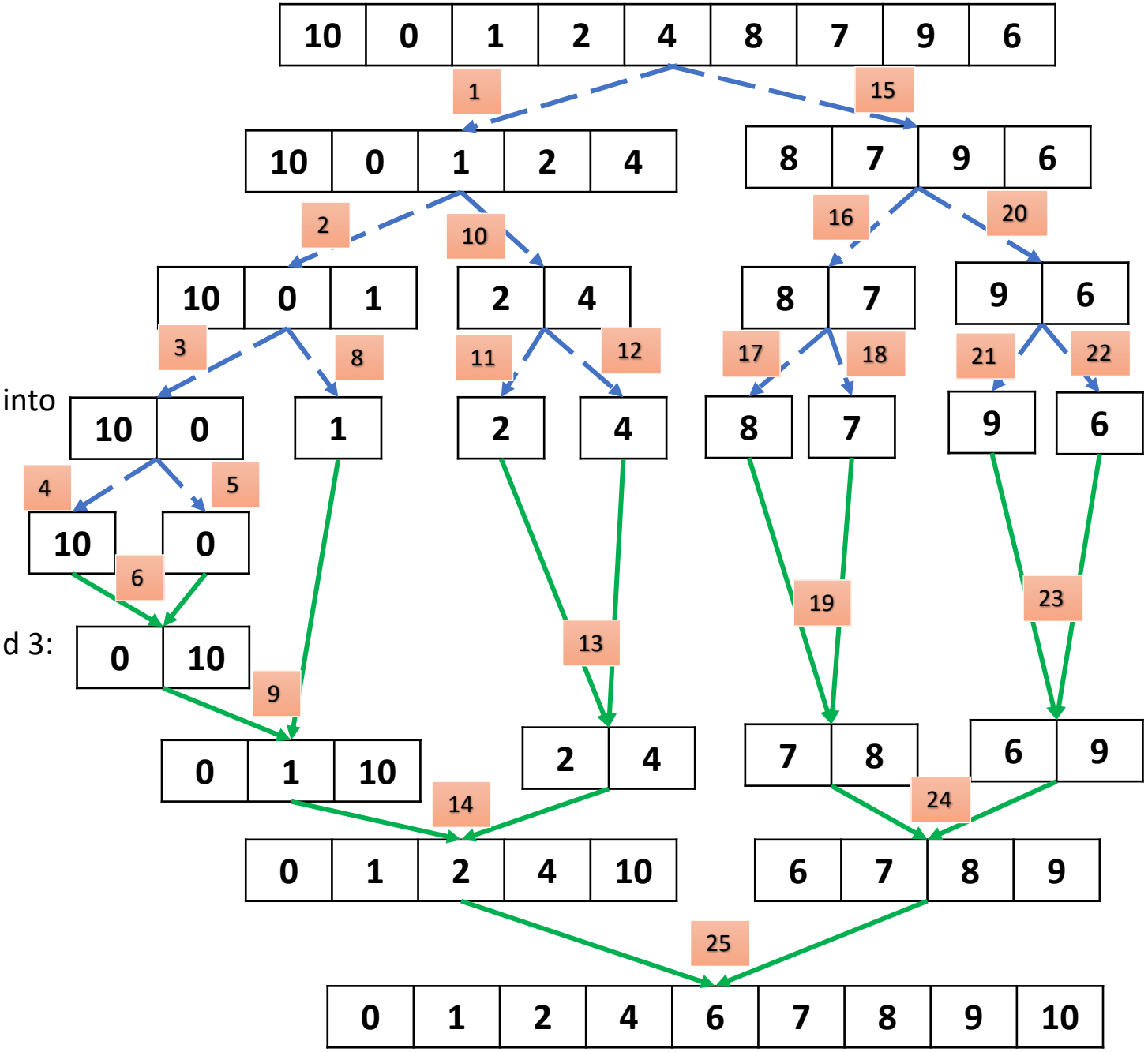
- It divides the input array into two halves
- calls itself for the two halves
- merges the two sorted halves



Merge Sort

$\boldsymbol{merge\_sort}(A, l, r)$
If $r > l$

1. Find the middle point to divide the array into two halves: $m = (l + r)/2$
2. Call mergeSort for first half: $\boldsymbol{merge\_sort}(A, l, m)$
3. Call mergeSort for second half: $\boldsymbol{merge\_sort}(A, m + 1, r)$
4. Merge the two halves sorted in step 2 and 3: $\boldsymbol{merge}(A, l, m, r)$



Calls order

```
def merge_sort(arr):
    if len(arr) > 1:

         # Finding the mid of the array
        mid = len(arr)//2

        # Dividing the array elements
        L = arr[:mid]

        # into 2 halves
        R = arr[mid:]

        # Sorting the first half
         merge_sort(L)

        # Sorting the second half
         merge_sort(R)
```

```
         i = j = k = 0

         while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

Merge Sort

$\boldsymbol{merge\_sort(A, l, r)}$ → $T(n)$

If $r > l$

1. Find the middle point to divide the array into two halves: $m = (l + r)/2$ → 1
2. Call mergeSort for first half: $\boldsymbol{merge\_sort}(A, l, m)$ → $T(n/2)$
3. Call mergeSort for second half: $\boldsymbol{merge\_sort}(A, m + 1, r)$ → $T(n/2)$
4. Merge the two halves sorted in step 2 and 3: $\boldsymbol{merge}(A, l, m, r)$ → n

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

Time Complexity: Merge Sort is a recursive algorithm and time complexity can be expressed as the following recurrence relation

$$T(n) = \begin{cases} 1 & , n = 1 \\ 2T\left(\frac{n}{2}\right) + n & , n > 1 \end{cases}$$

Merge Sort

$\boldsymbol{merge\_sort}(\boldsymbol{A}, \boldsymbol{l}, \boldsymbol{r})$
If $r > l$

1. Find the middle point to divide the array into two halves: $m = (l + r)/2$
2. Call mergeSort for first half: $\boldsymbol{merge\_sort}(A, l, m)$
3. Call mergeSort for second half: $\boldsymbol{merge\_sort}(A, m + 1, r)$
4. Merge the two halves sorted in step 2 and 3: $\boldsymbol{merge}(A, l, m, r)$

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

$$= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$$

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

$$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

$$\vdots \qquad \vdots$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$

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

Time complexity of Merge Sort in all 3 cases (worst, average and best) is

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn = n + nlogn = O(nlogn)$$

Auxiliary Space: $O(n)$

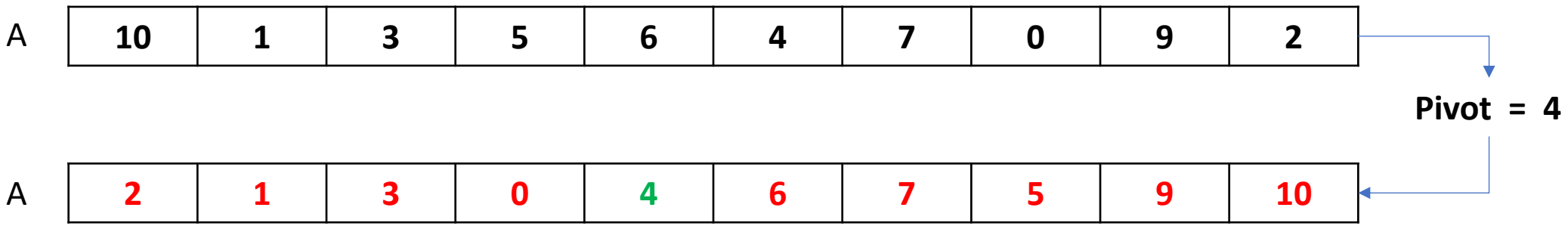
Quick Sort

Quick Sort

Basic idea:

- It picks an element in the array, which will be **pivot**

- Partitions the given array around the picked pivot.
  - a **partition step**, re-arranging the entries so that:
  - entries smaller than the pivot are to the left of the pivot.
  - entries larger than the pivot are to the right
- **Recursively apply quicksort** to the part of the array that is to the left of the pivot, and to the part on its right.

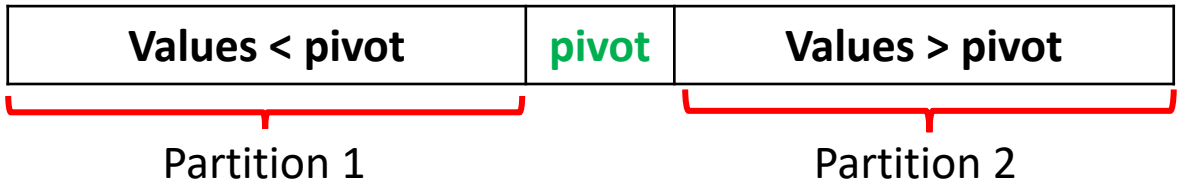


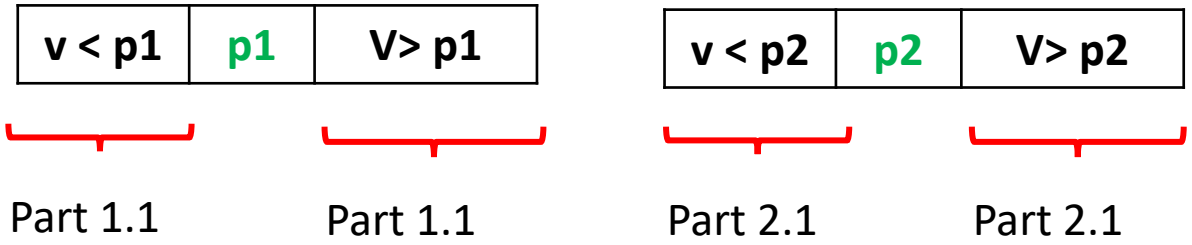
Quick Sort

QuickSort is a Divide and Conquer algorithm.

Choosing pivot:

- Always pick first element.
- Always pick last element.
- Pick a random element.
- Pick median.






```
/* l --> Starting index, h --> Ending index */

quick_sort(arr[], l, h)
{
        if (low < high)
        {
        pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
        }
}
```

Quick Sort

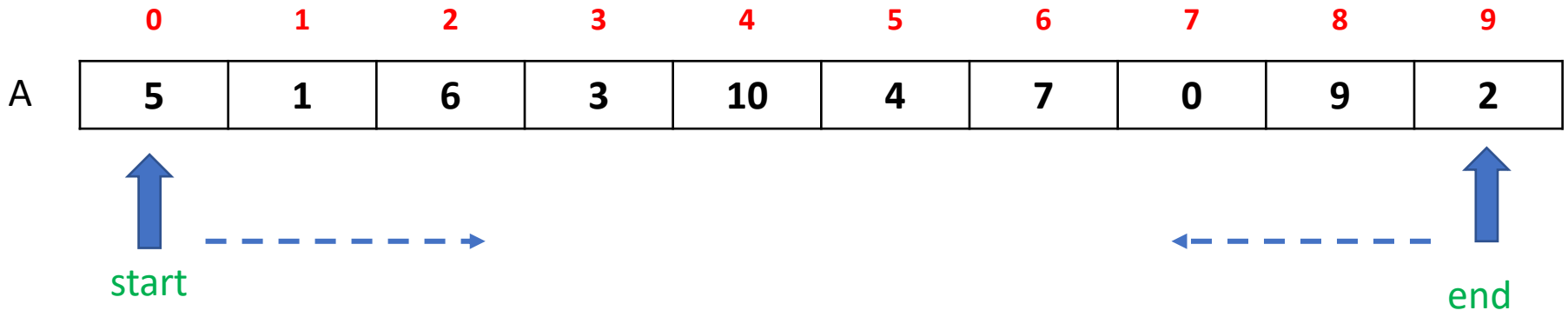
- The key process in quickSort is **partition()**.
- Given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

A	5	1	3	6	10	4	7	0	9	2

pivot = first element

Quick Sort

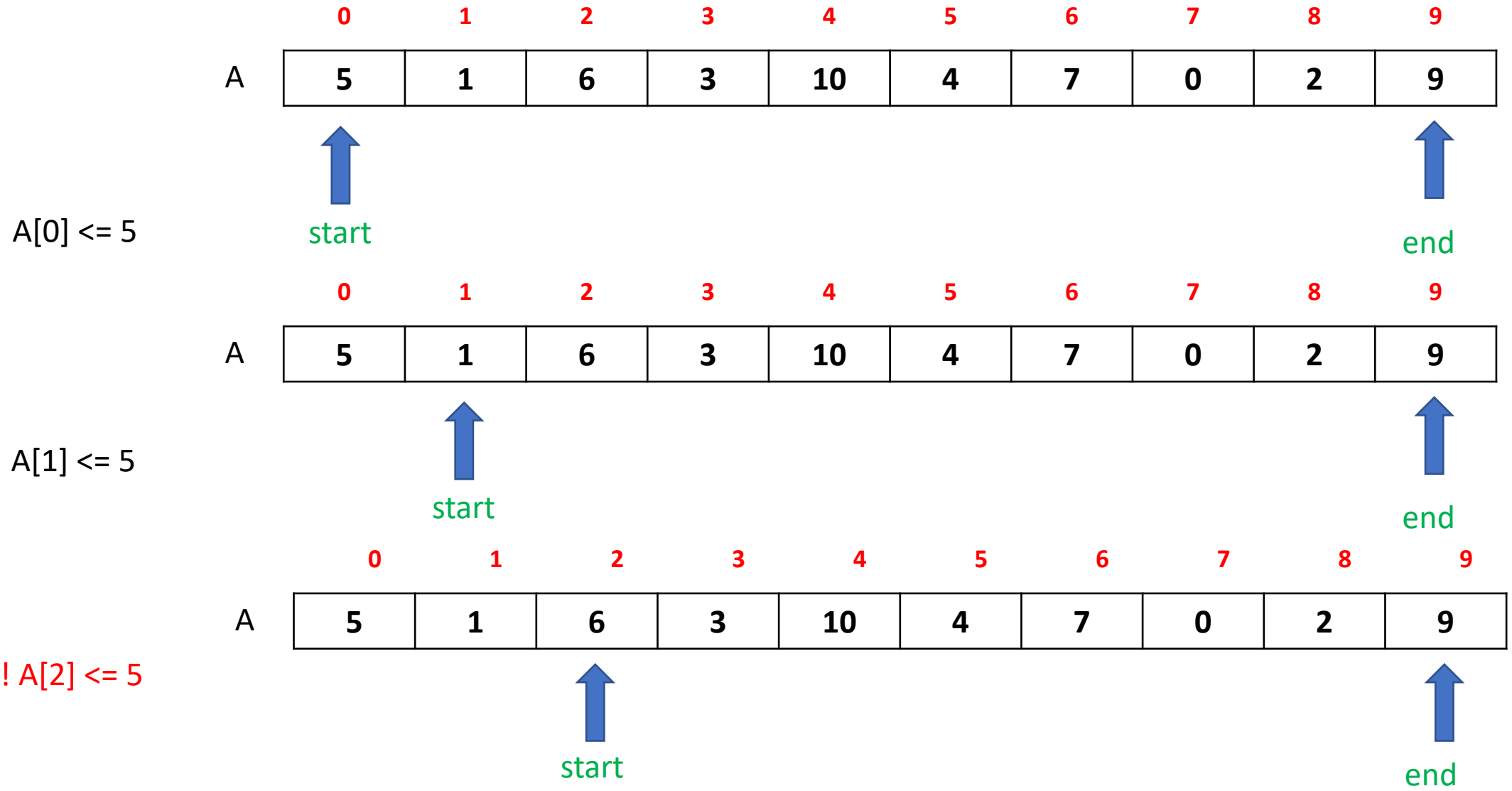
- The key process in quickSort is **partition()**.
- Given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.



pivot = 5

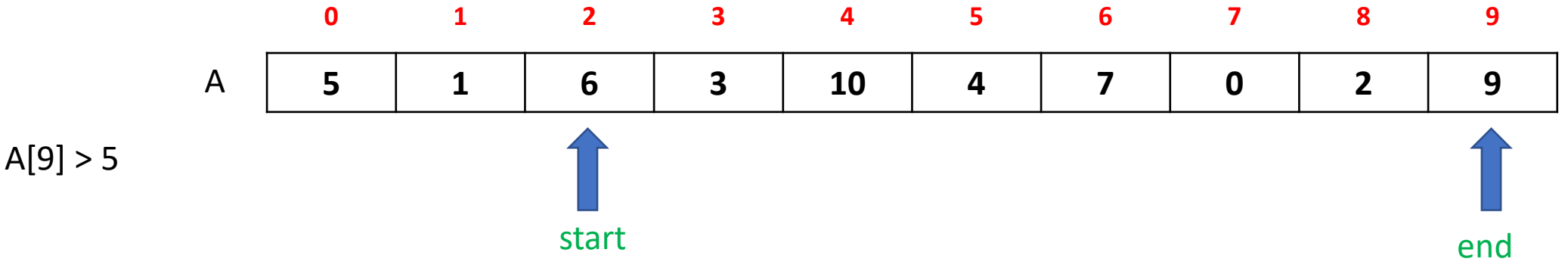
Quick Sort

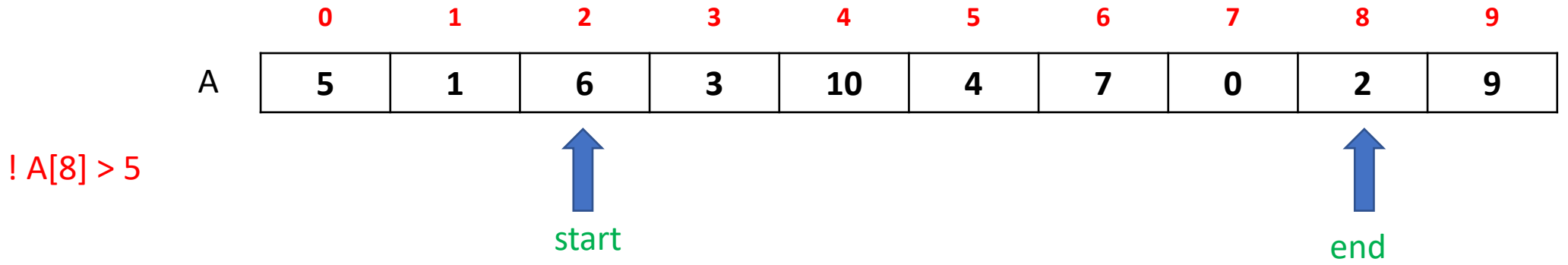
pivot = A[start] = 5

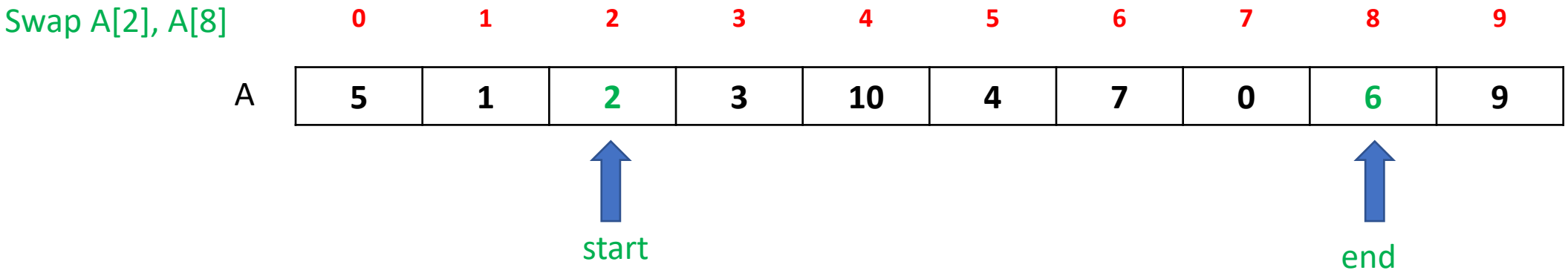


Quick Sort

pivot = A[start] = 5

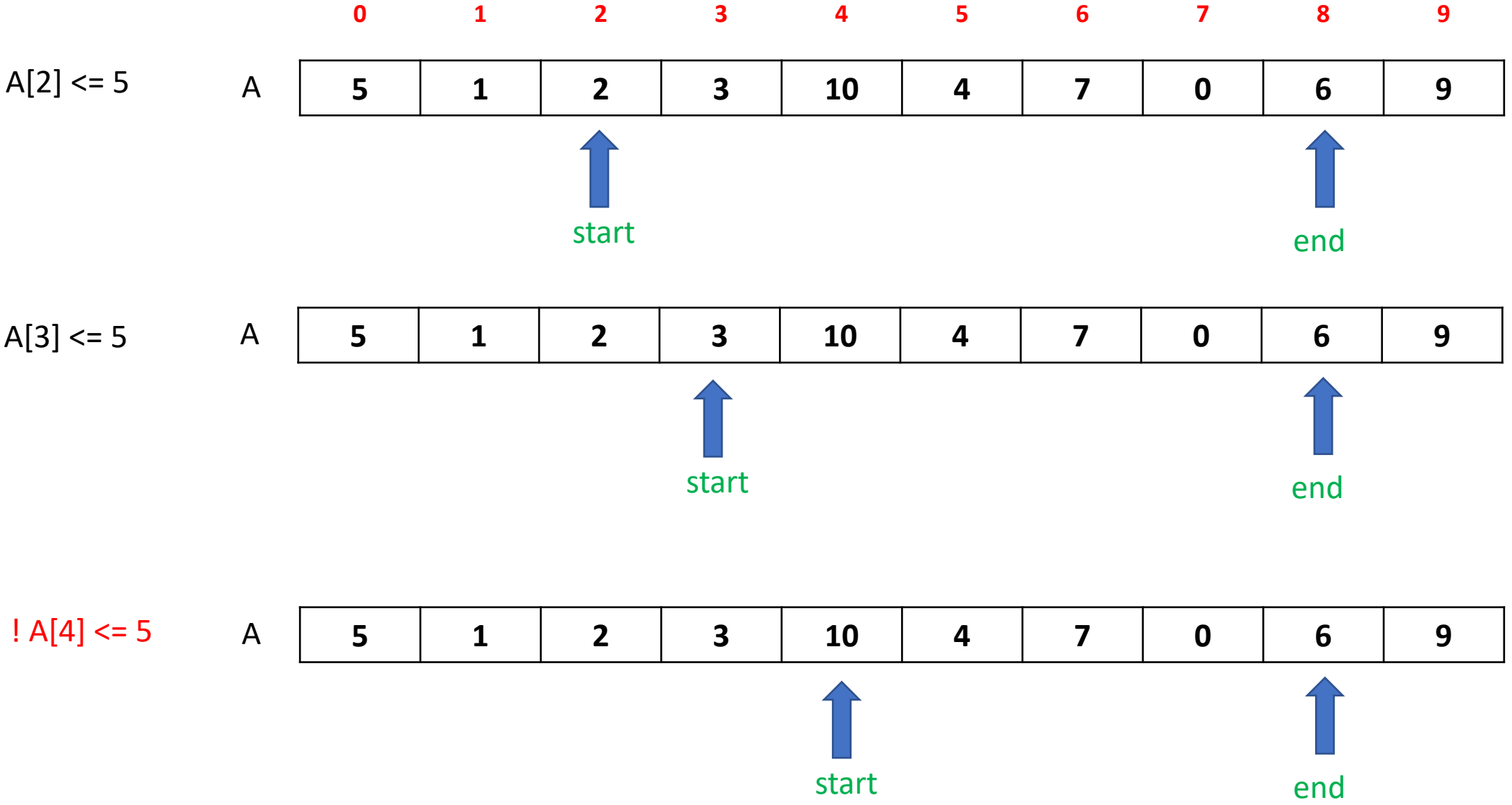






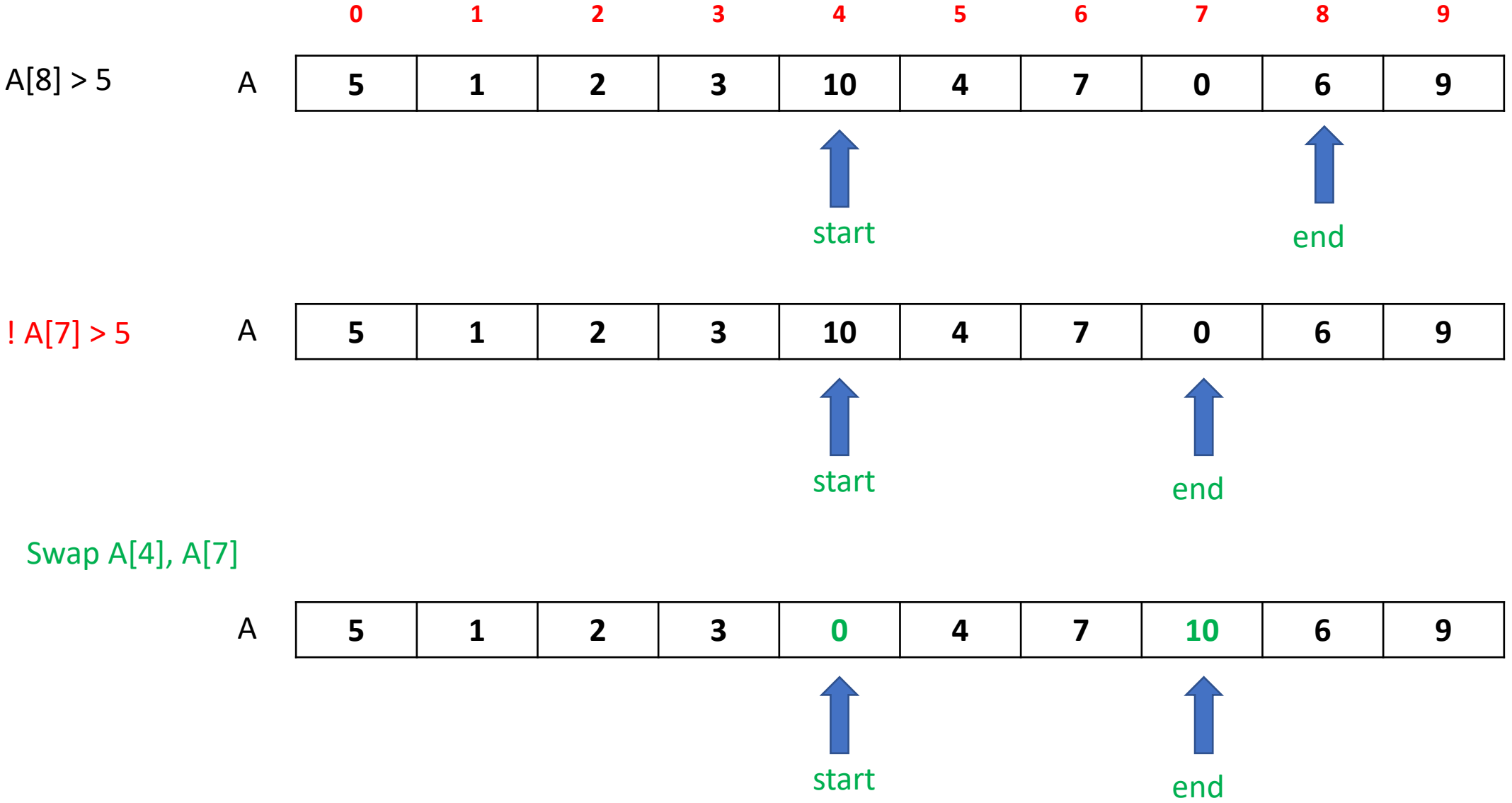
Quick Sort

pivot = A[start] = 5

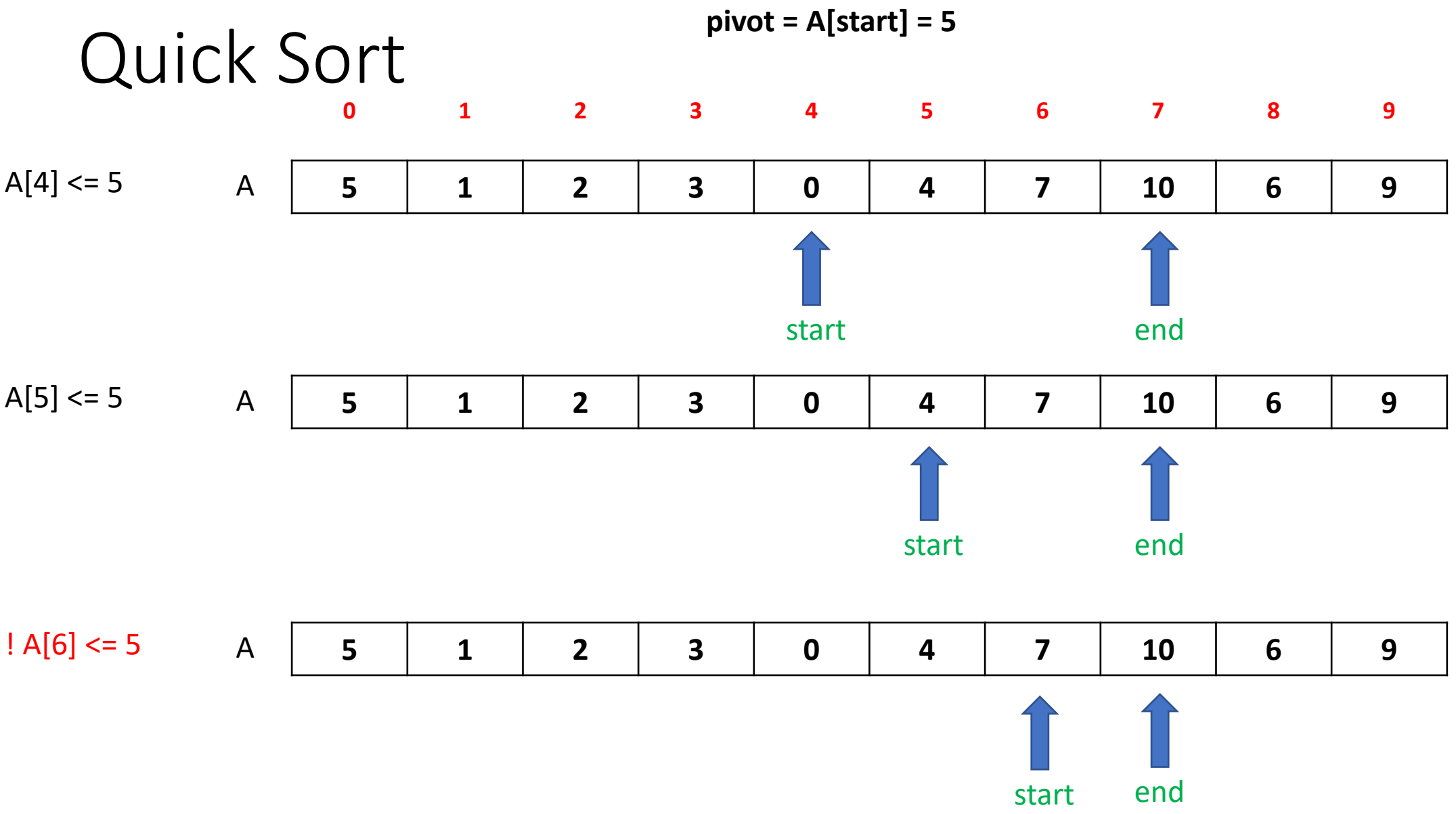


Quick Sort

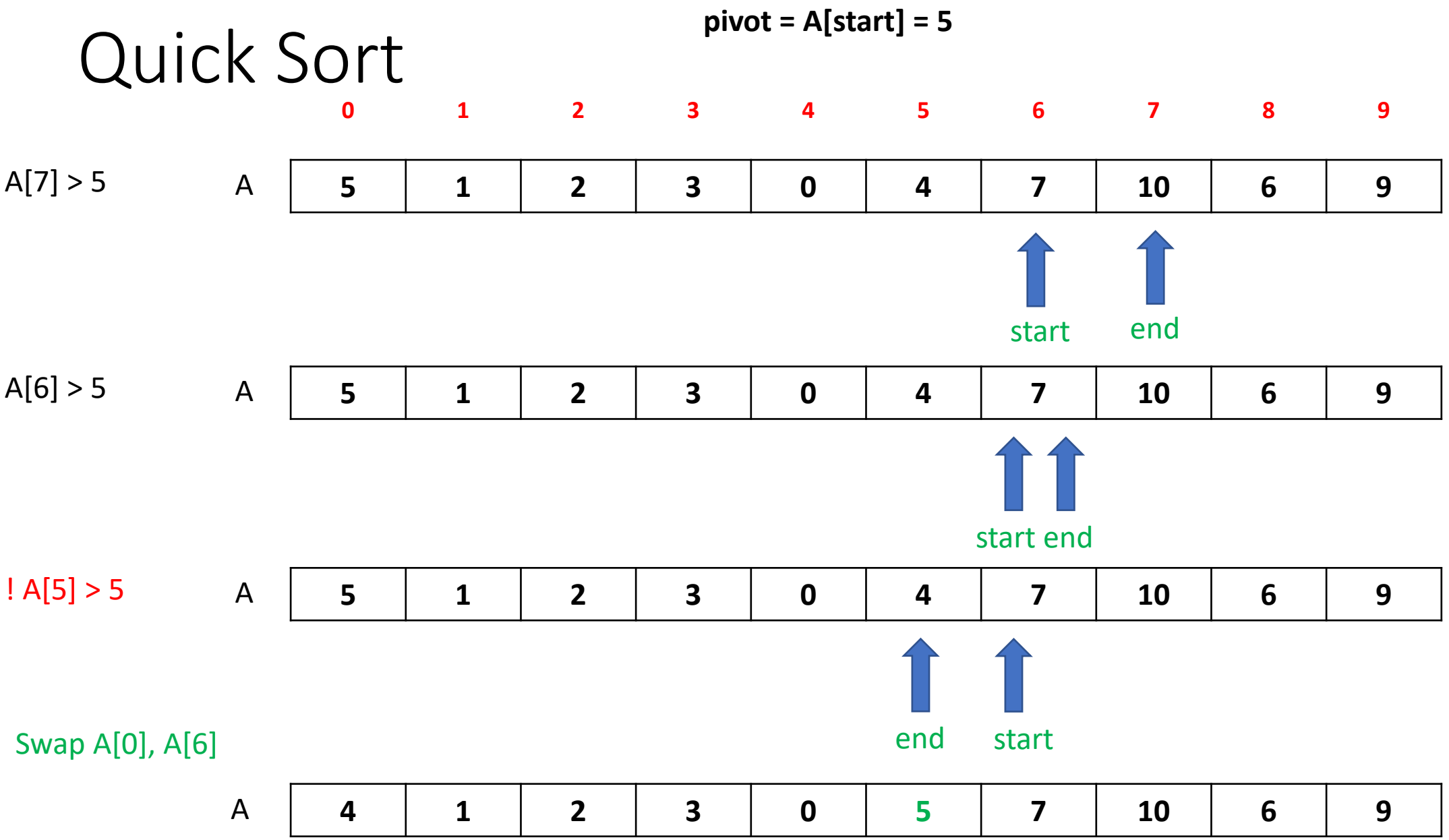
pivot = A[start] = 5



Quick Sort


pivot = A[start] = 5
0 1 2 3 4 5 6 7 8 9
A[4] <= 5
A
5 1 2 3 0 4 7 10 6 9
start
end
A[5] <= 5
A
5 1 2 3 0 4 7 10 6 9
start
end
! A[6] <= 5
A
5 1 2 3 0 4 7 10 6 9
start
end

Quick Sort


pivot = A[start] = 5
0 1 2 3 4 5 6 7 8 9
A[7] > 5
A
5 1 2 3 0 4 7 10 6 9
start
end
A[6] > 5
A
5 1 2 3 0 4 7 10 6 9
start end
! A[5] > 5
A
5 1 2 3 0 4 7 10 6 9
end
start
Swap A[0], A[6]
A
4 1 2 3 0 5 7 10 6 9

```
def partition(start, end, array):

    pivot_index = start
    pivot = array[pivot_index]

    while start < end:

       while start < len(array) and array[start] <= pivot:
            start += 1

       while array[end] > pivot:
            end -= 1

       if(start < end):
            array[start], array[end] = array[end], array[start]

    array[end], array[pivot_index] = array[pivot_index], array[end]

    return end
```

```
def quick_sort(start, end, array):

    if (start < end):

        # p is partitioning index, array[p] is at right place
        p = partition(start, end, array)

        quick_sort(start, p - 1, array)
        quick_sort(p + 1, end, array)
```

Quick Sort

```
quick_sort(arr[], l, h)                                  → T(n)
{
        if (low < high)                                  → 1
        {
        pi = partition(arr, low, high);                  → n
        quick_sort(arr, low, pi - 1);                    → T(k)
        quick_sort(arr, pi + 1, high);                   → T(n − k − 1)
        }
}
```

k is the number of elements which are smaller than pivot

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

The time taken by QuickSort depends upon the input array and partition strategy.

Quick Sort

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot.

Example: Always picking first or last element in a sorted array.

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

$$T(n) = T(0) + T(n-1) + n$$

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

$$O(n^2)$$

Quick Sort

Best Case: The best case occurs when the partition process always picks the median element as pivot.

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

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

The solution of above recurrence is $O(n\,Logn)$

In average quicksort runs in $O(nlogn)$

Comparison

Name	Best case	Average	Worst case	Memory	Stable
Selection sort	n^2	n^2	n^2	1	No
Insertion sort	n	n^2	n^2	1	Yes
Bubble sort	n	n^2	n^2	1	Yes
Merge sort	$nlogn$	$nlogn$	$nlogn$	n	Yes
Quick sort	$nlogn$	$nlogn$	n^2	$logn$	No