DS286 | 2016-10-05

L17: Sorting Algorithms

Yogesh Simmhan
simmhan@cds.iisc.ac.in

Slides courtesy Venkatesh Babu, CDS

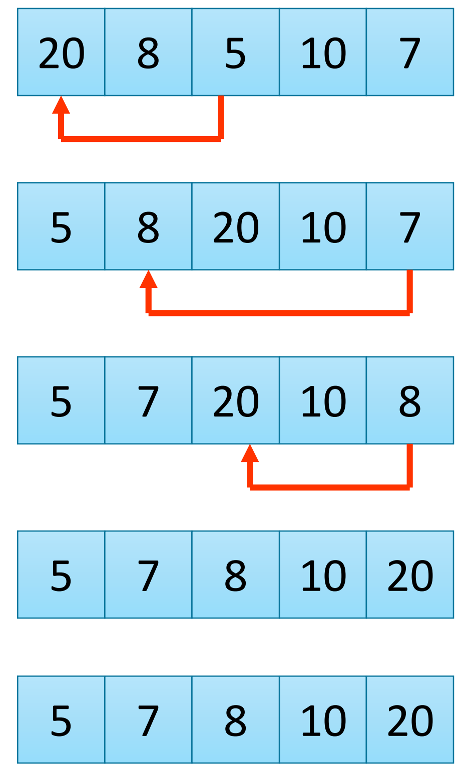
Sorting

- Ordering items based on a particular relationship between items
  - *Ascending* and *descending* order of sorting
- **Keys** are used to sort **`<Key,Value>`** pairs
  - Multiple sort keys are also possible, e.g. first name+last name
- Natural order for different primitive data types
  - Numbers by their larger or smaller values
  - Lexical order of English words
- Custom order is possible as well
  - "Comparator" function that takes (K1, K2) and returns if K1>K2, K1==K2 or K1<K2
- We will deal with single keys of integer type sorted by natural order

Selection Sorting

Steps:

- Select the minimum value in the list
- Swap it with the value in the first position
- Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)
- **Invariant**: After i^{th} step, first i elements have the smallest i values in sorted



Example

64	25	12	22	11
64	25	12	22	11
11	25	12	22	64
11	12	25	22	64
11	12	22	25	64
11	12	22	25	64
11	12	22	25	64

Complexity

64	25	12	22	11	(n-1)
11	25	12	22	64	(n-2)
11	12	25	22	64	(n-3)
11	12	22	25	64	(n-4)
11	12	22	25	64	(n-5)

$(n - 1) + (n - 2) + \ldots + 2 + 1 = n(n - 1) / 2$

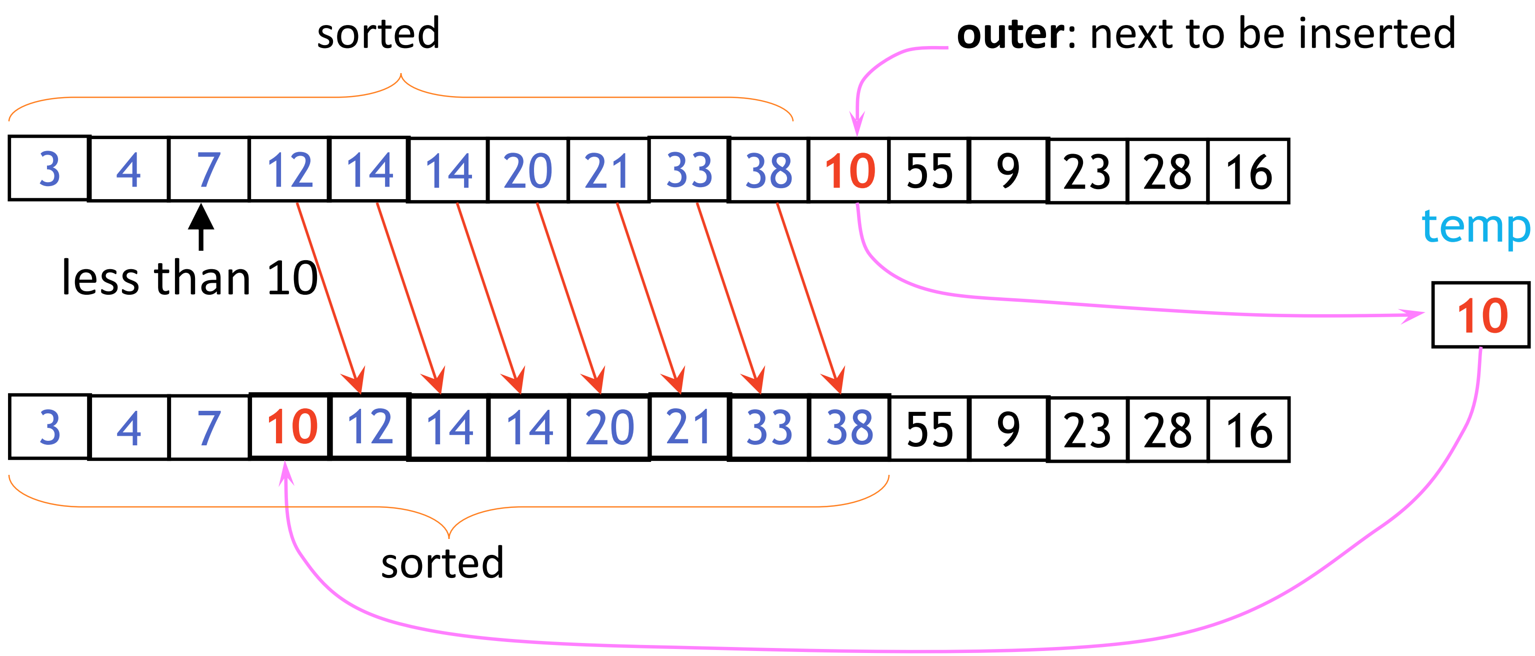
$\in \Theta(n^2)$

Worst case performance	$O(n^2)$
Best case performance	$O(n^2)$
Average case performance	$O(n^2)$

Insertion sort

- The outer loop of insertion sort is:
  `for (outer = 1; outer < a.length; outer++) {...}`
- The invariant is that all the elements to the left of outer are *sorted with respect to one another*
  - For all i < outer, j < outer, if i < j then a[i] <= a[j]
  - This does *not* mean they are all in their final correct place; the remaining array elements may need to be inserted
  - When we increase outer, a[outer-1] becomes to its left; we must keep the invariant true by inserting a[outer-1] into its proper place
  - This means:
    - Finding the element's proper place
    - Making room for the inserted element (by shifting over other elements)
    - Inserting the element

One step of insertion sort



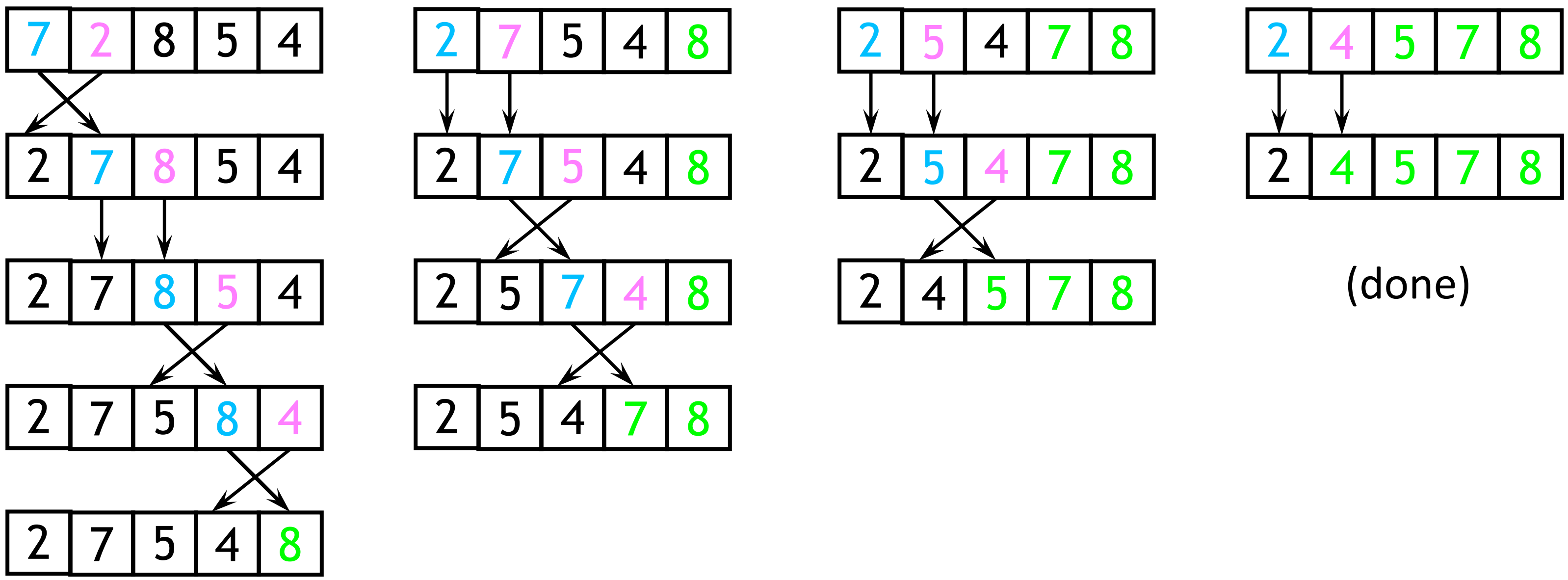
Analysis of insertion sort

- We run once through the outer loop, inserting each of *n* elements; this is a factor of n
- On average, there are n/2 elements already sorted
  - The inner loop looks at (and moves) half of these
  - This gives a second factor of n/4
- Hence, the time required for an insertion sort of an array of n elements is proportional to $n^2/4$
- Discarding constants, we find that insertion sort is $O(n^2)$
- *Can we reduce cost of element shift?*

Bubble sort

- Compare each element (except the last one) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the largest element at the very end
  - The last element is now in the correct and final place
- Compare each element (except the last *two*) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the second largest element next to last
  - The last two elements are now in their correct and final places
- Compare each element (except the last *three*) with its neighbor to the right
  - Continue as above until you have no unsorted elements on the left

Example of bubble sort



Code for bubble sort

```
public static void bubbleSort(int[] a) {
   int outer, inner;
   for(outer=a.length-1; outer>0; outer--) {
     // count down
      for (inner = 0; inner < outer; inner++) {
         // bubble up
         if (a[inner] > a[inner + 1]) {
            // is out of order? ...then swap
            int temp = a[inner];
            a[inner] = a[inner + 1];
            a[inner + 1] = temp;
         }
      }
   }
}
```

Analysis of bubble sort

```
for (outer = a.length - 1; outer > 0; outer--) {
   for (inner = 0; inner < outer; inner++) {
      if (a[inner] > a[inner + 1]) {
         // code for swap omitted
      }
   }
}
```

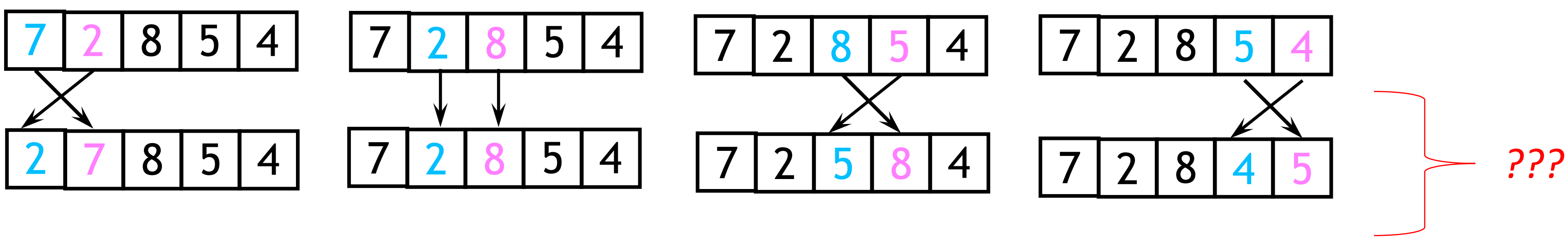
- Let n = a.length = size of the array
- The outer loop is executed n-1 times (call it n, that's close enough)
- Each time the outer loop is executed, the inner loop is executed
  - Inner loop executes n-1 times at first, linearly dropping to just once
  - On average, inner loop executes about n/2 times for each execution of the outer loop
  - In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time)
- Result is n * n/2 * k, that is, $O(n^2)$

Loop invariants

- You run a loop in order to change things
- Oddly enough, what is usually most important in understanding a loop is finding an invariant: that is, *a condition that doesn't change*
- In bubble sort, we *put the largest elements at the end*, and once we put them there, we don't move them again
  - The variable outer starts at the last index in the array and decreases to 0
  - Our invariant is: Every element to the right of outer is in the correct place
  - That is, for all j > outer, if i < j, then a[i] <= a[j]
  - When this is combined with outer == 0, we know that *all* elements of the array are in the correct place

Parallelizing Bubble Sort

- Complexity of bubble sort is $O(n^2)$
- Can we parallelize?
  - If we have 'p' processors, can we sort in $O(n^2/p)$?
- Problem: Swaps can interfere with each other



- Solution
  - Do odd and its predecessor, **or** even and its successor in 1 phase
  - All odds can be tested in parallel in odd phase, same with even
  - Alternate between these two phases

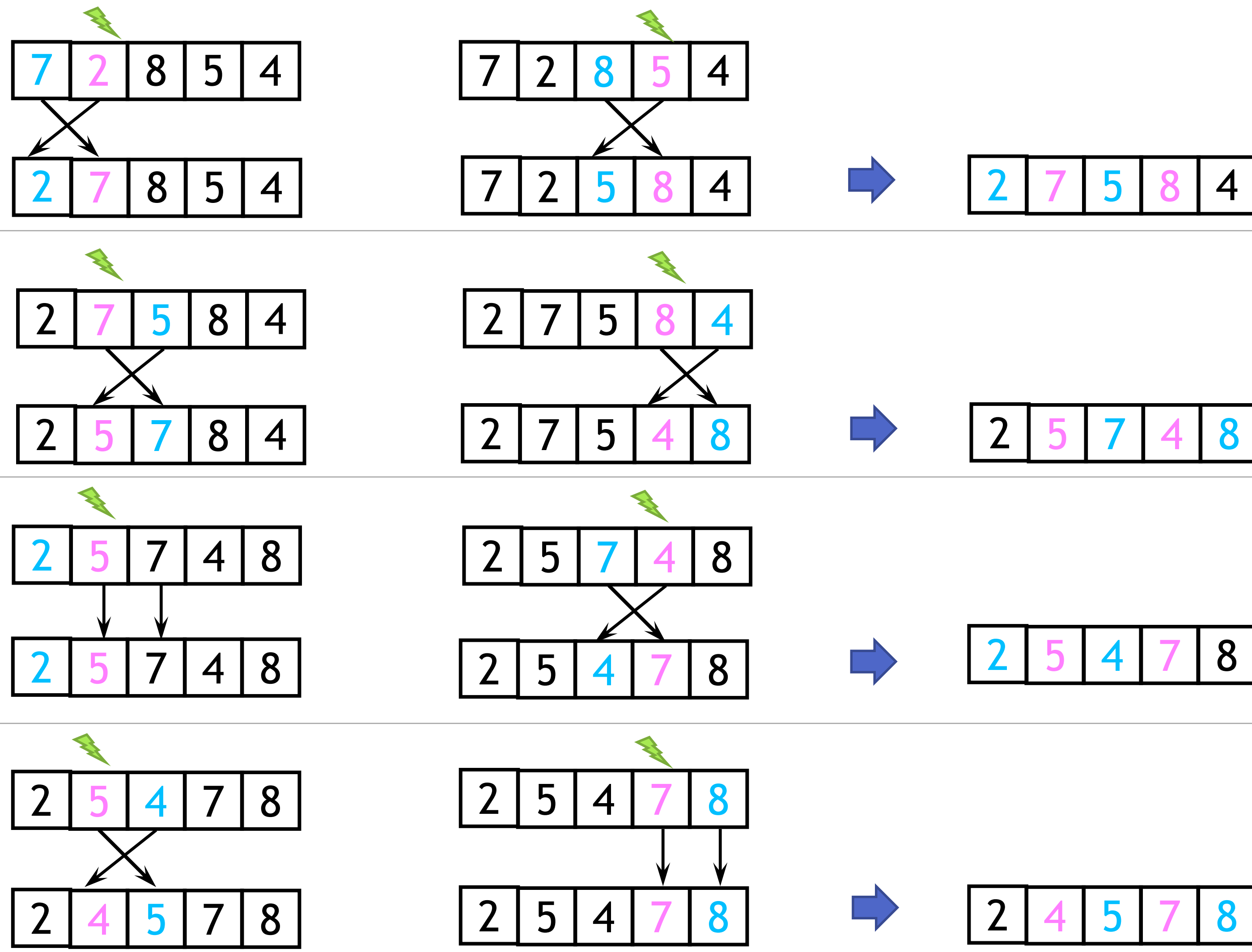
Parallelizing Bubble Sort

Odd iter,
test with pred

Even iter,
test with succ

Odd iter,
test with pred

Even iter,
test with succ



Parallelizing Bubble Sort

```
public static void bubbleSort(int[] a) {
  int n = a.length;
  for(int phase=0; phase<n; phase++) {
    if(phase%2 == 0) { // Even phase
      for(int i=1; i<n; i+=2) // Test even w/ prev
        if (a[i-1] > a[i]) {
          int t=a[i]; a[i]=a[i-1]; a[i-1]=t;
        }
    } else {  // Odd phase
      for(int i=1; i<n-1; i+=2) // Test odd w/ next
        if (a[i] > a[i+1]) {
          int t=a[i]; a[i]=a[i-1]; a[i-1]=t;
        }
     }
   }
}
```

Do in Parallel

Do in Parallel

Divide and Conquer

- Divide and Conquer
- Merge Sort
- Quick Sort

Divide and Conquer

1. Base Case, solve the problem directly if it is small enough

2. Divide the problem into two or more similar and smaller subproblems

3. Recursively solve the subproblems

4. Combine solutions to the subproblems

Divide and Conquer - Sort

Problem:

- Input: A[left..right] – **unsorted** array of integers
- Output: A[left..right] – **sorted** in non-decreasing order

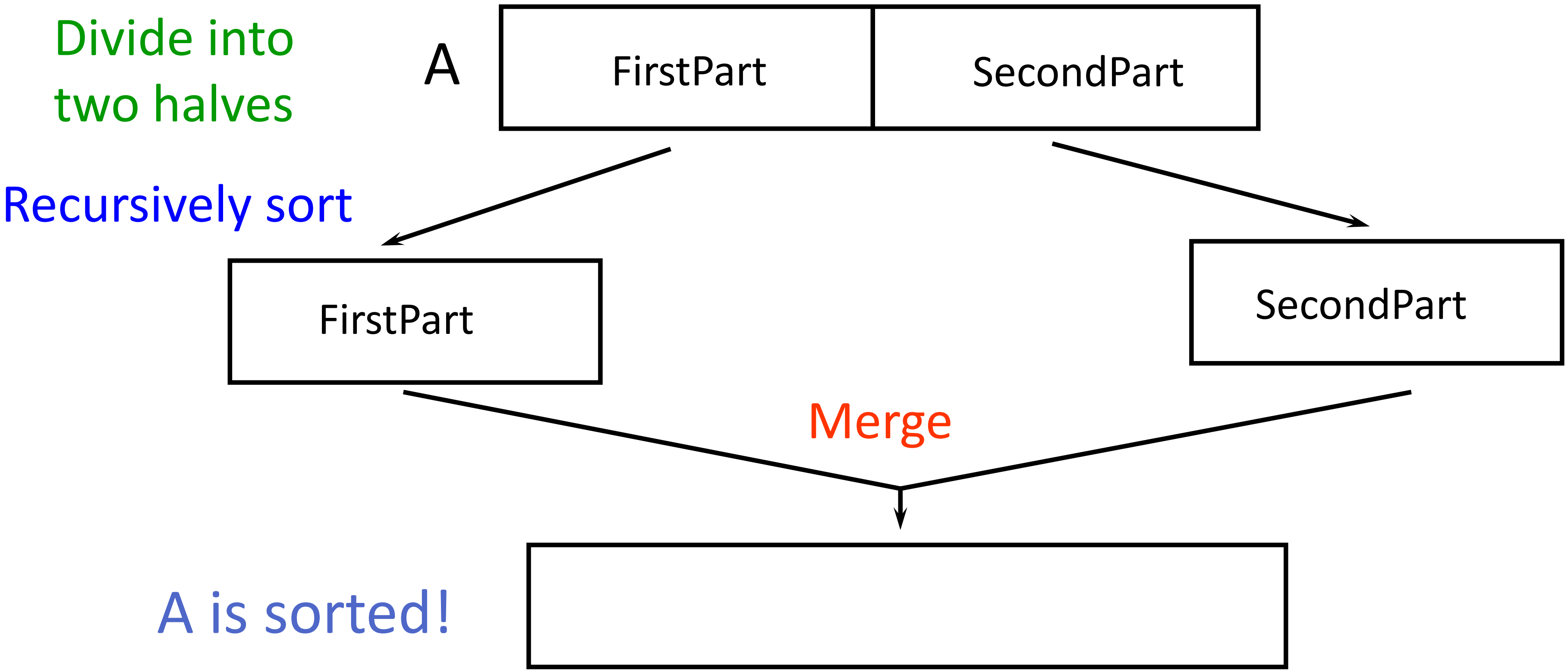
Divide and Conquer - Sort

1. Base case
   at most one element to sort, return
2. Divide A into two subarrays: FirstPart, SecondPart
   Two Subproblems:
   sort the FirstPart
   sort the SecondPart
3. Recursively
   sort FirstPart
   sort SecondPart
4. Combine sorted FirstPart and sorted SecondPart

Overview

- Divide and Conquer
- **Merge Sort**
- Quick Sort

Merge Sort: Idea

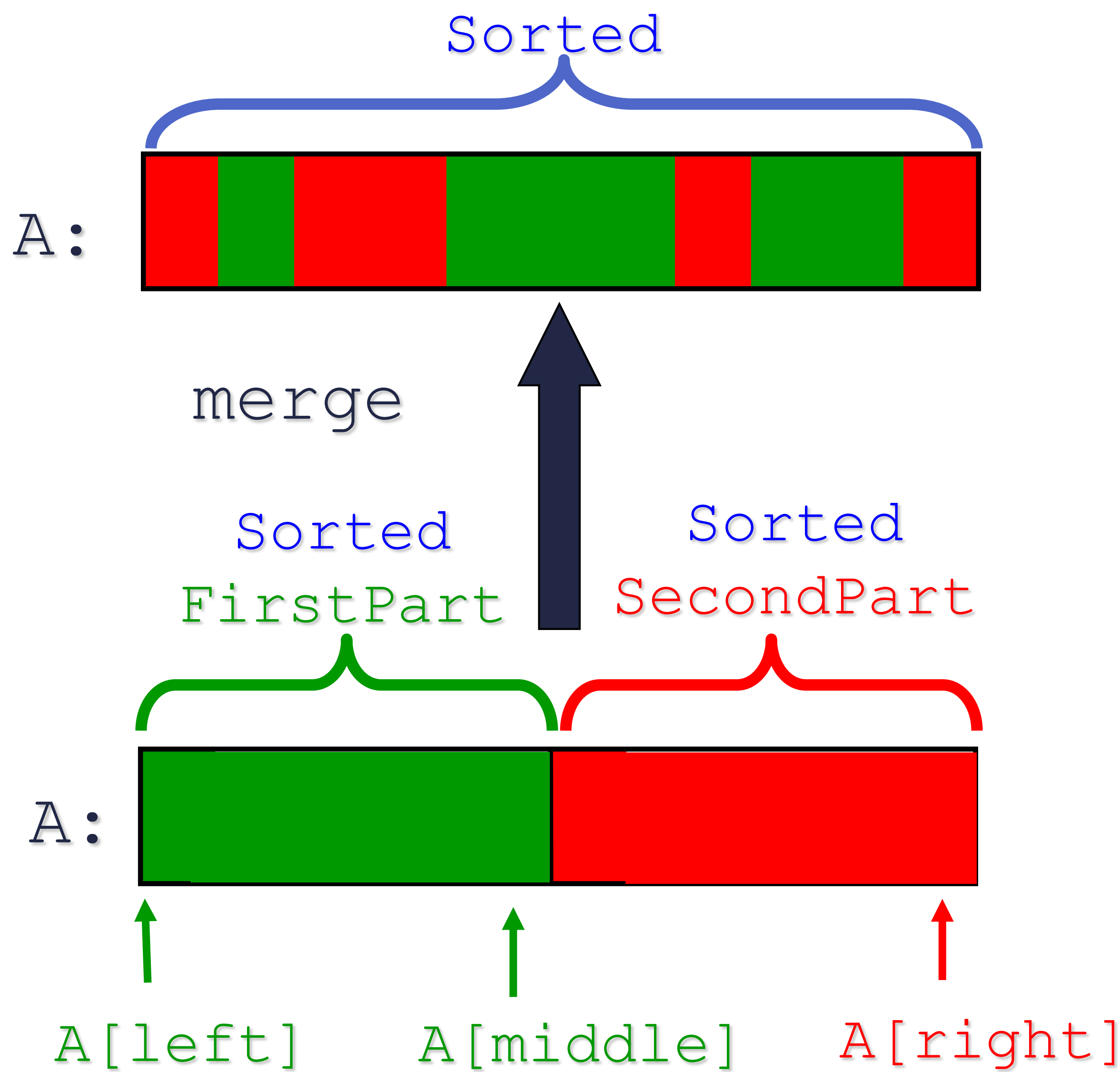


Merge Sort: Algorithm

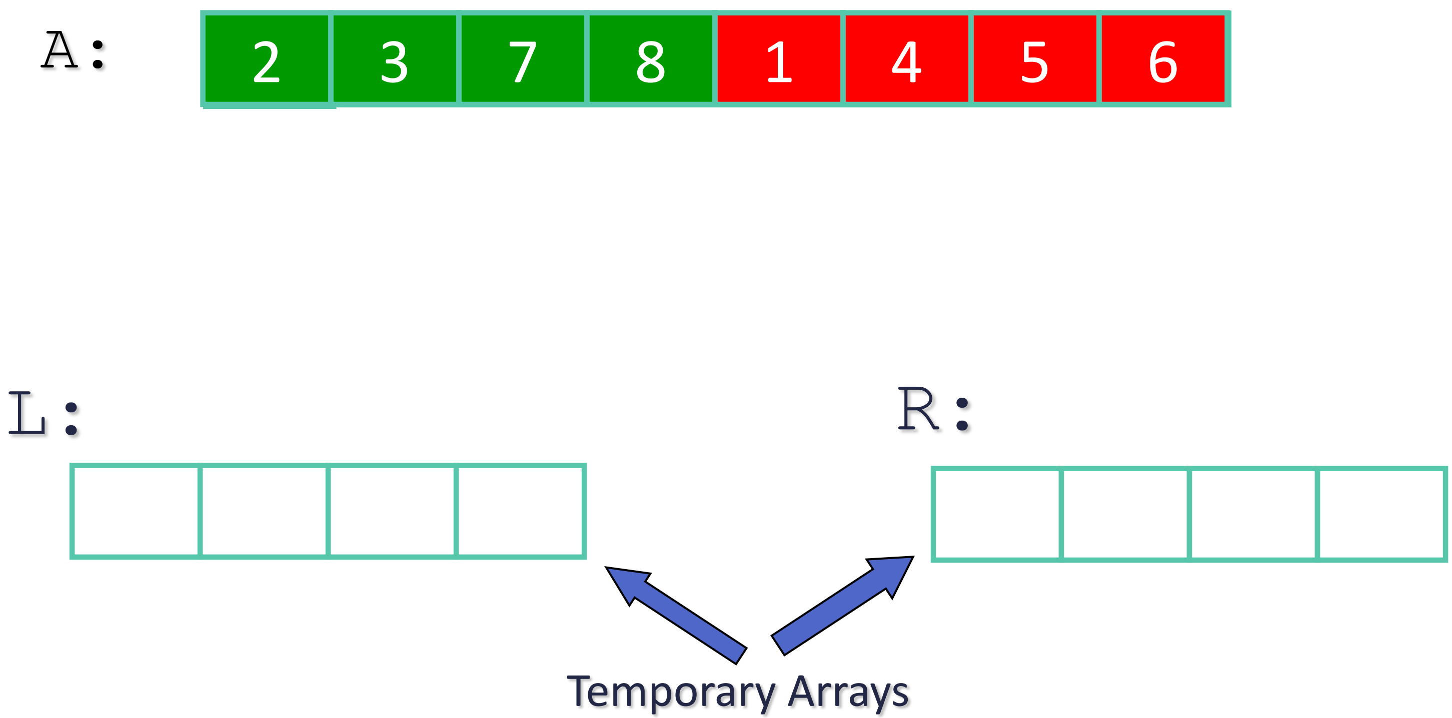
```
MergeSort (A, left, right)
  if (left >= right) return
  else {
     middle = Floor(left+right/2)
     MergeSort(A, left, middle)
     MergeSort(A, middle+1, right)
     Merge(A, left, middle, right)
   }
}
```

Recursive Call

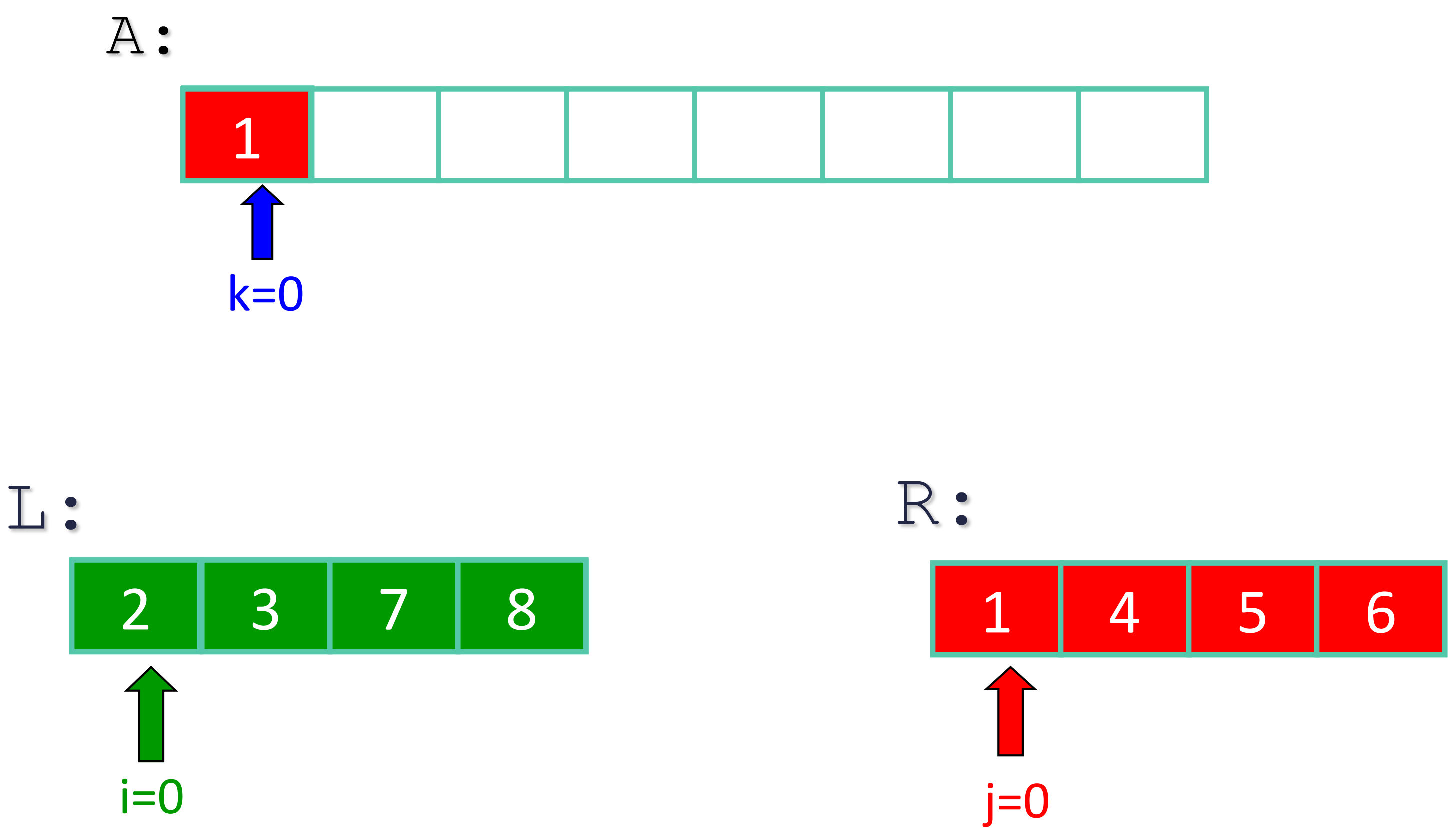
Merge-Sort: Merge



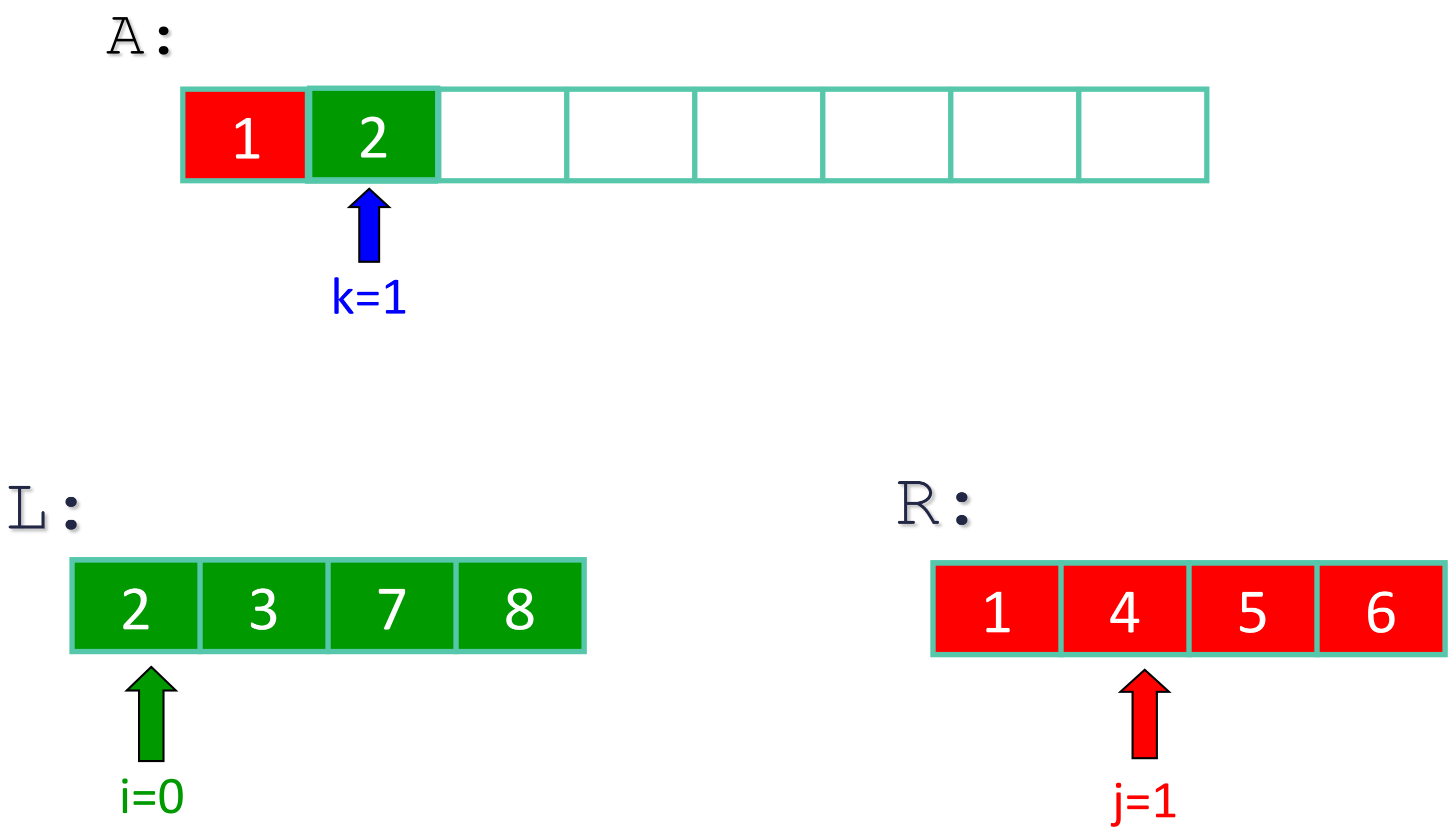
Merge-Sort: Merge



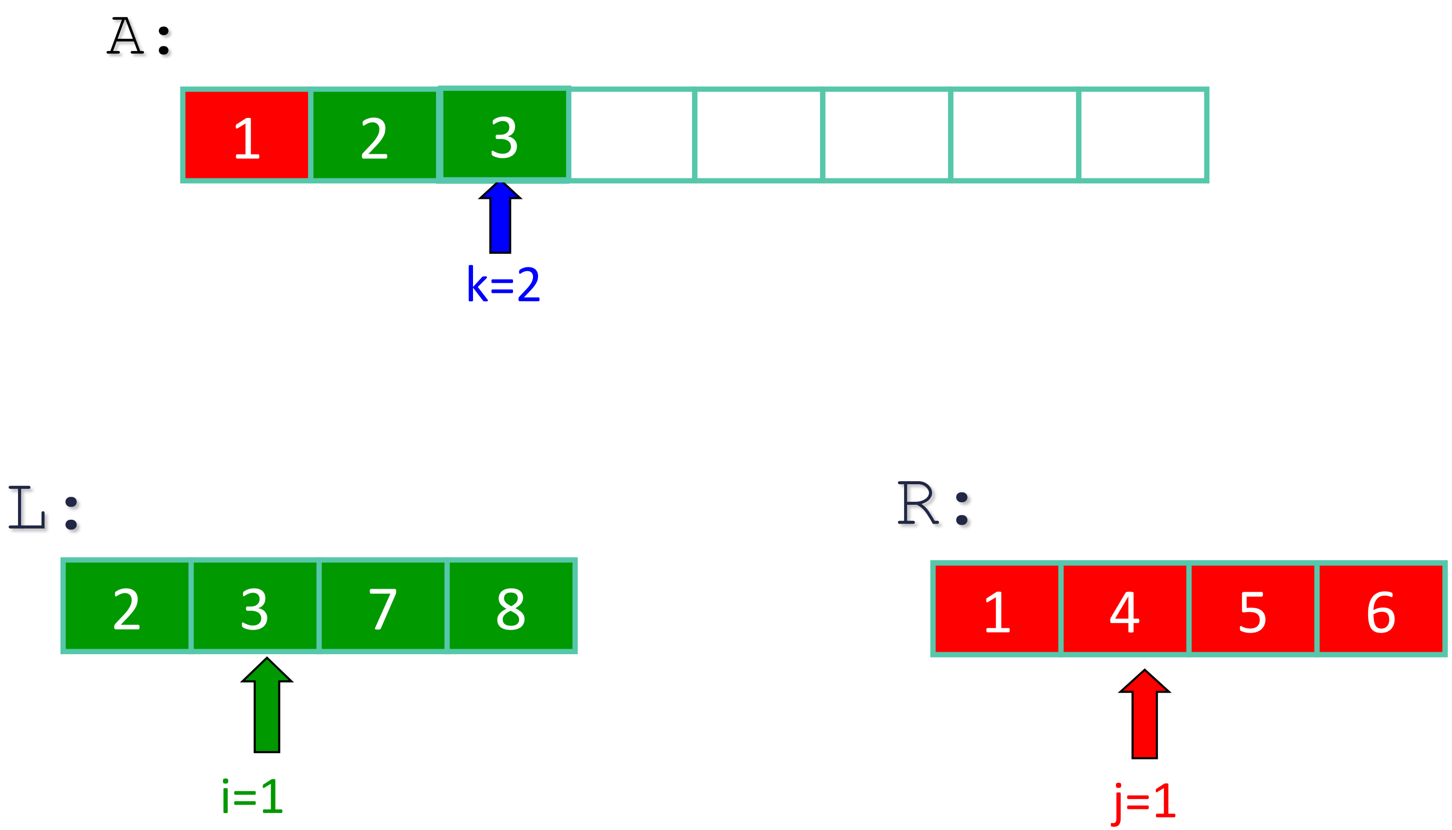
Merge-Sort: Merge



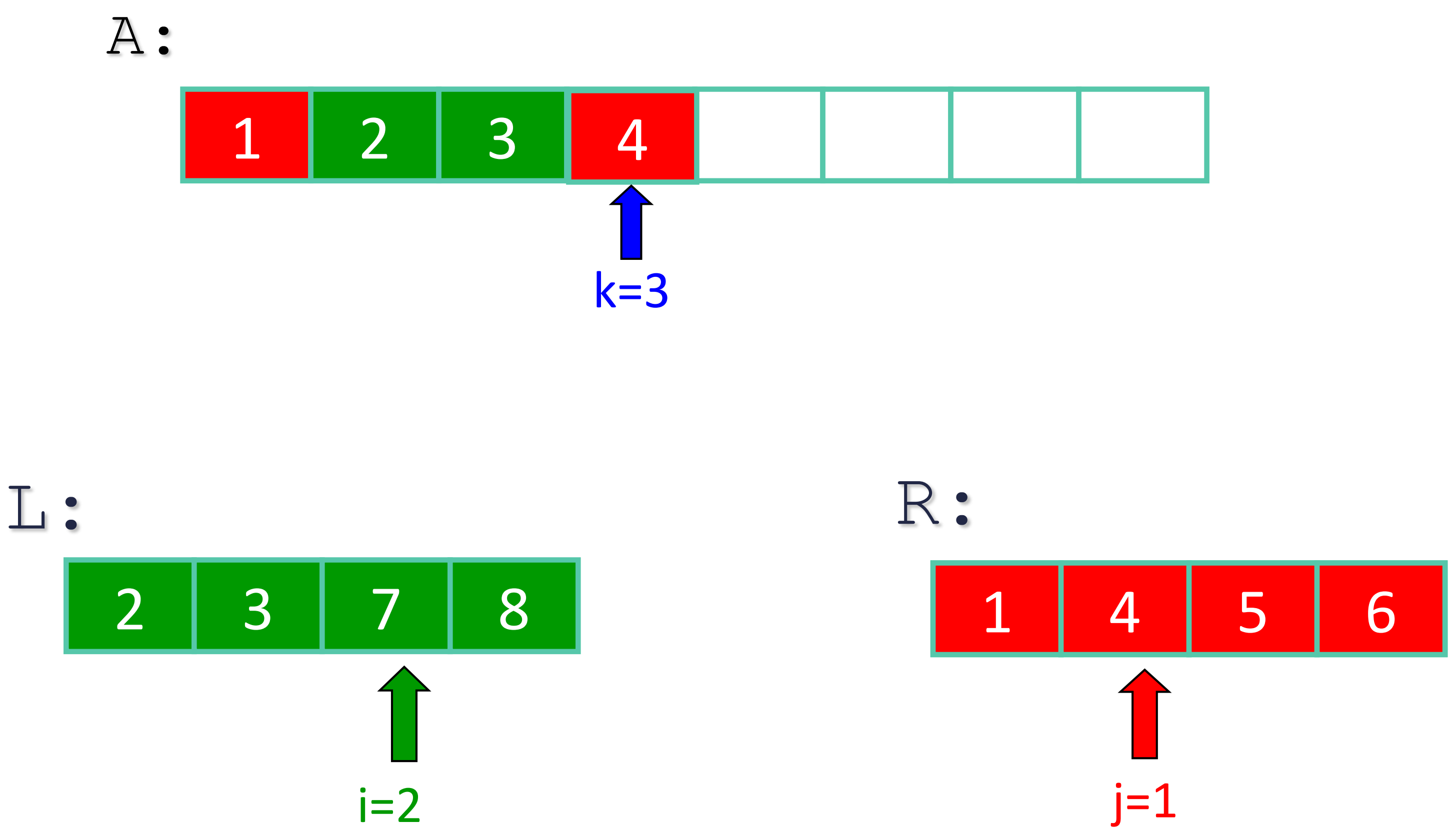
Merge-Sort: Merge



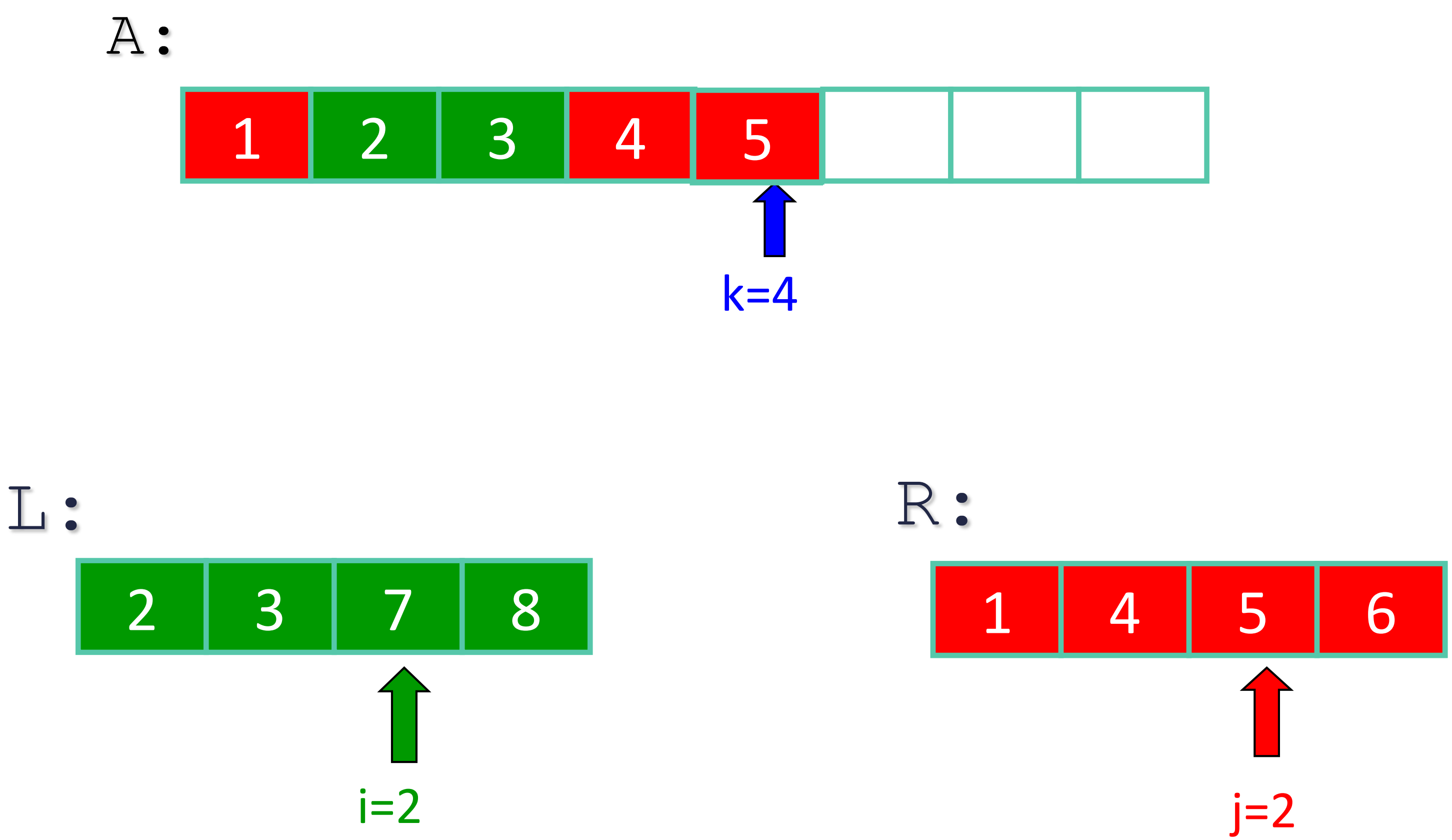
Merge-Sort: Merge



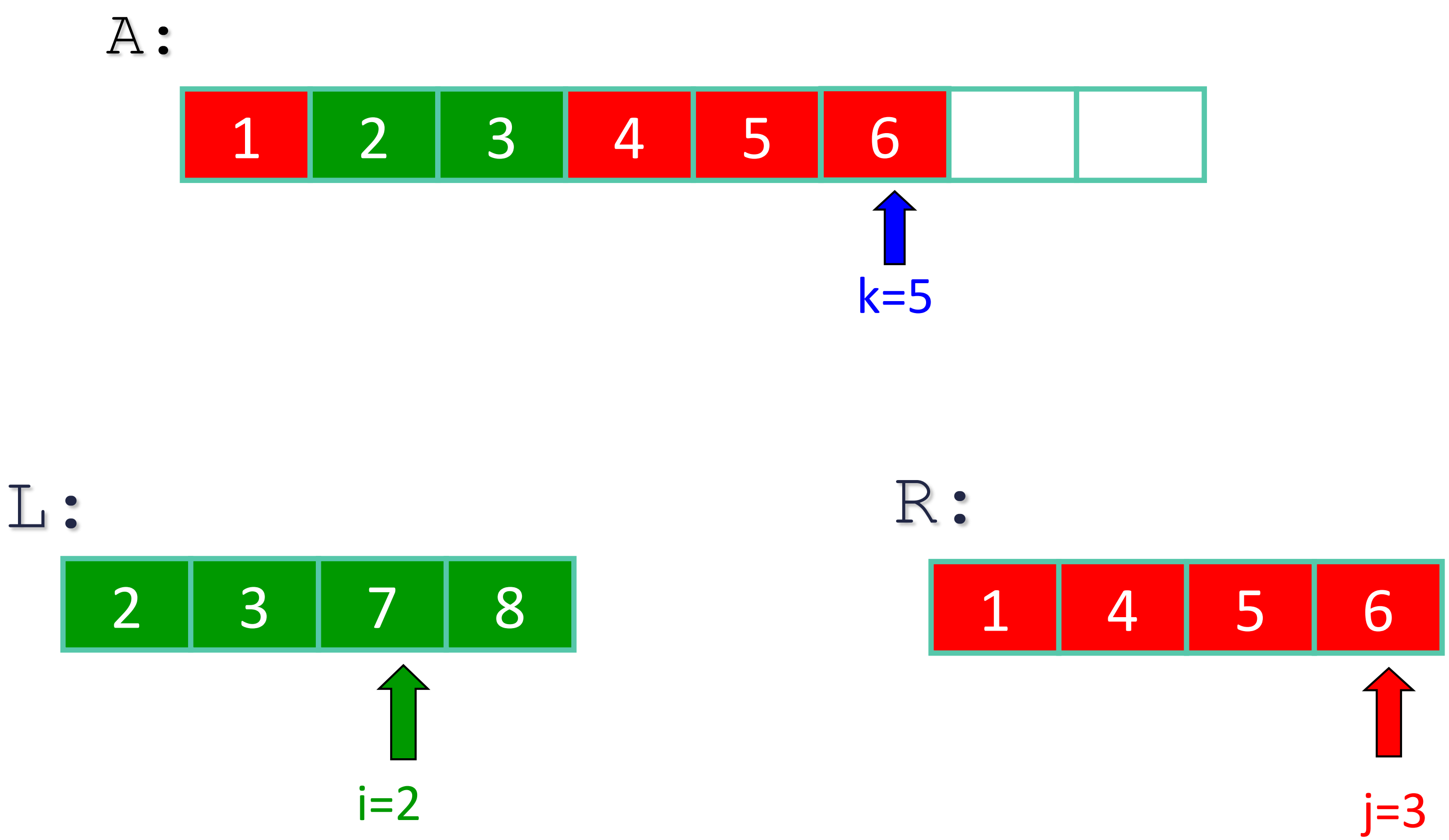
Merge-Sort: Merge



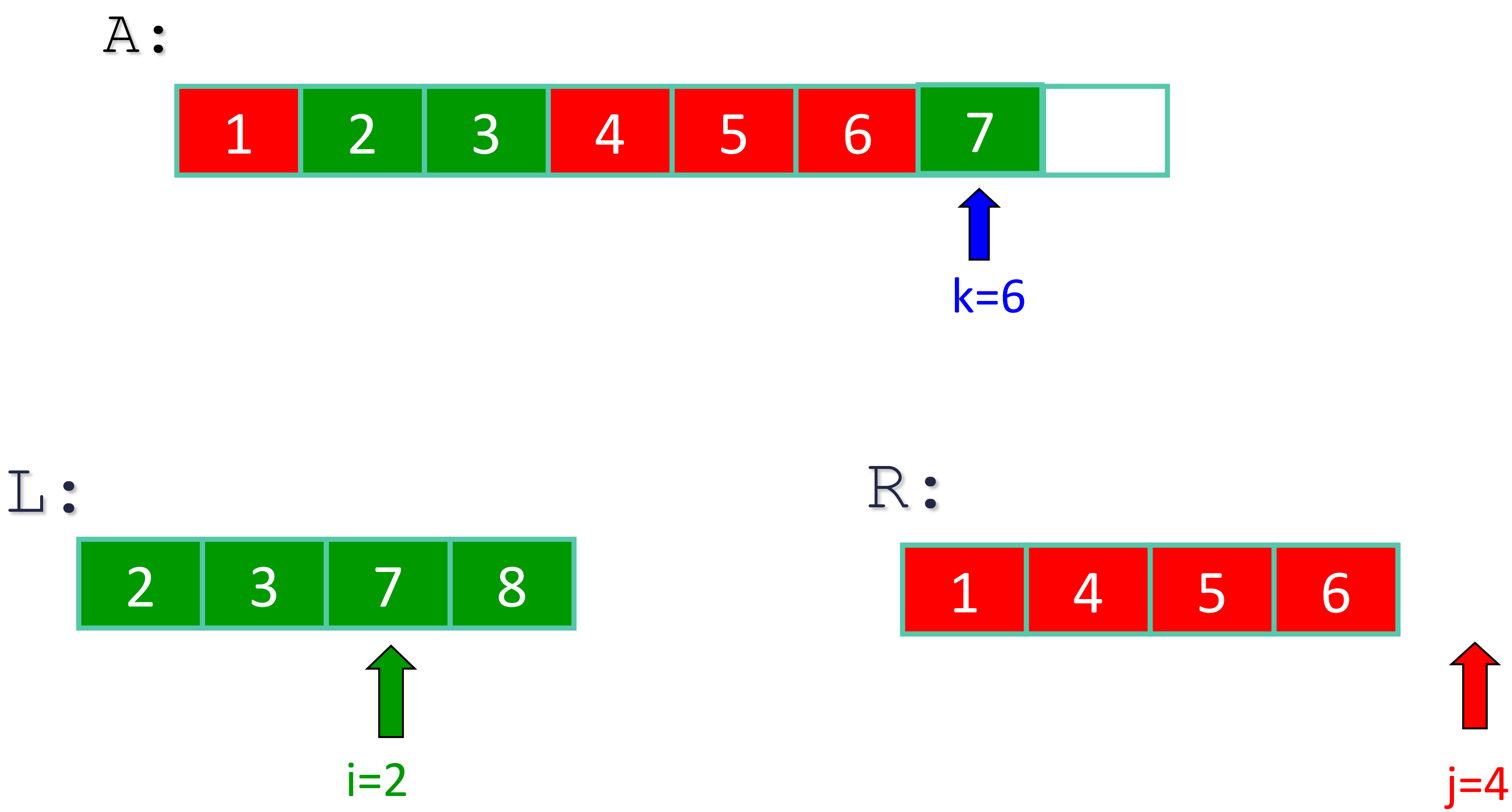
Merge-Sort: Merge



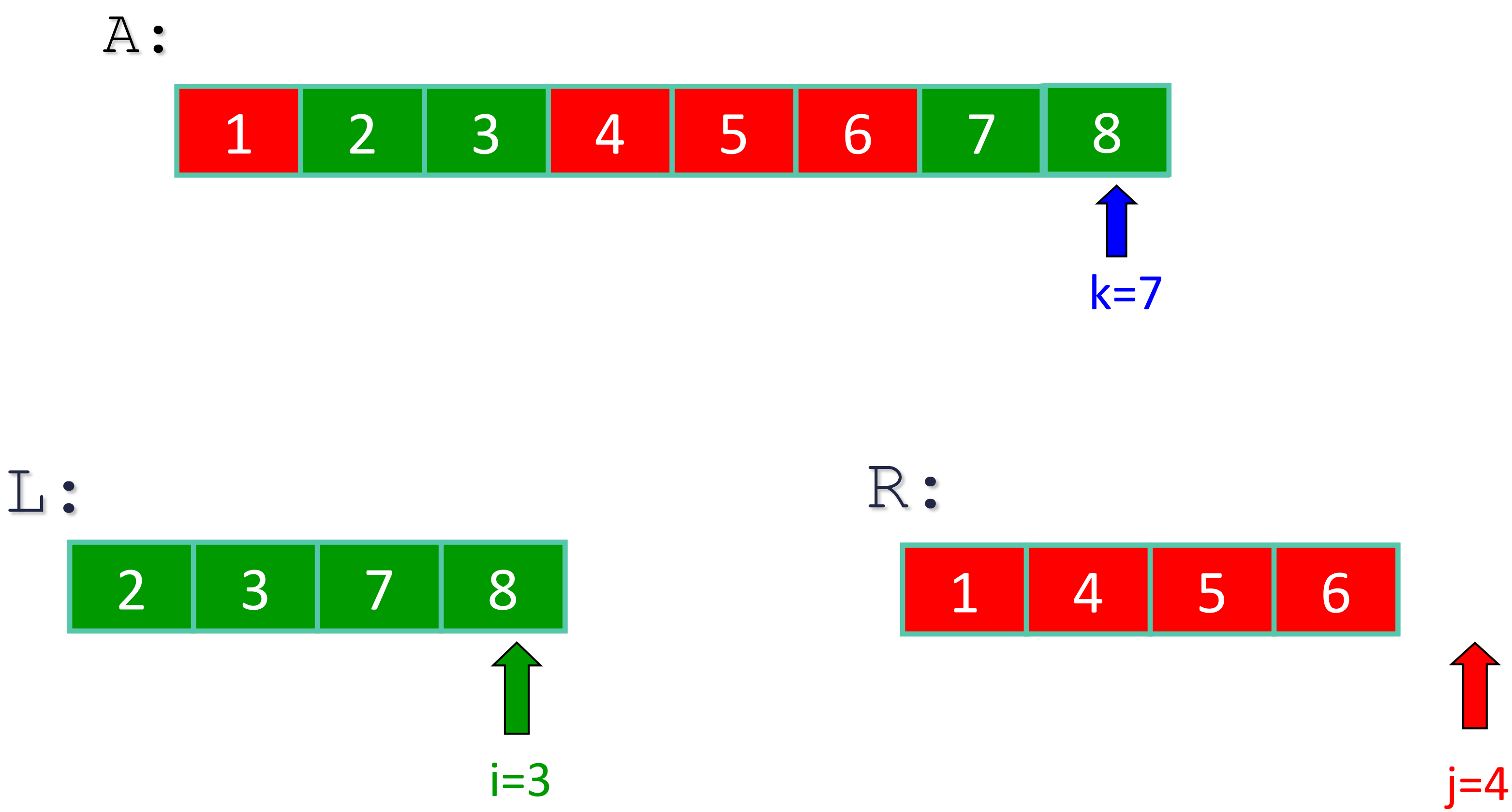
Merge-Sort: Merge



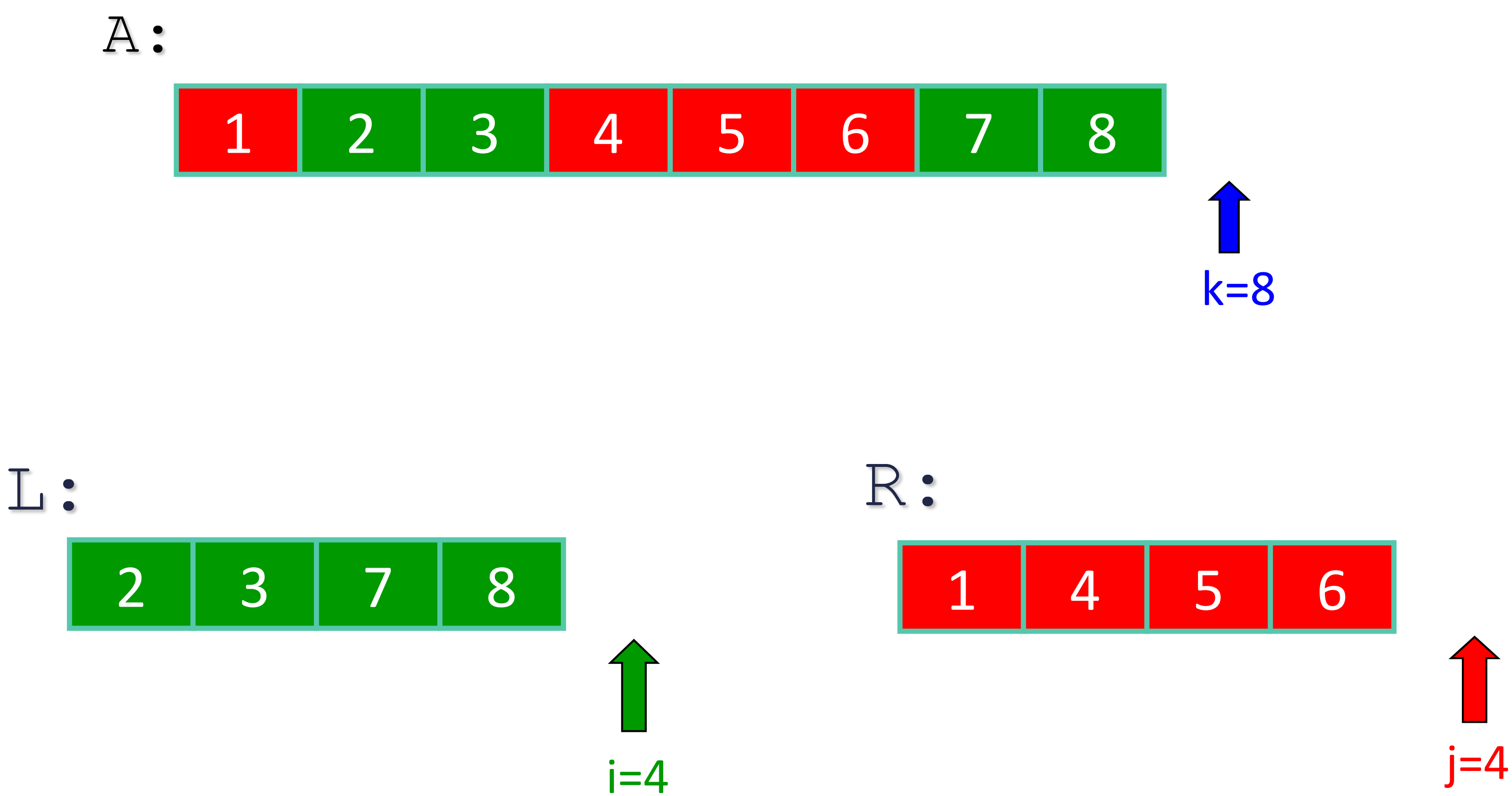
Merge-Sort: Merge



Merge-Sort: Merge



Merge-Sort: Merge



```
Merge(A, left, middle, right)
  n1 = middle - left + 1
  n2 = right - middle
  create array L[n1], R[n2]
  for i = 0 to n1-1 do L[i] = A[left+i]
  for j = 0 to n2-1 do R[j] = A[middle+j]
  k = i = j = 0
  while i < n1 &  j < n2
    if L[i] < R[j]
      A[k++] = L[i++]
    else
      A[k++] = R[j++]
  while i < n1
    A[k++] = L[i++]
  while j < n2
    A[k++] = R[j++]
```

$n = n_1+n_2$

Space: n

Time : cn for some constant c

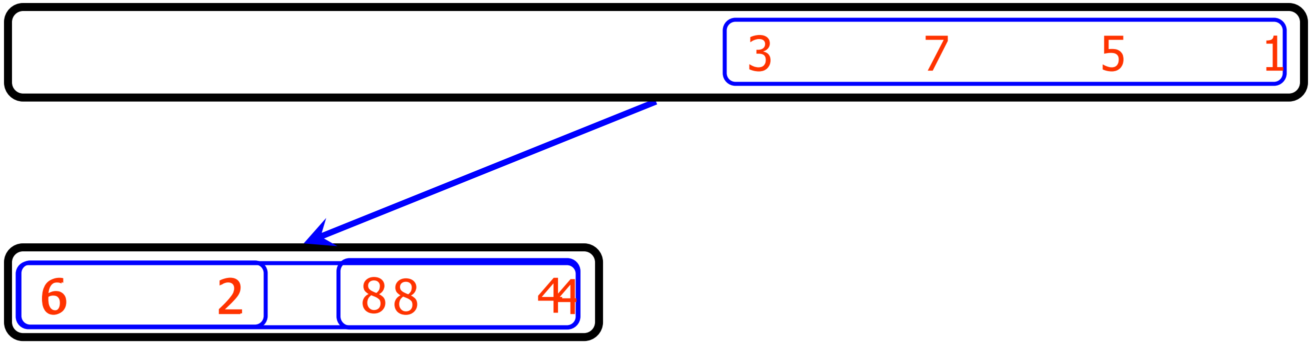
MergeSort(A,0,7)

Divide

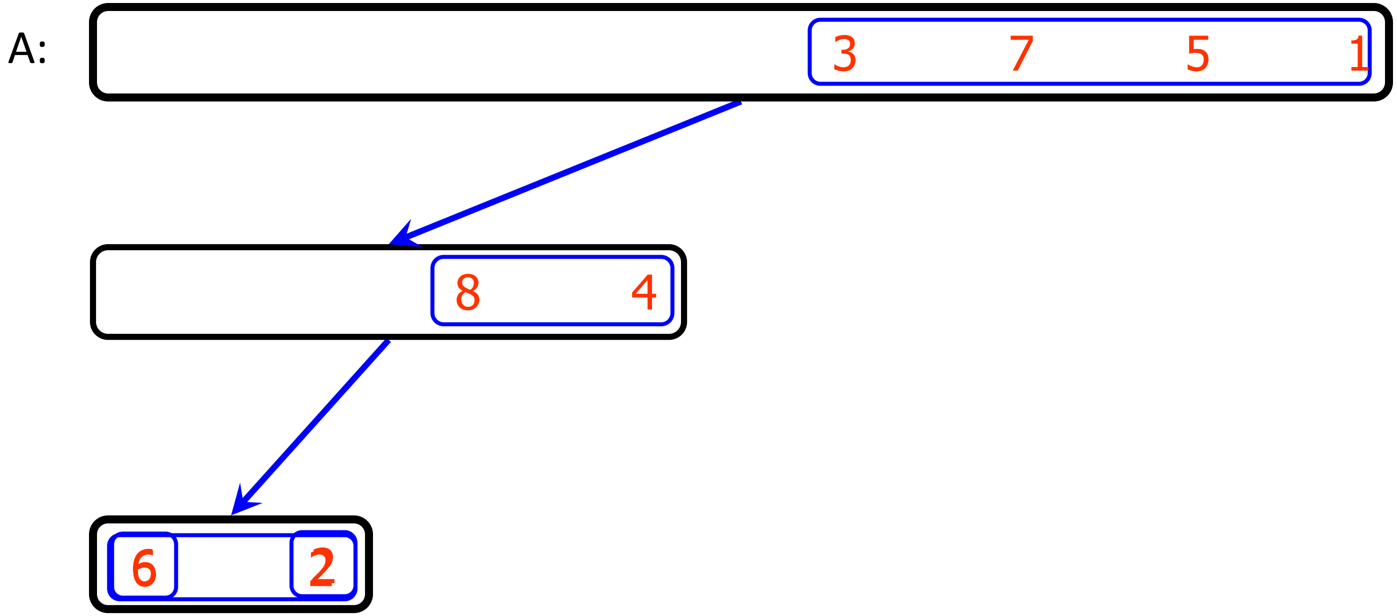
A:								
	6	2	8	4	3	7	5	1

Merge-Sort(A, 0, 3) , divide

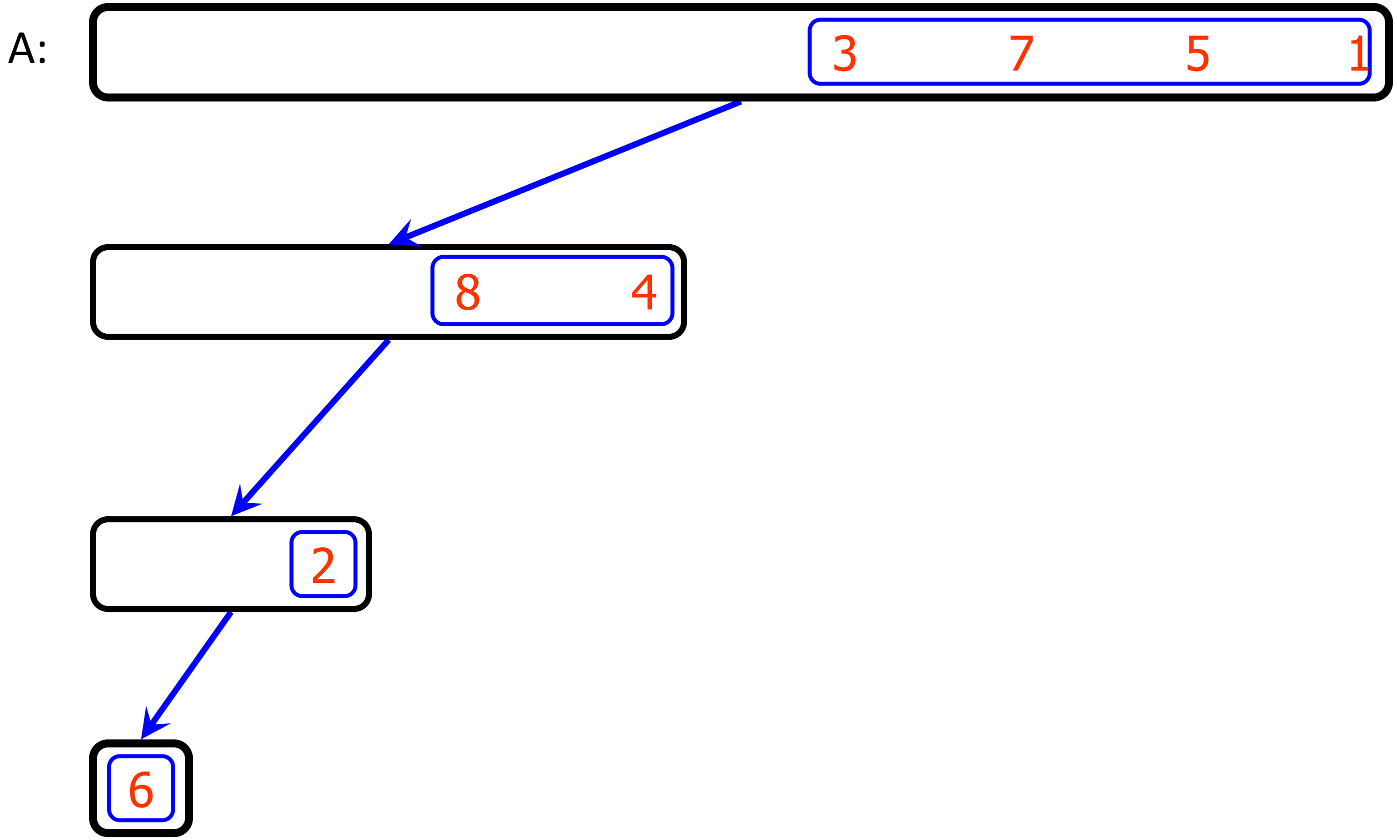
A:



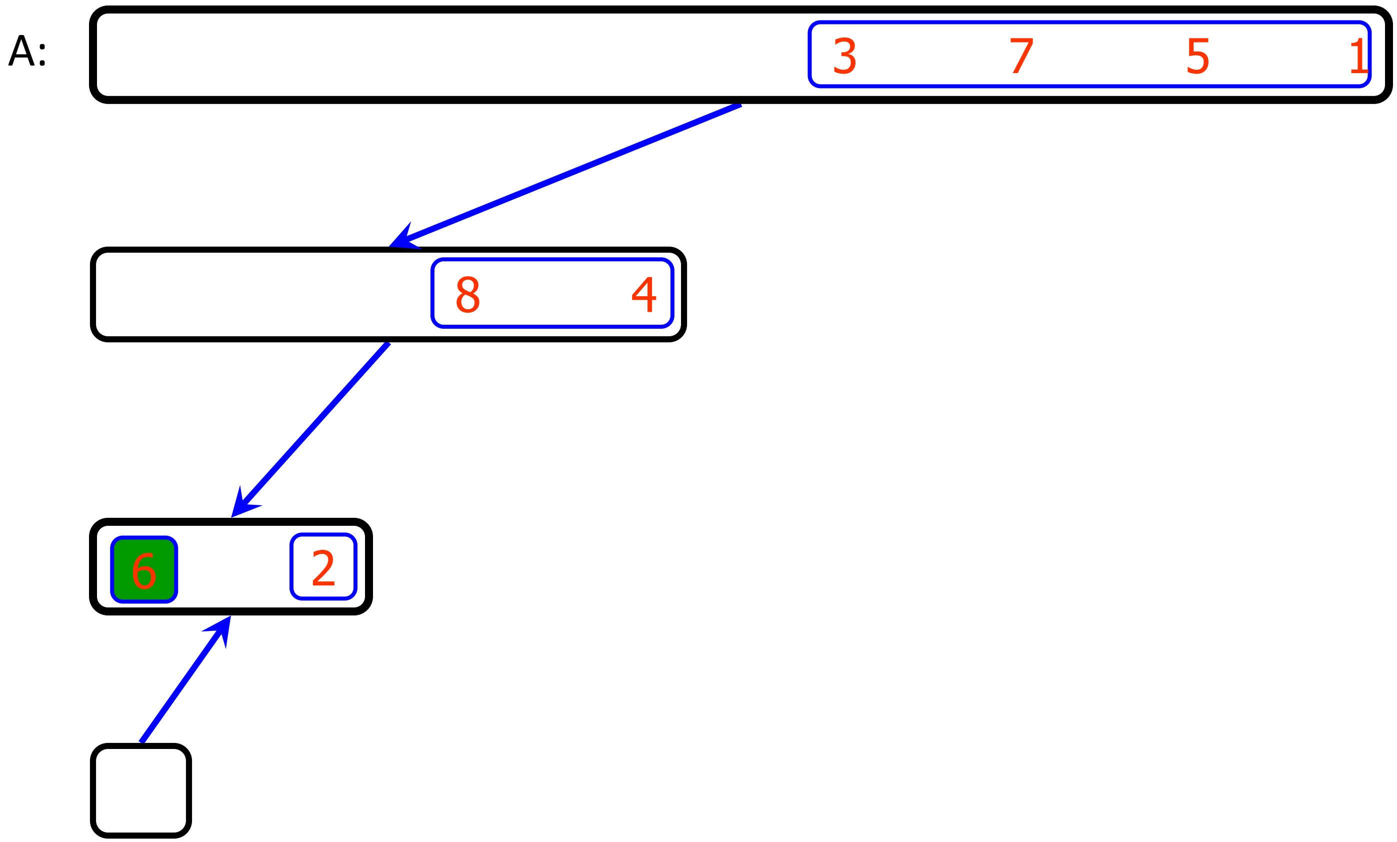
Merge-Sort(A, 0, 1) , divide



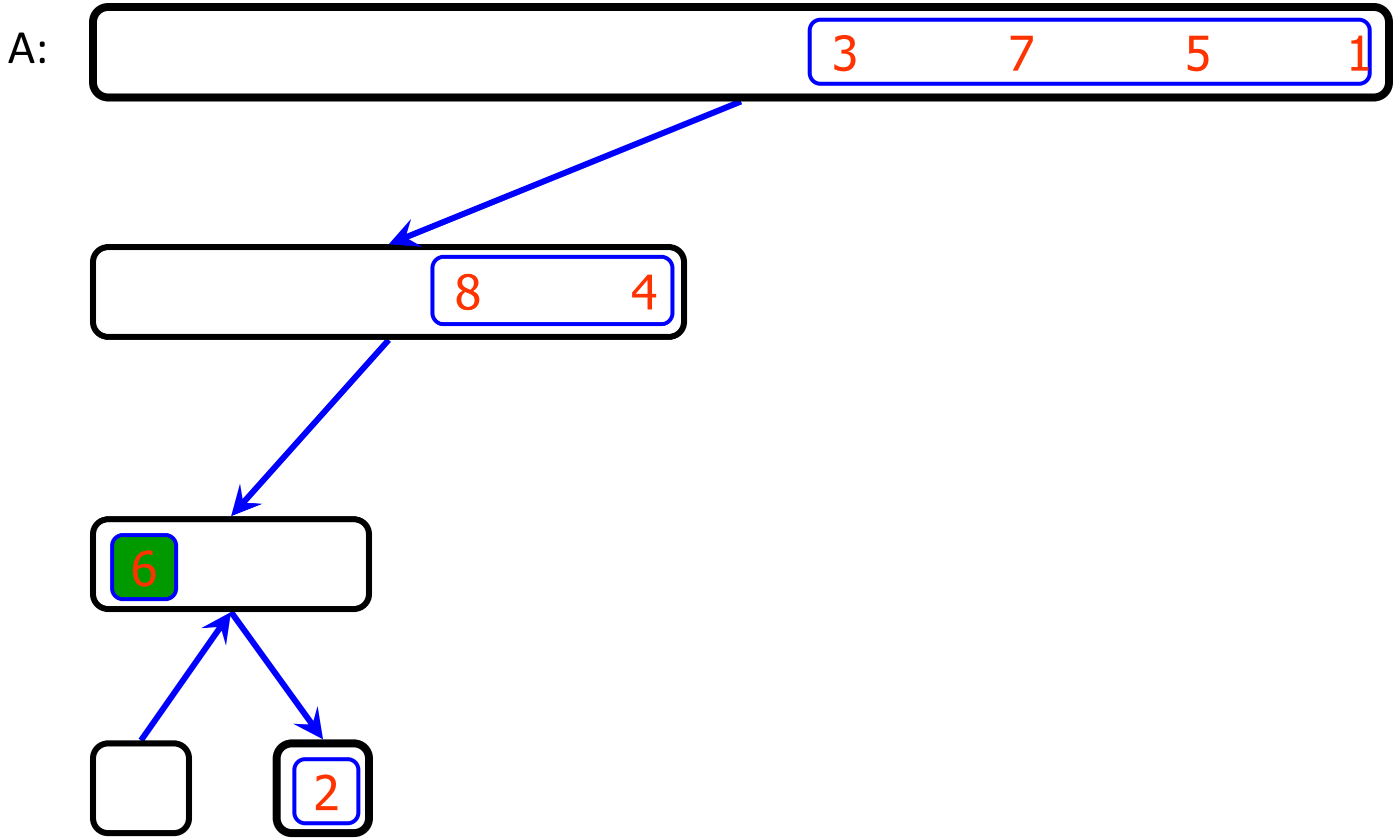
Merge-Sort(A, 0, 0) , base case



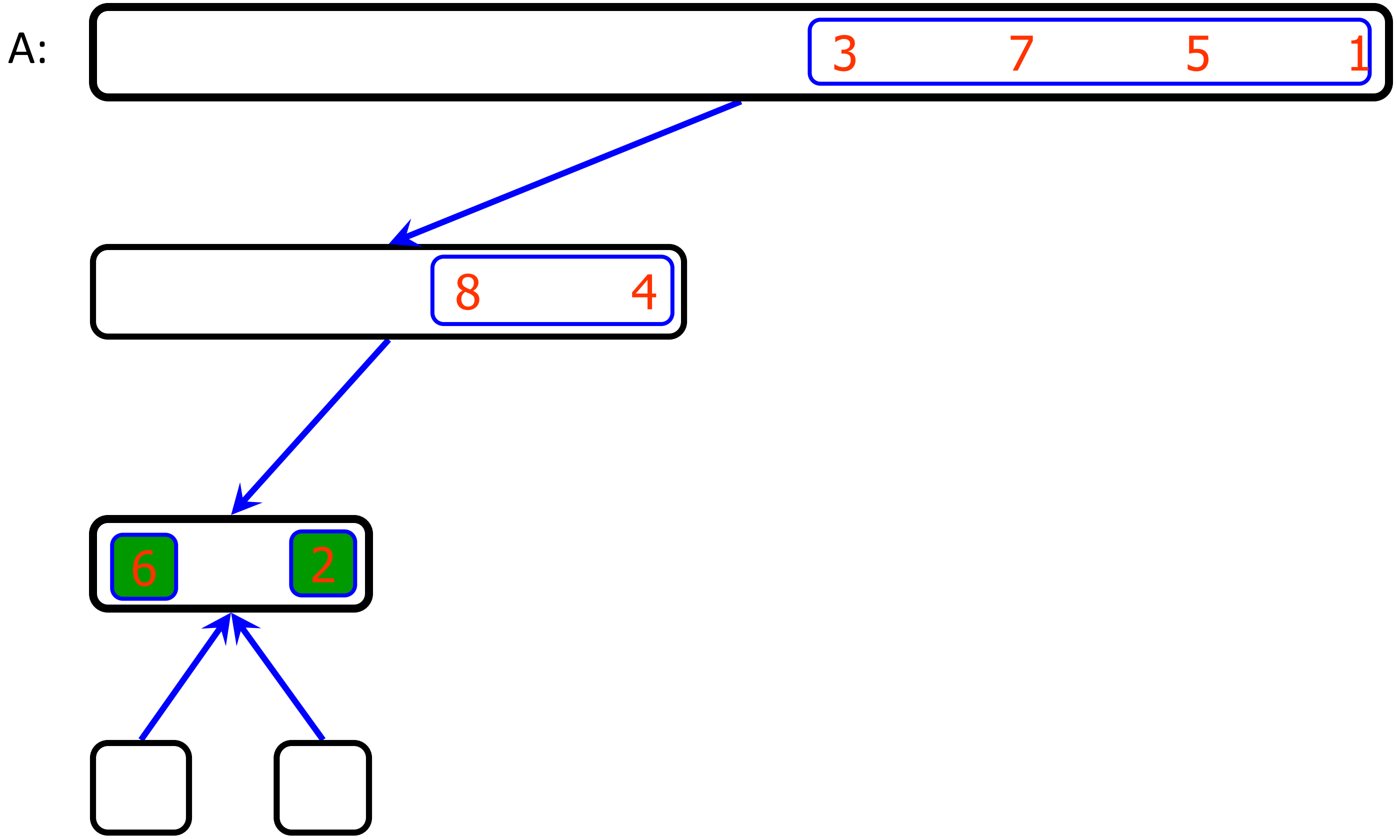
Merge-Sort(A, 0, 0), return



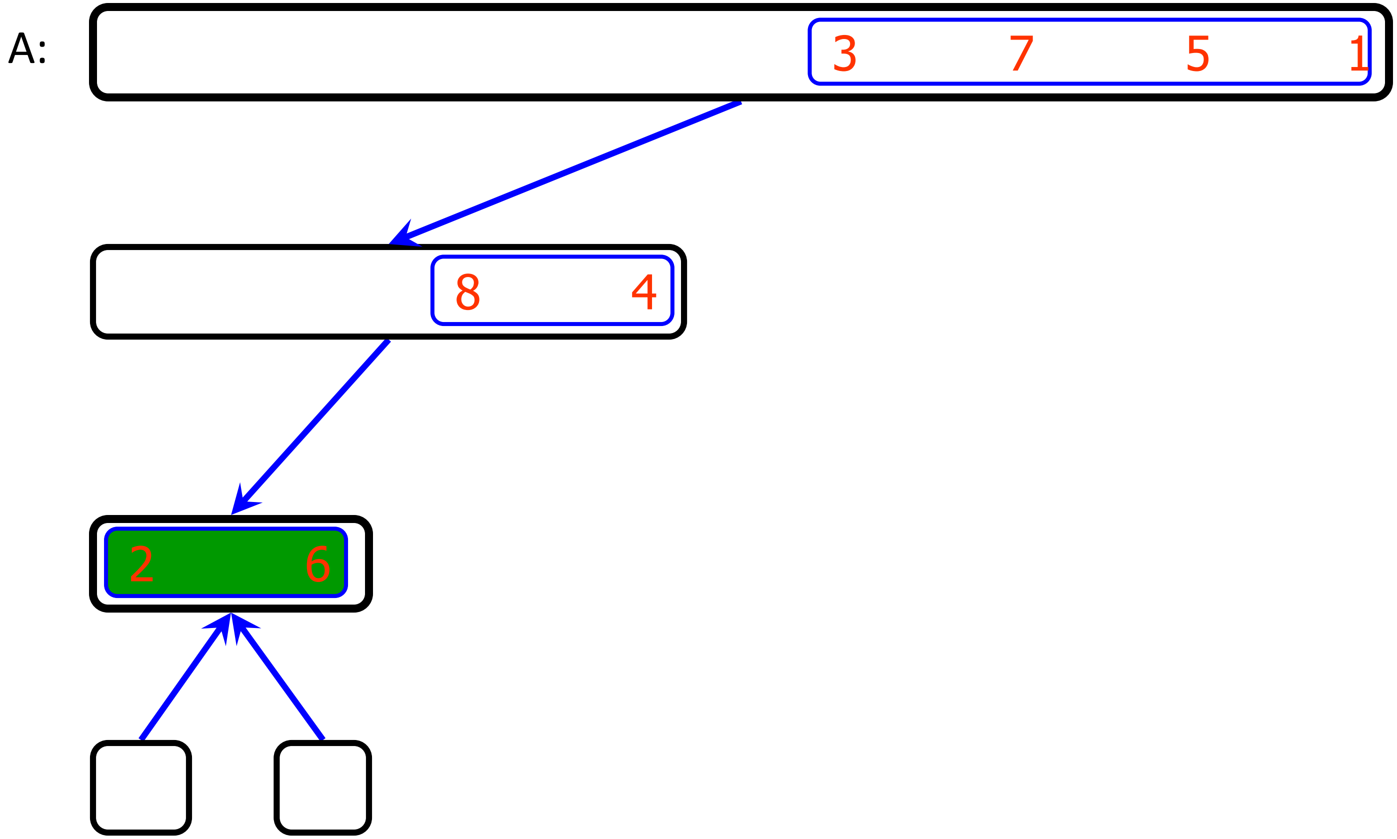
Merge-Sort(A, 1, 1) , base case



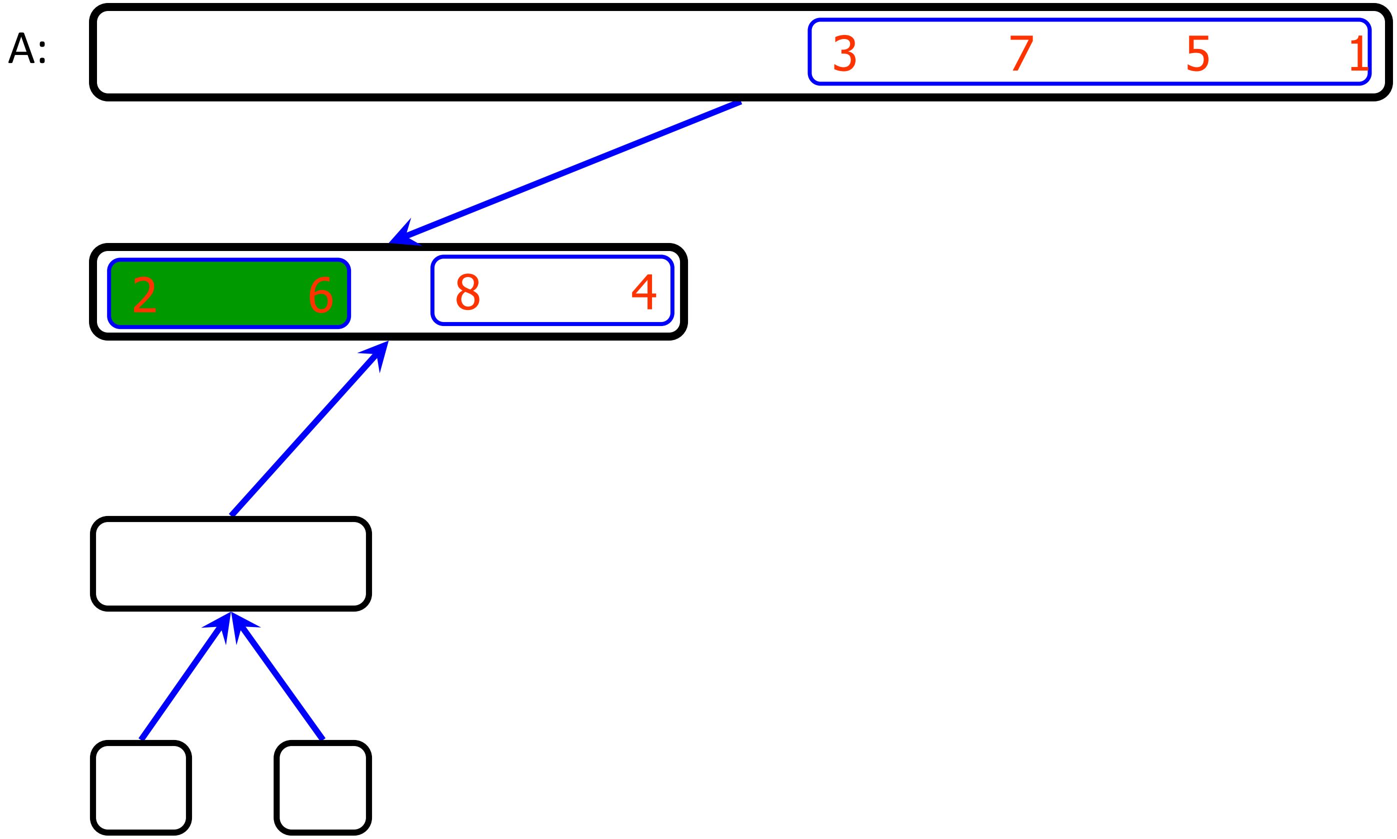
Merge-Sort(A, 1, 1), return



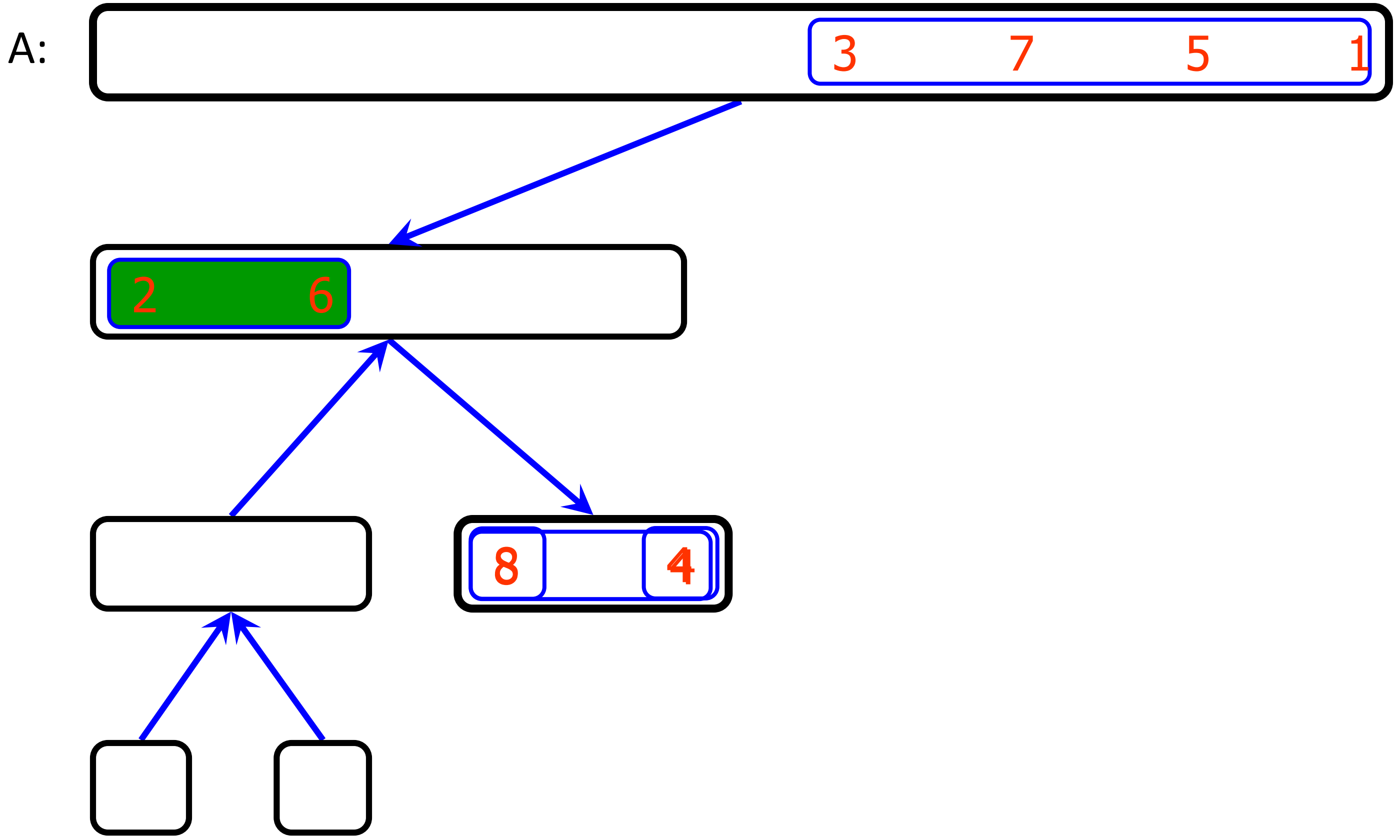
Merge(A, 0, 0, 1)



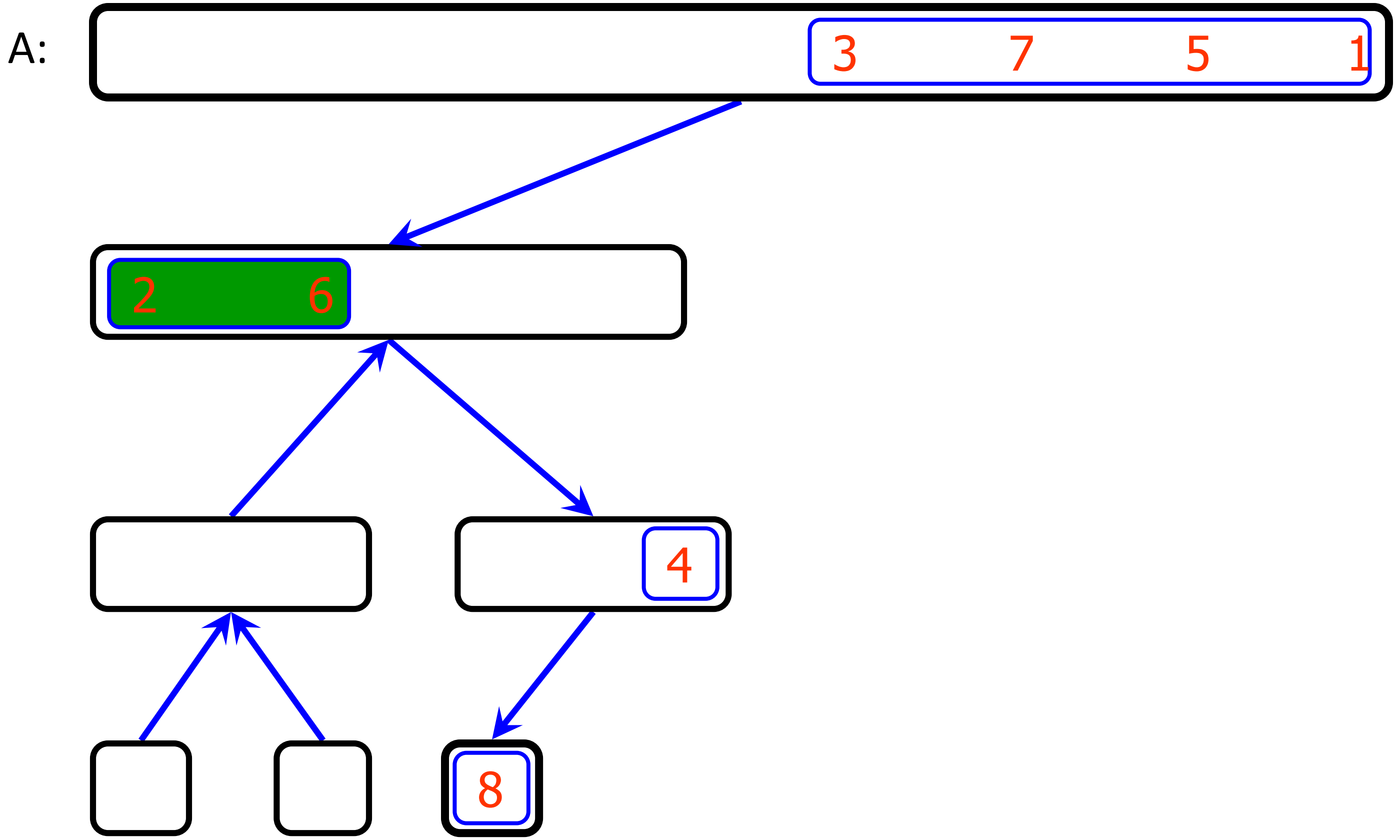
Merge-Sort(A, 0, 1), return



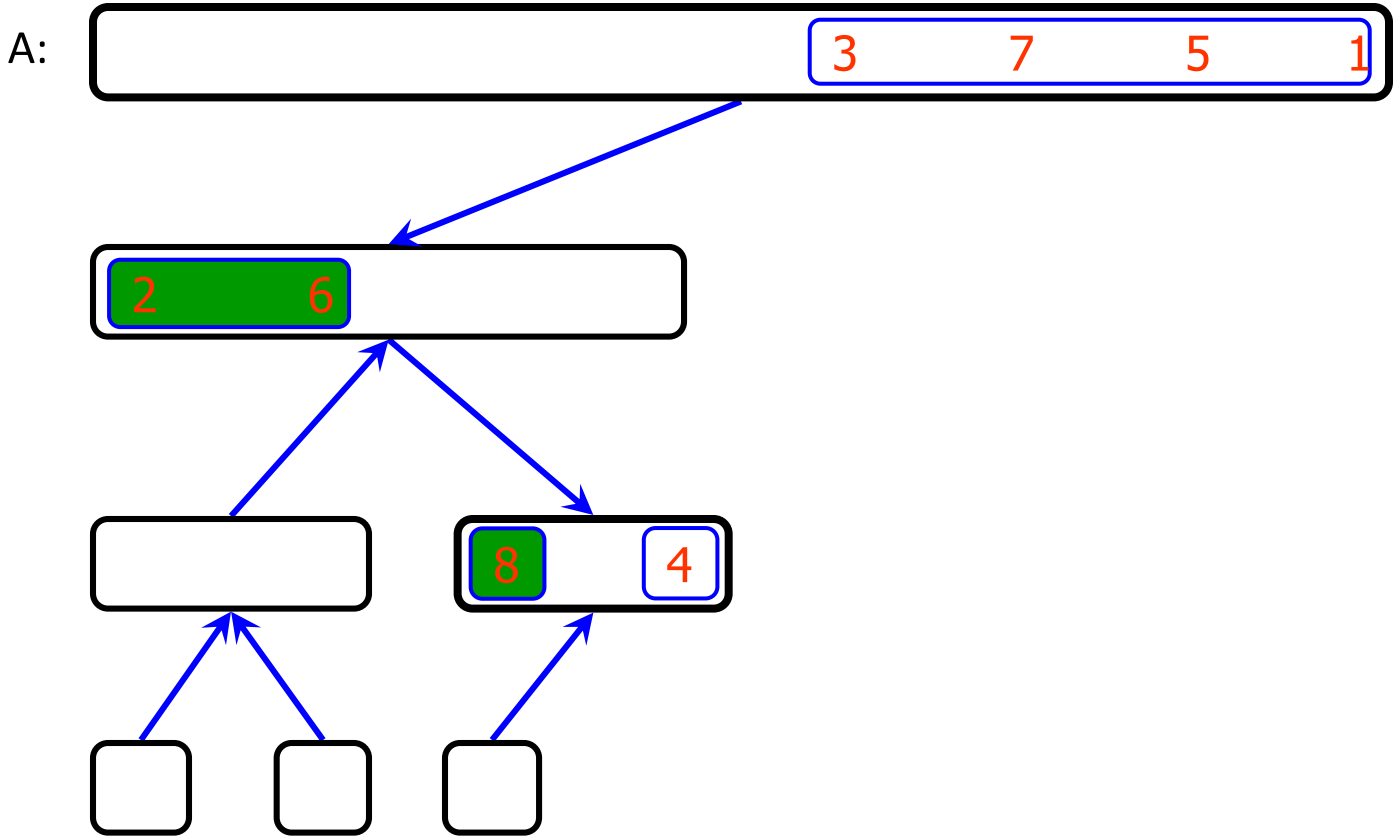
Merge-Sort(A, 2, 3) , divide



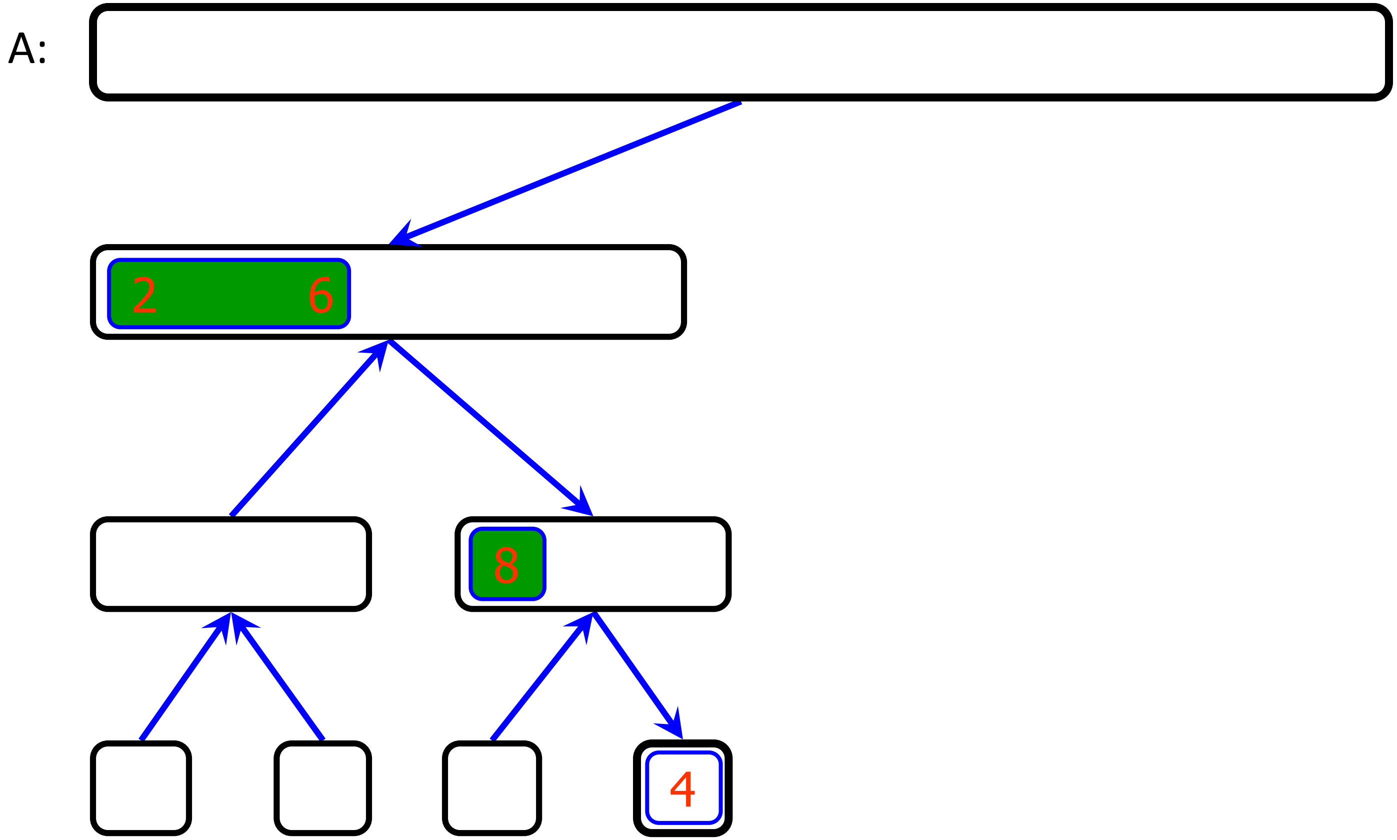
Merge-Sort(A, 2, 2), base case



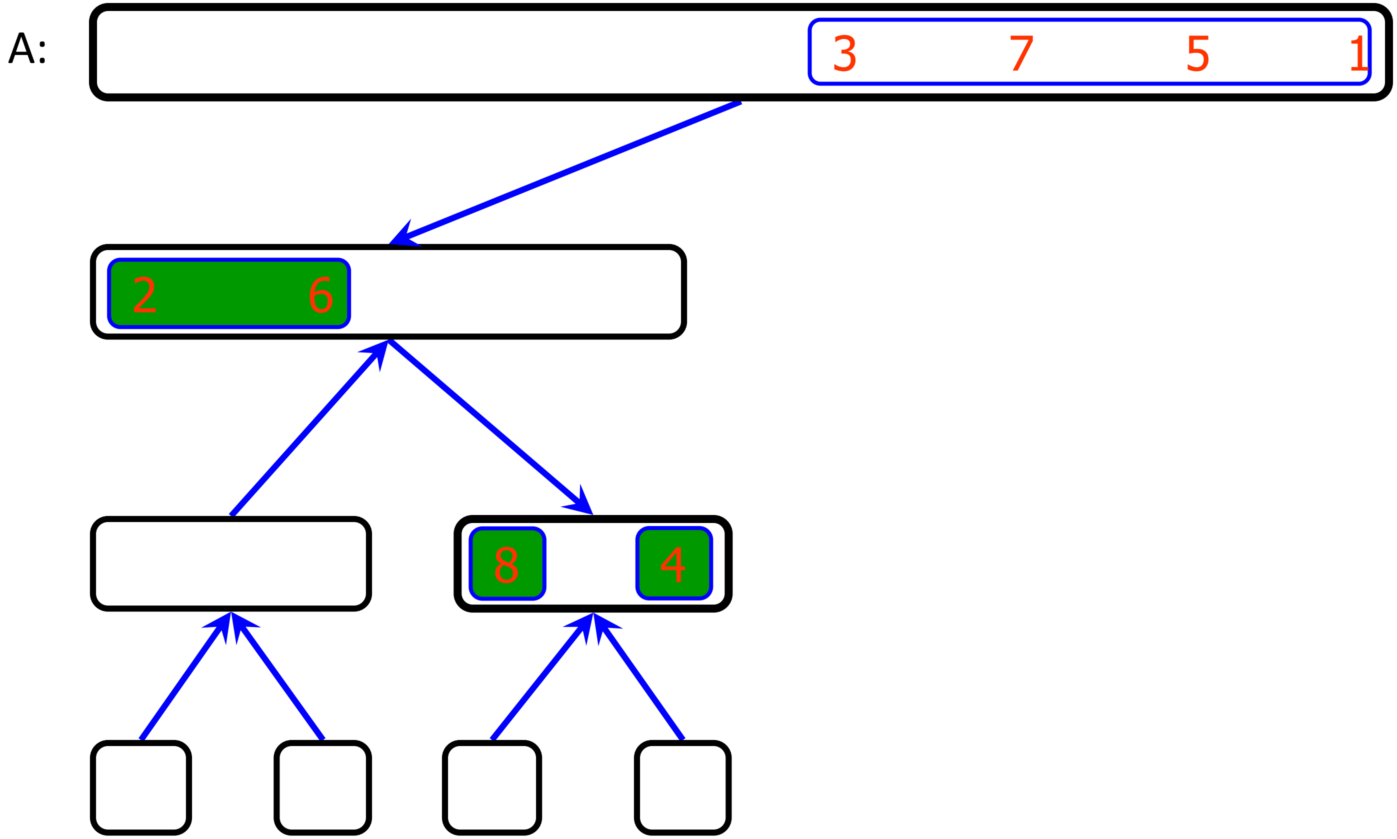
Merge-Sort(A, 2, 2), return



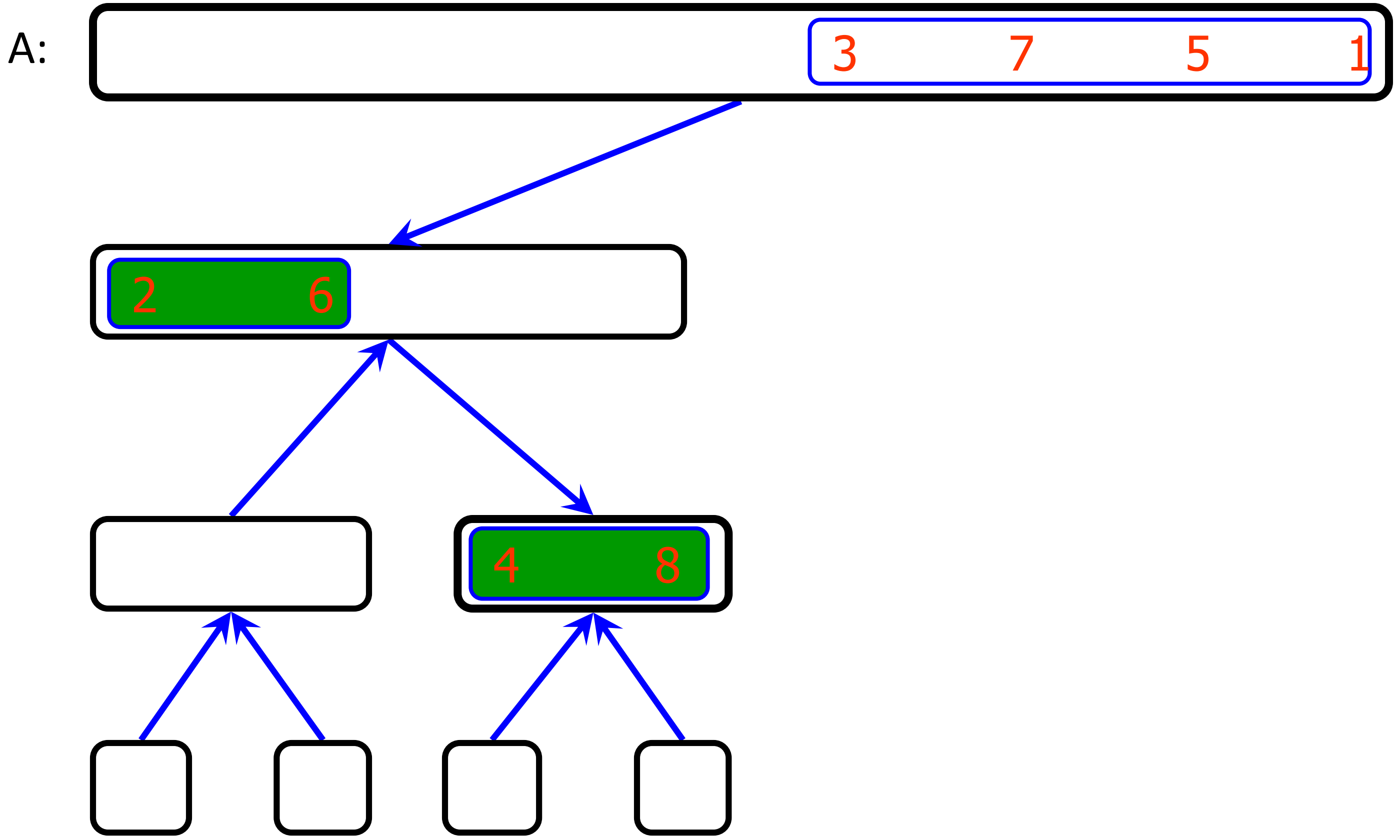
Merge-Sort(A, 3, 3), base case



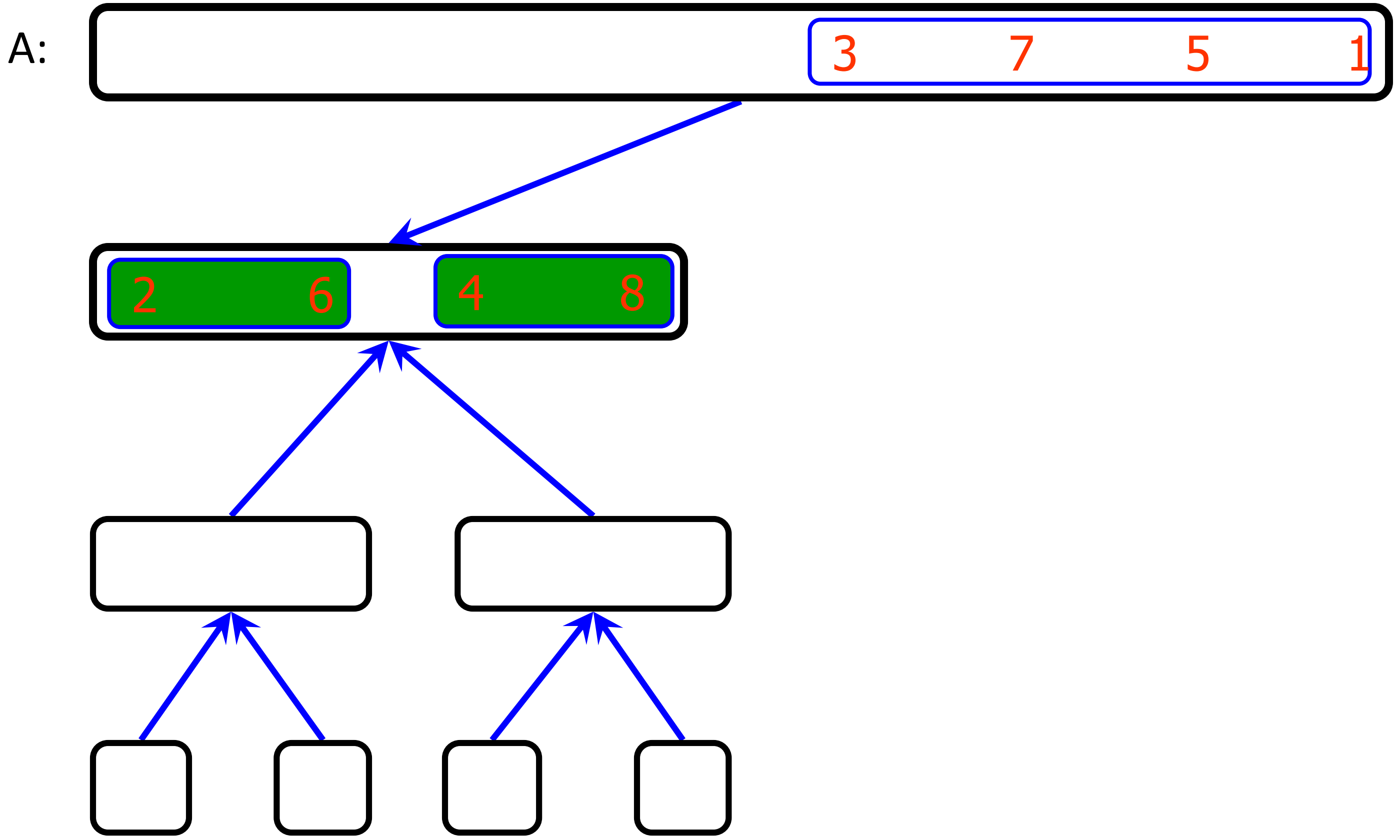
Merge-Sort(A, 3, 3), return



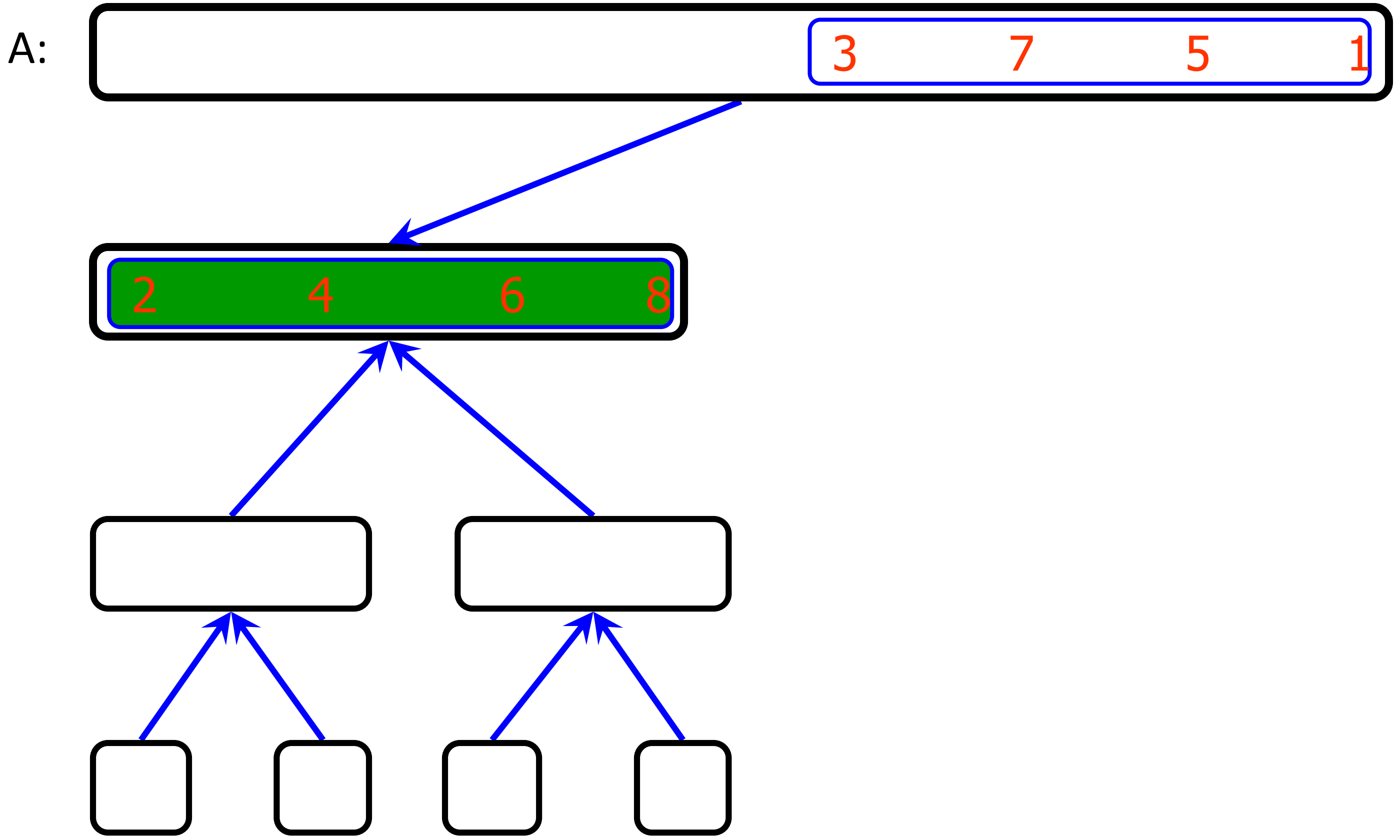
Merge(A, 2, 2, 3)



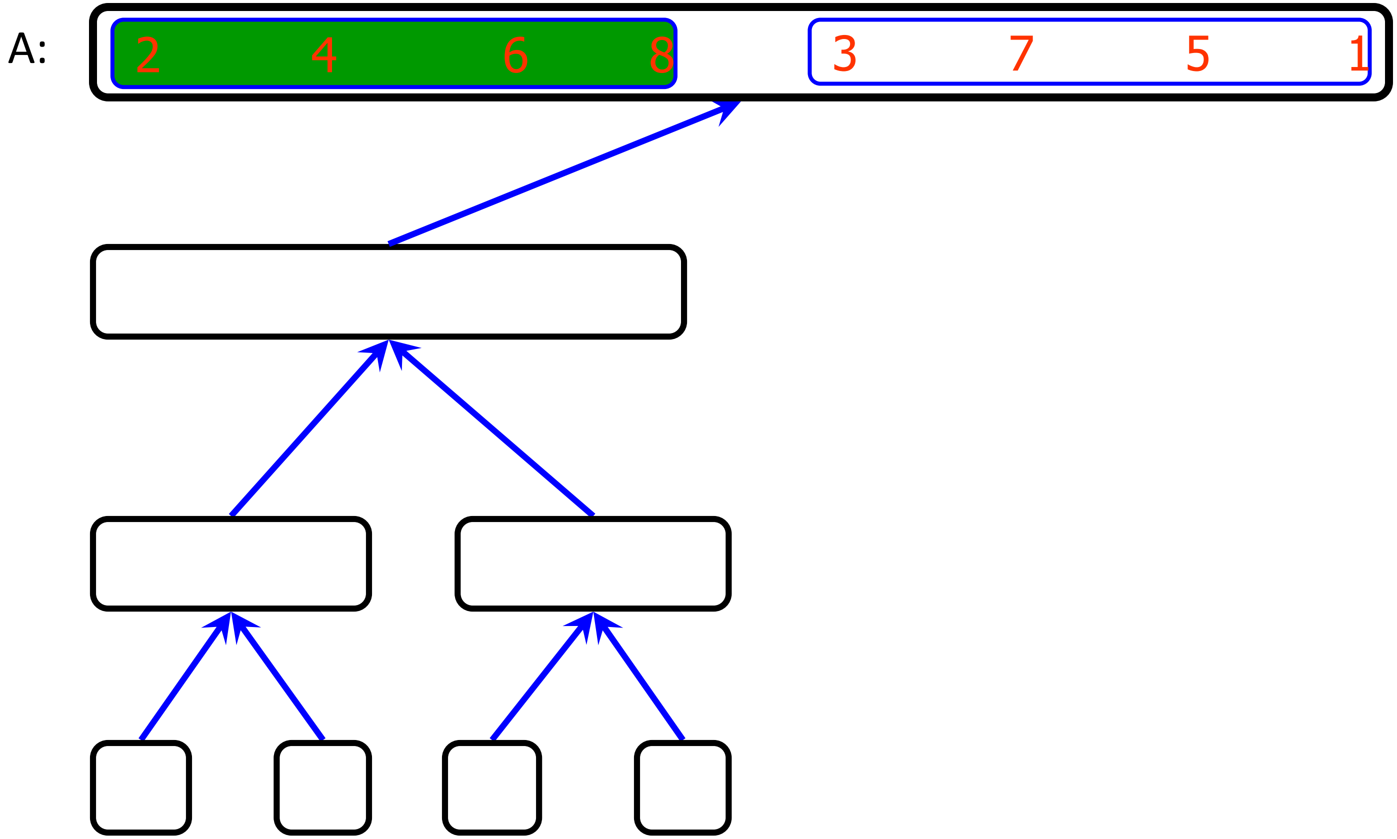
Merge-Sort(A, 2, 3), return



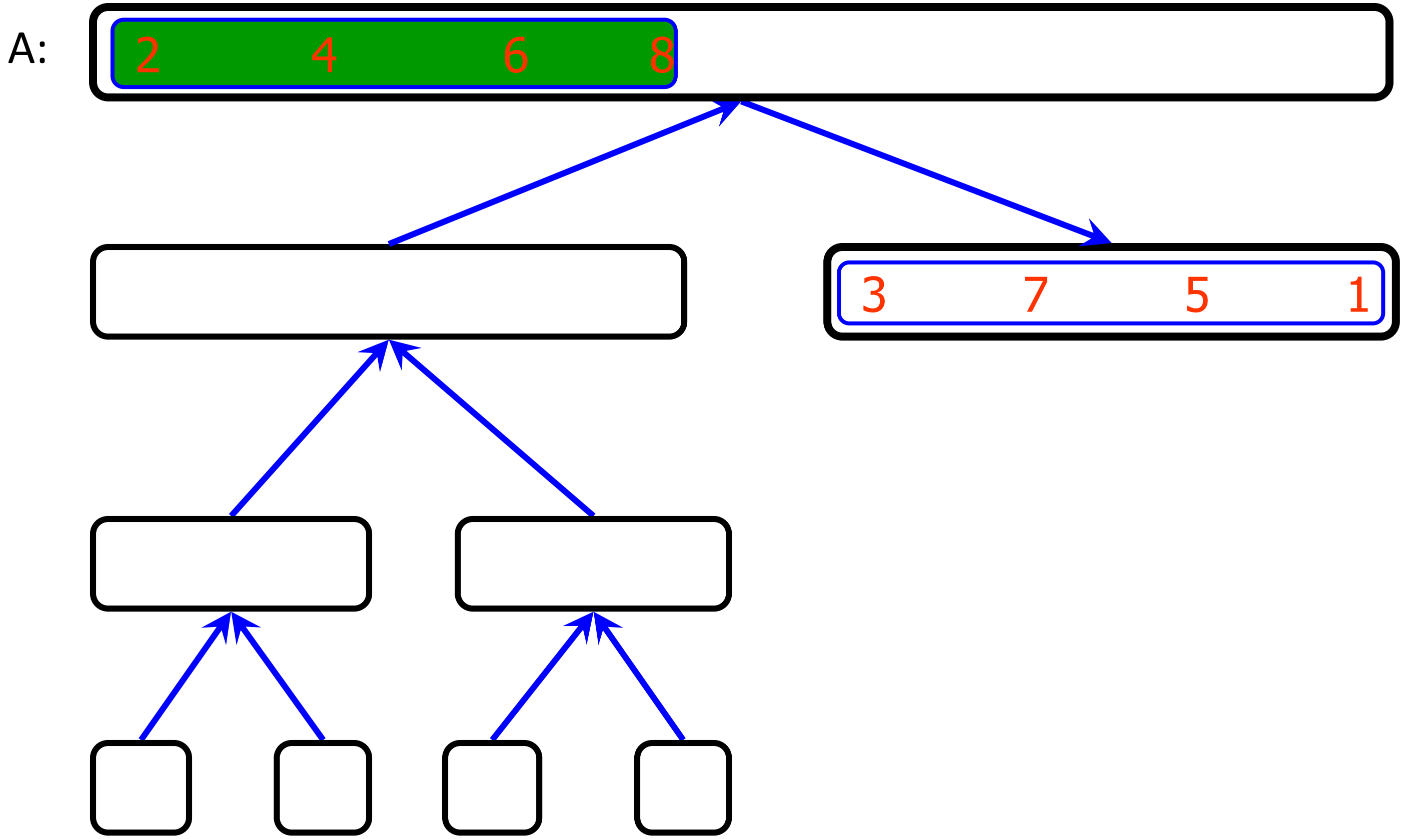
Merge(A, 0, 1, 3)



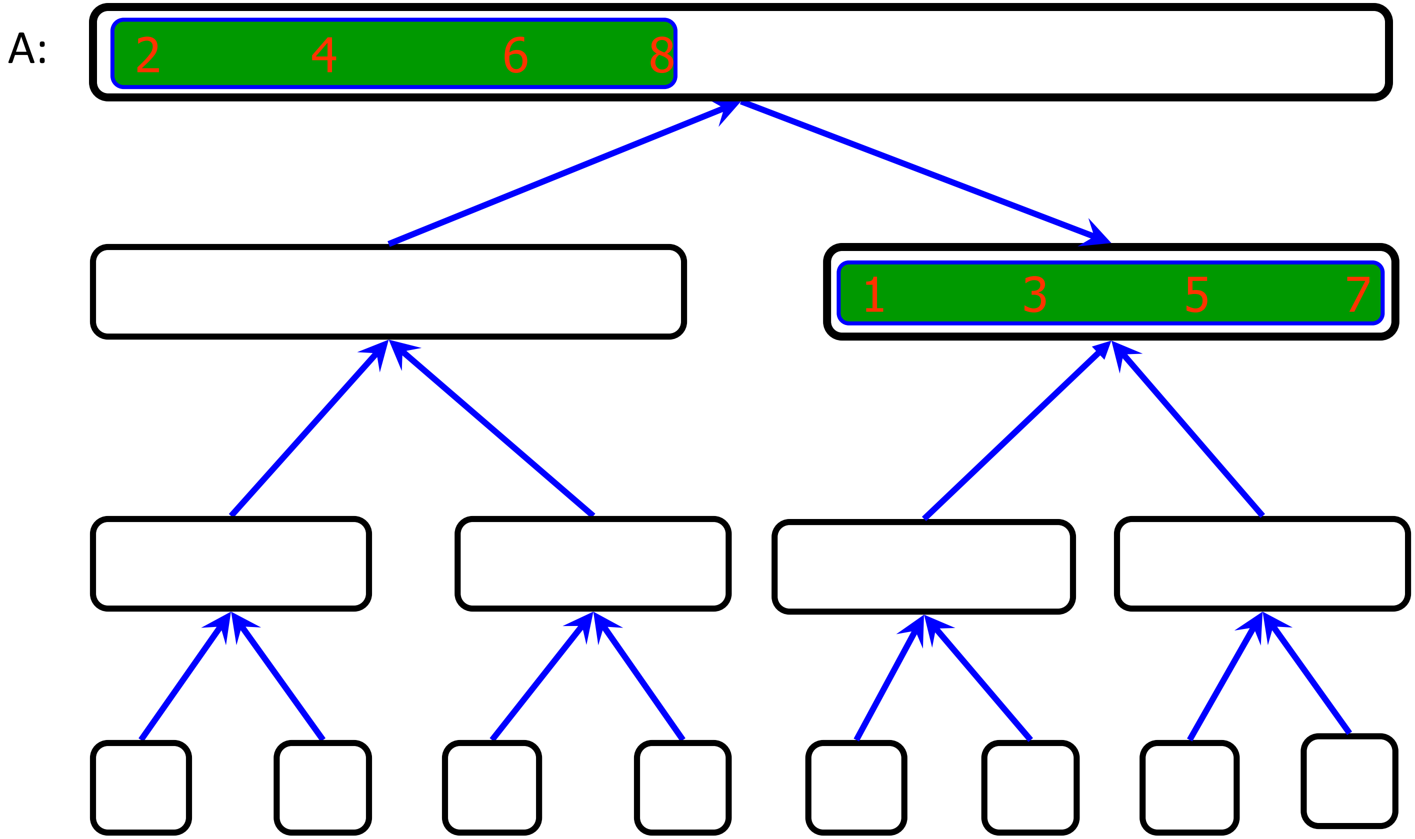
Merge-Sort(A, 0, 3), return



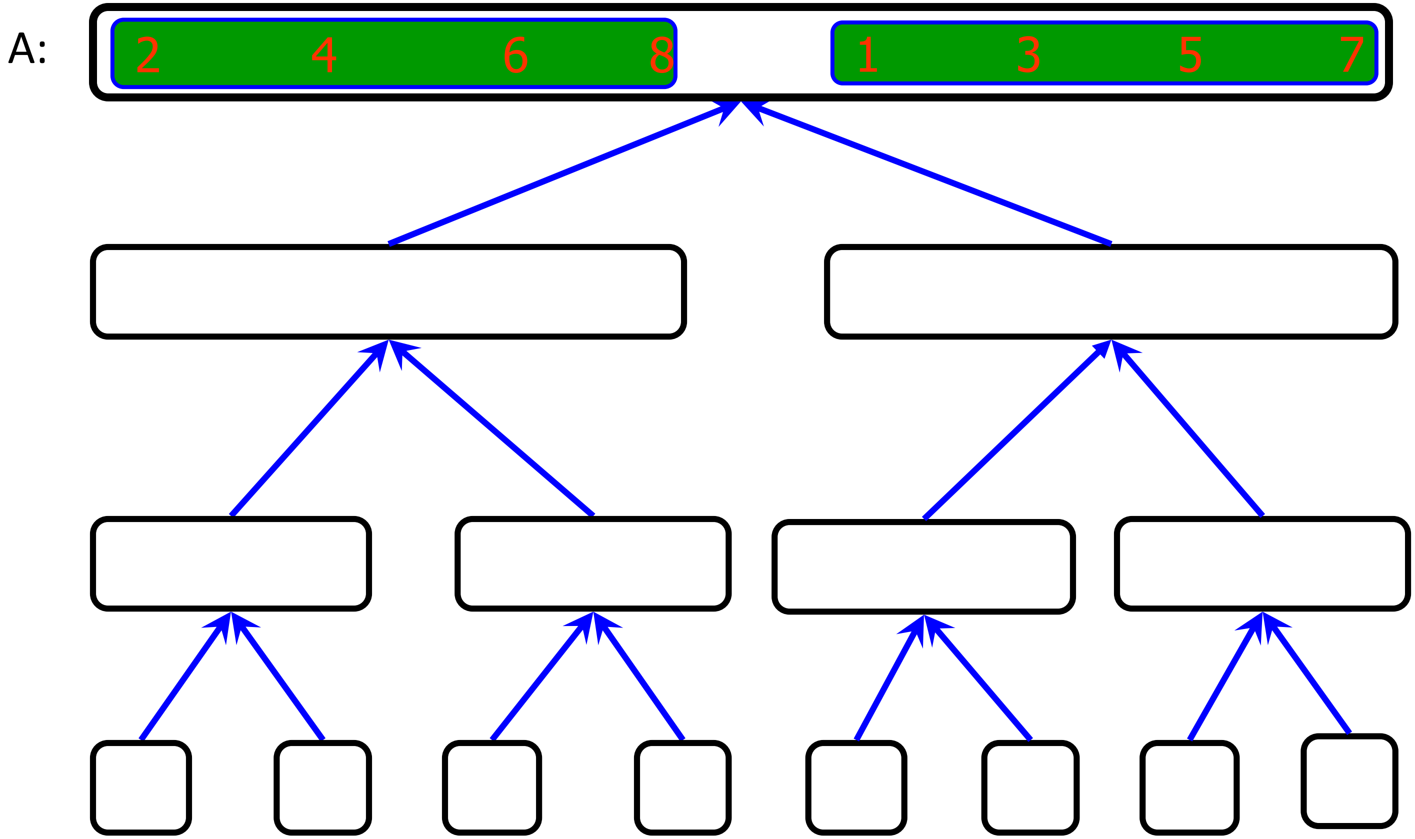
Merge-Sort(A, 4, 7)



Merge (A, 4, 5, 7)

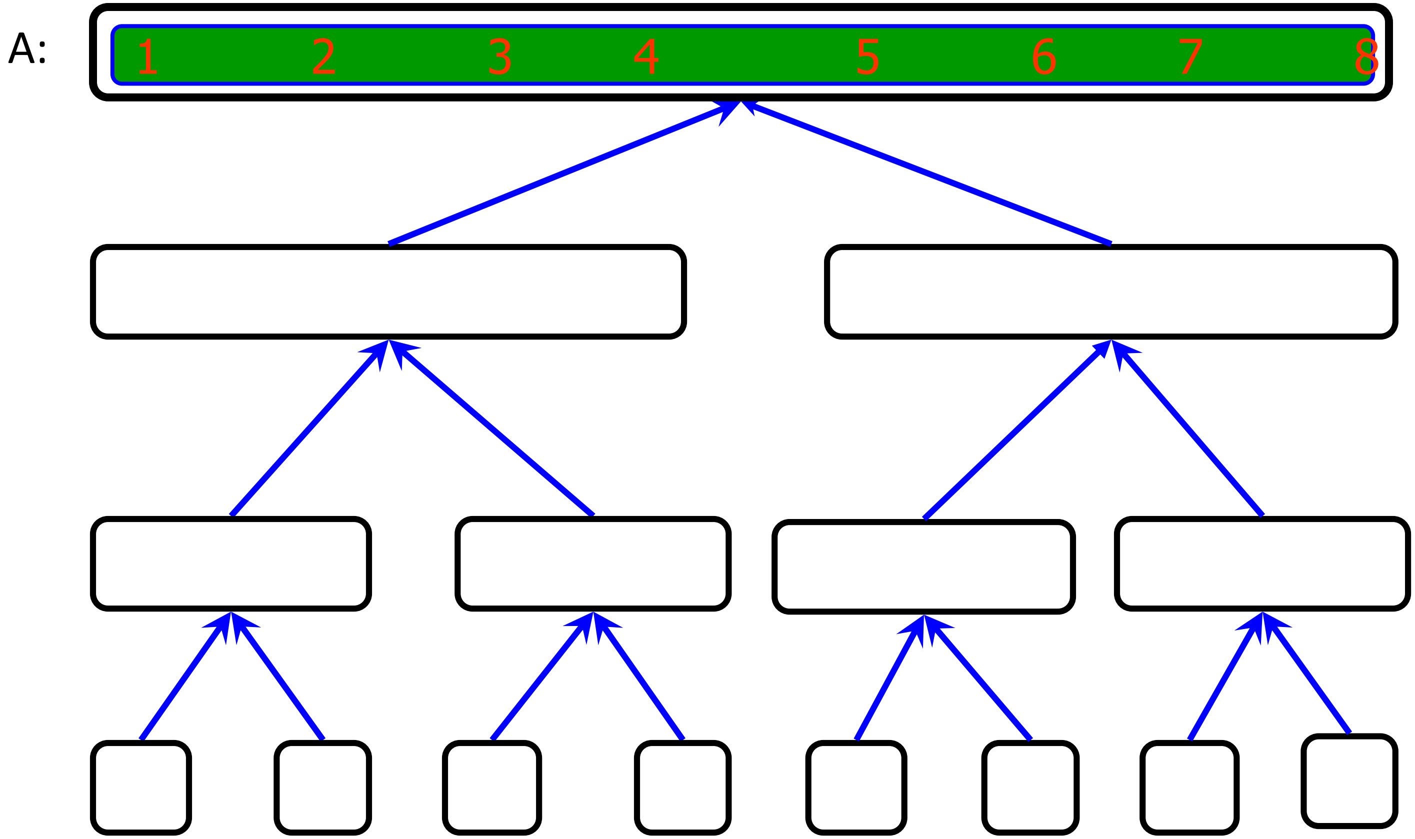


Merge-Sort(A, 4, 7), return

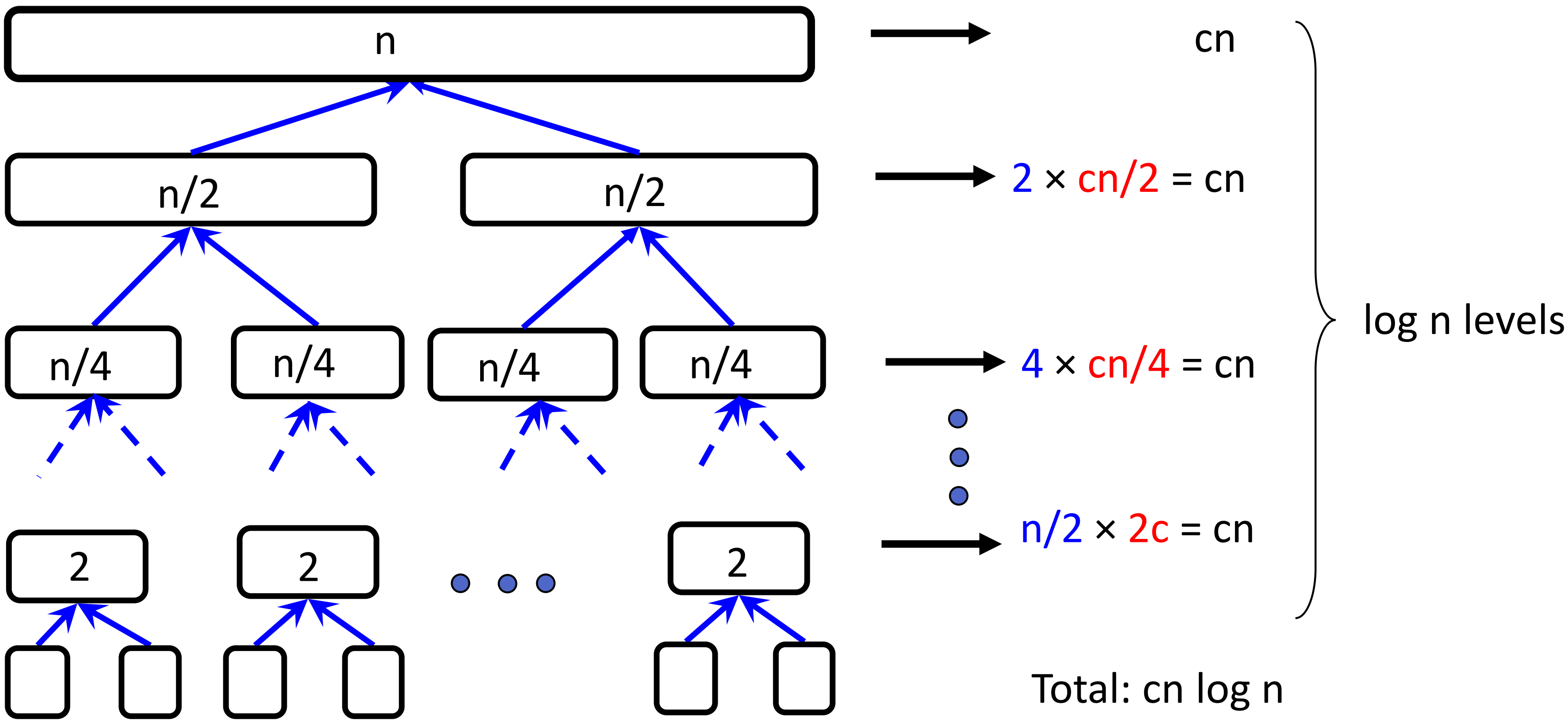


MergeSort(A,0,7) Done!

Merge(A, 0, 3, 7)



Merge-Sort Analysis



- Total running time: Θ(n.logn)
- Total Space: Θ (n)

Merge-Sort Summary

Approach: divide and conquer

Time

- Most of the work is in the merging
- Total time: $\Theta(n \log n)$

Space:

- $\Theta(n)$, more space than other sorts.

Overview

- Divide and Conquer
- Merge Sort
- **Quick Sort**

Quick Sort

- **Divide**:
  - Pick any element **p** as the **pivot**, e.g, the first element
  - Partition the remaining elements into

    FirstPart, which contains all elements < p

    SecondPart, which contains all elements ≥ p

- **Recursively sort** the FirstPart and SecondPart

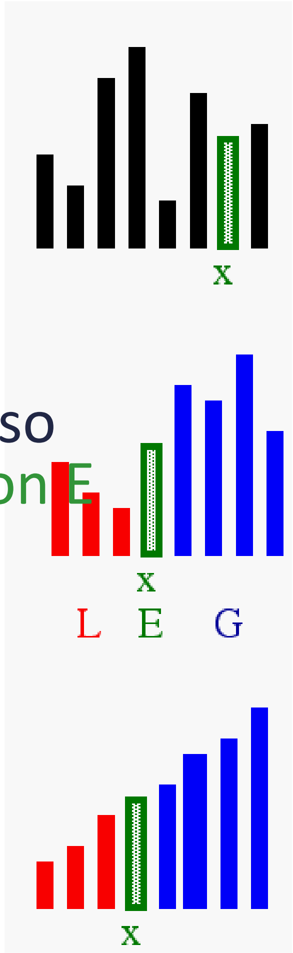
- **Combine**: no work is necessary since sorting is done in place

Idea of Quick Sort

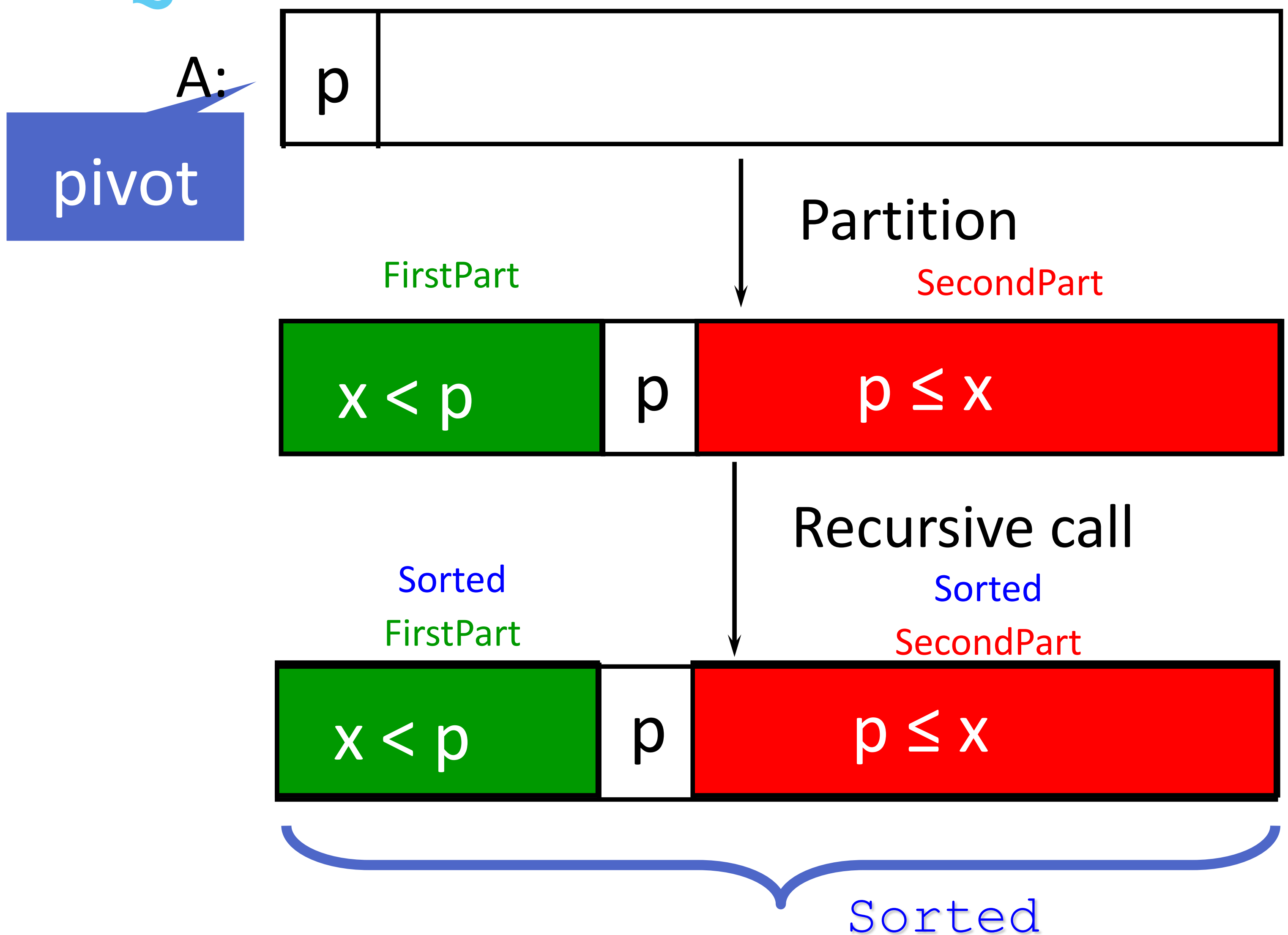
1) **Select**: pick an element

2) **Divide**: rearrange elements so that x goes to its final position E

3) **Recurse and Conquer**: recursively sort



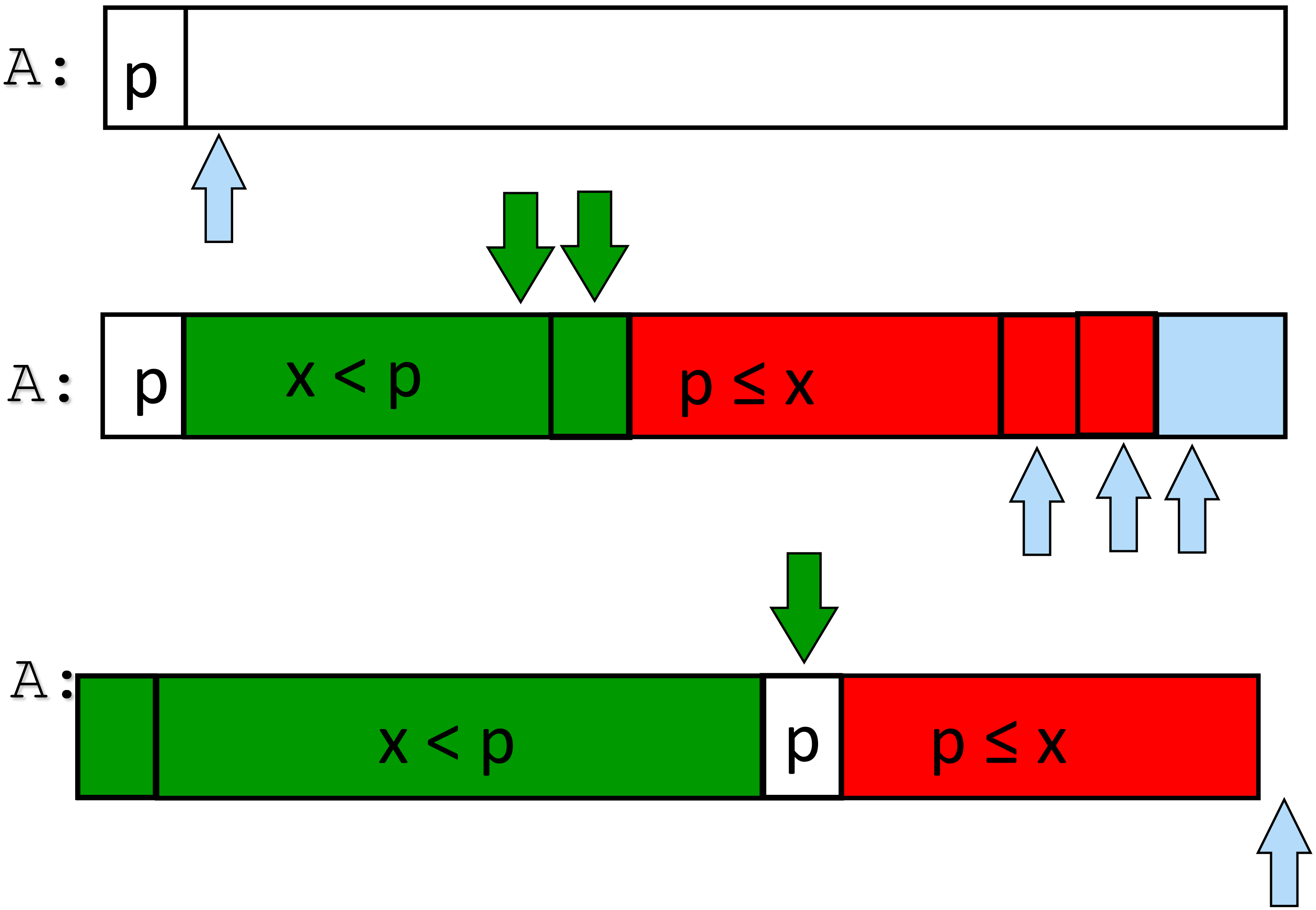
Quick Sort


A:
p
pivot
Partition
FirstPart
SecondPart
x < p
p
p ≤ x
Recursive call
Sorted
FirstPart
Sorted
SecondPart
x < p
p
p ≤ x
Sorted

Quick Sort

```
QuickSort(A, left, right)
 if    left >= right  return
  else
        middle=Partition(A, left, right)
        QuickSort(A, left, middle-1 )
        QuickSort(A, middle+1, right)
  end if
```

Partition

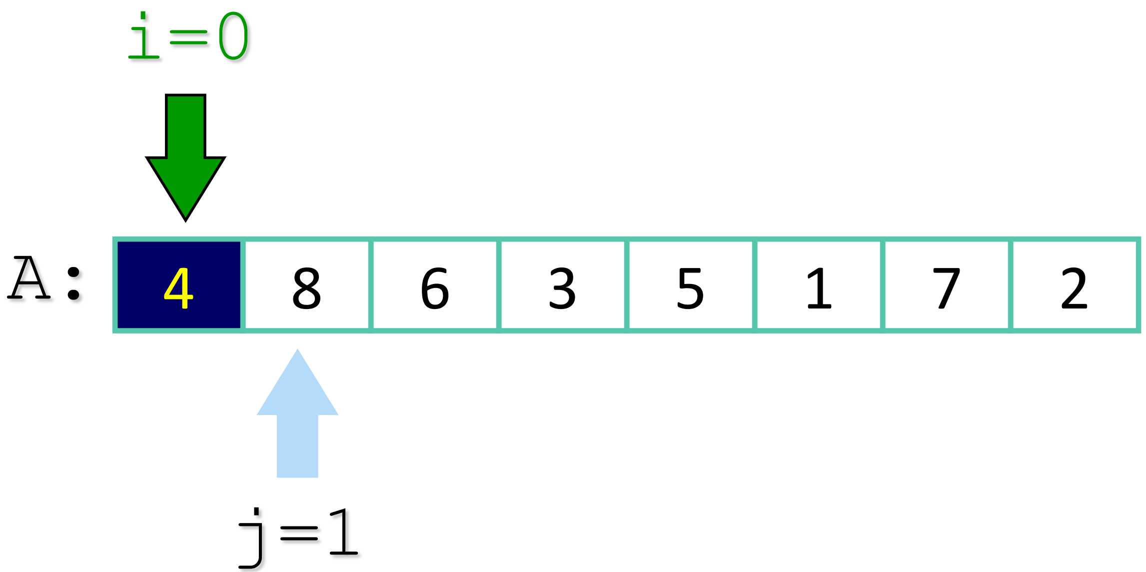


Partition Example

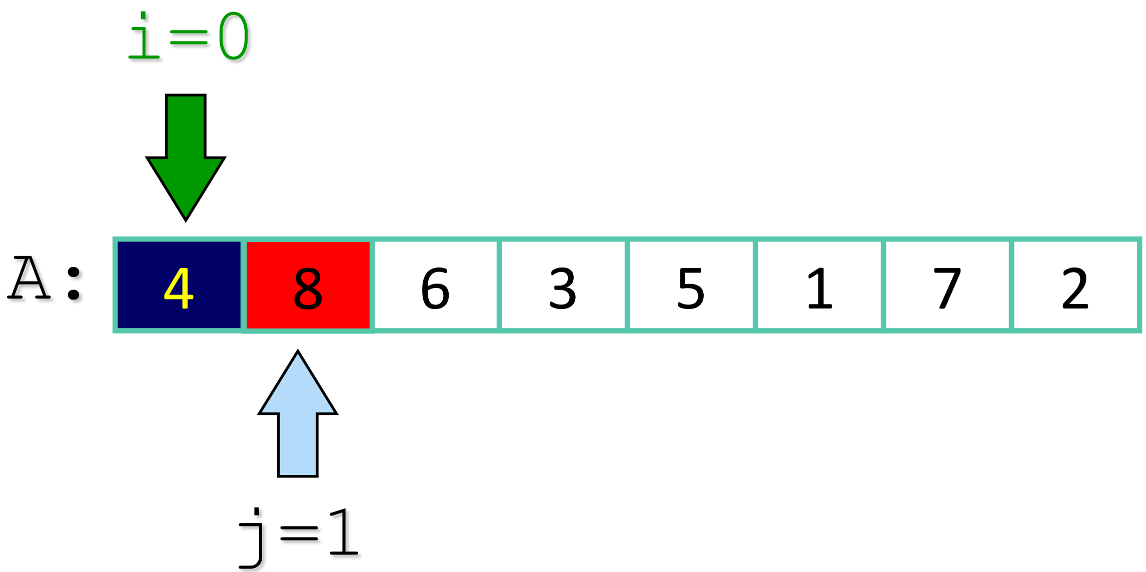
A:

4	8	6	3	5	1	7	2

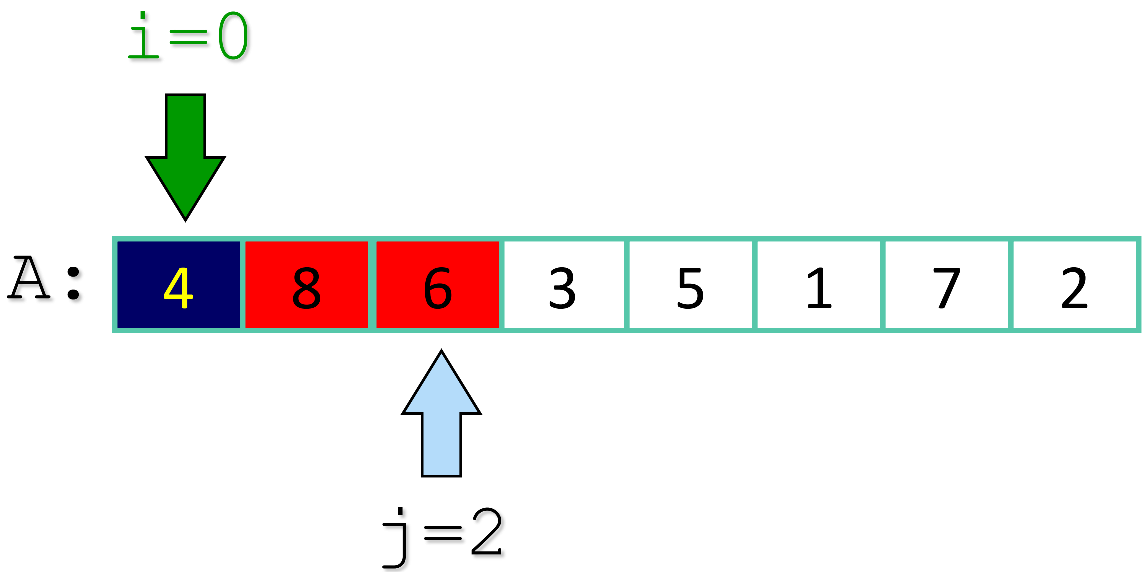
Partition Example



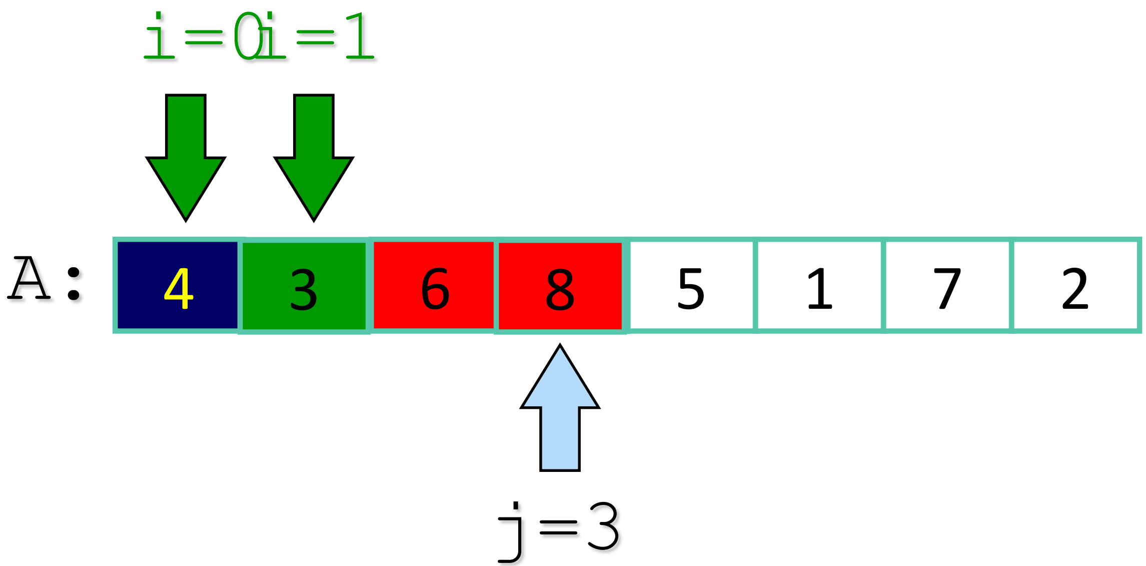
Partition Example



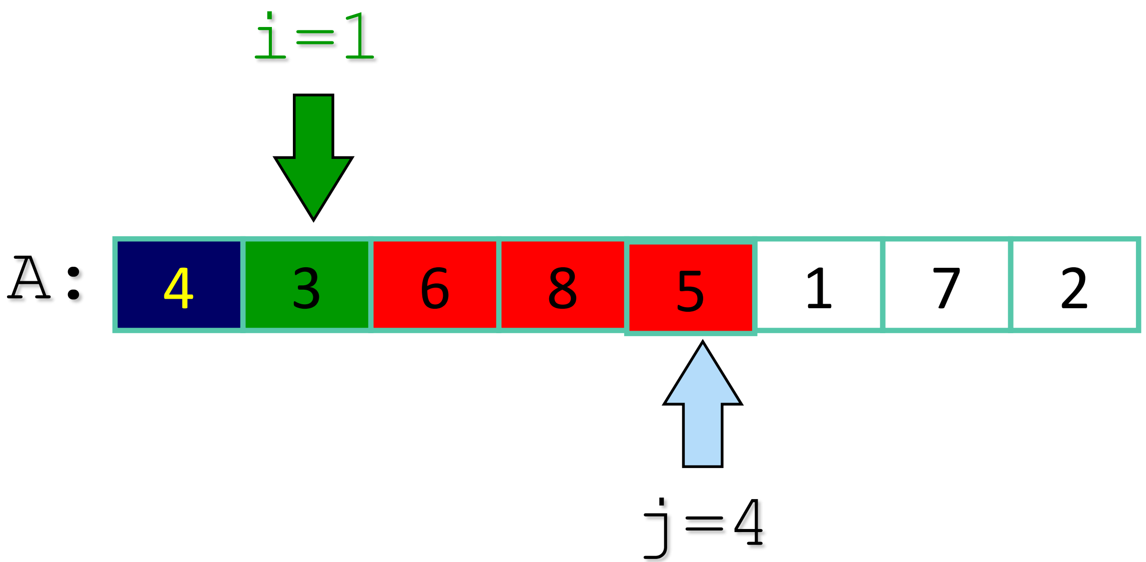
Partition Example



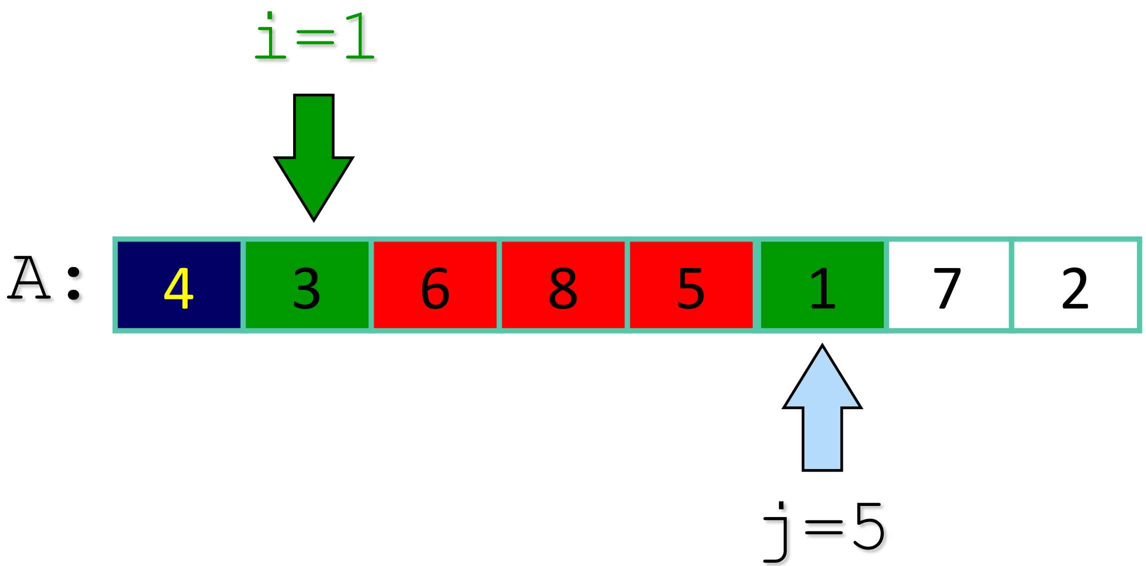
Partition Example



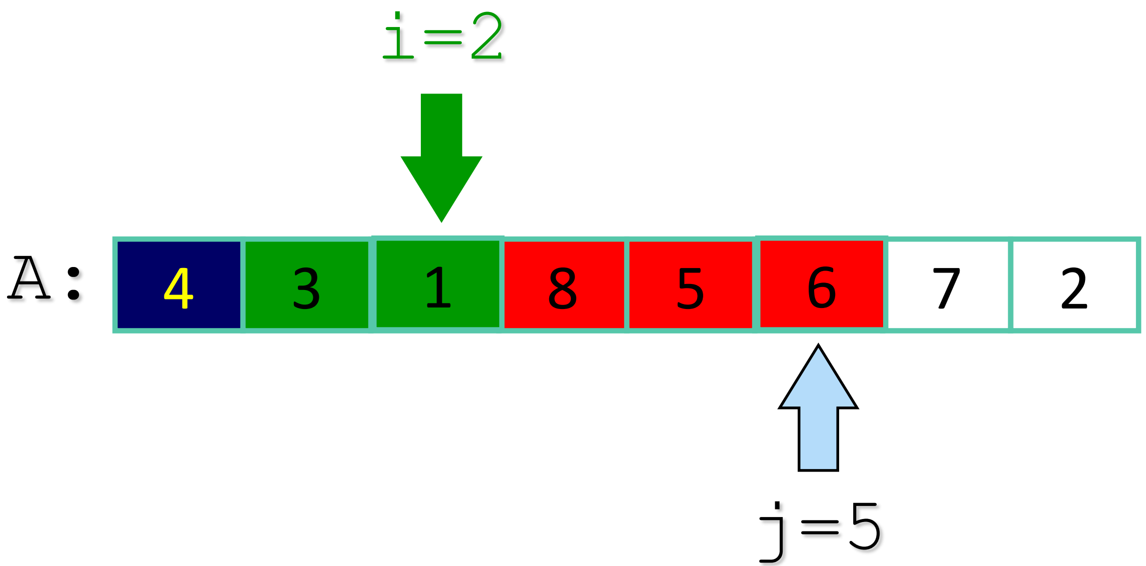
Partition Example



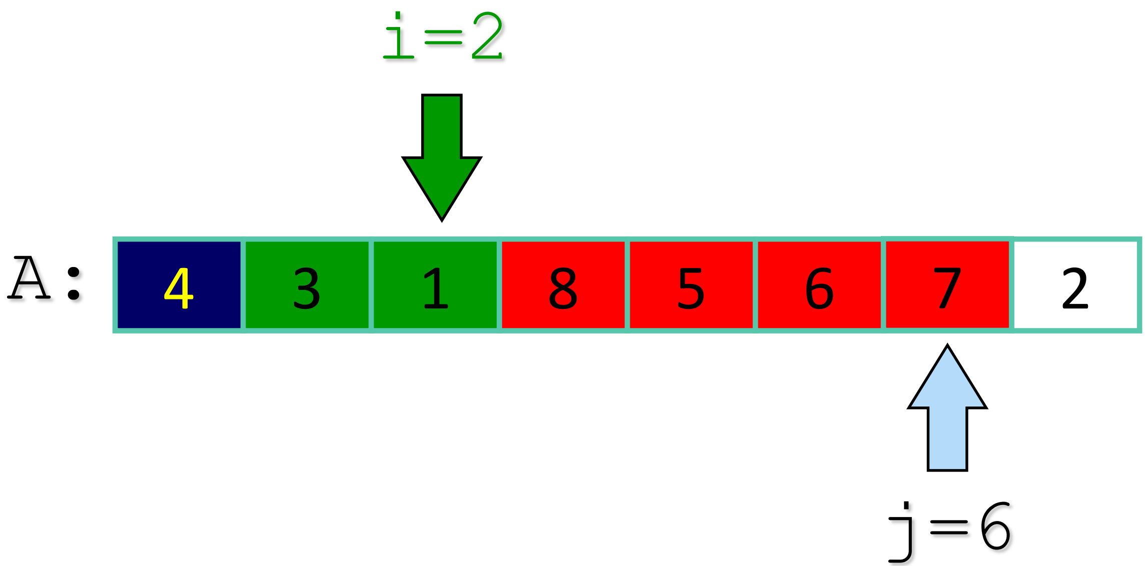
Partition Example



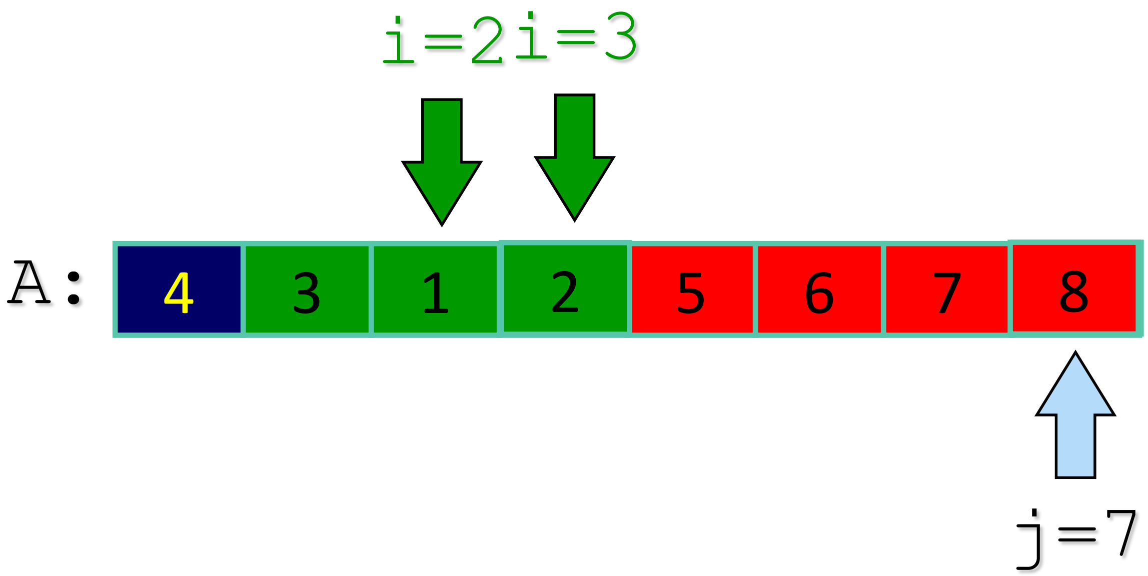
Partition Example



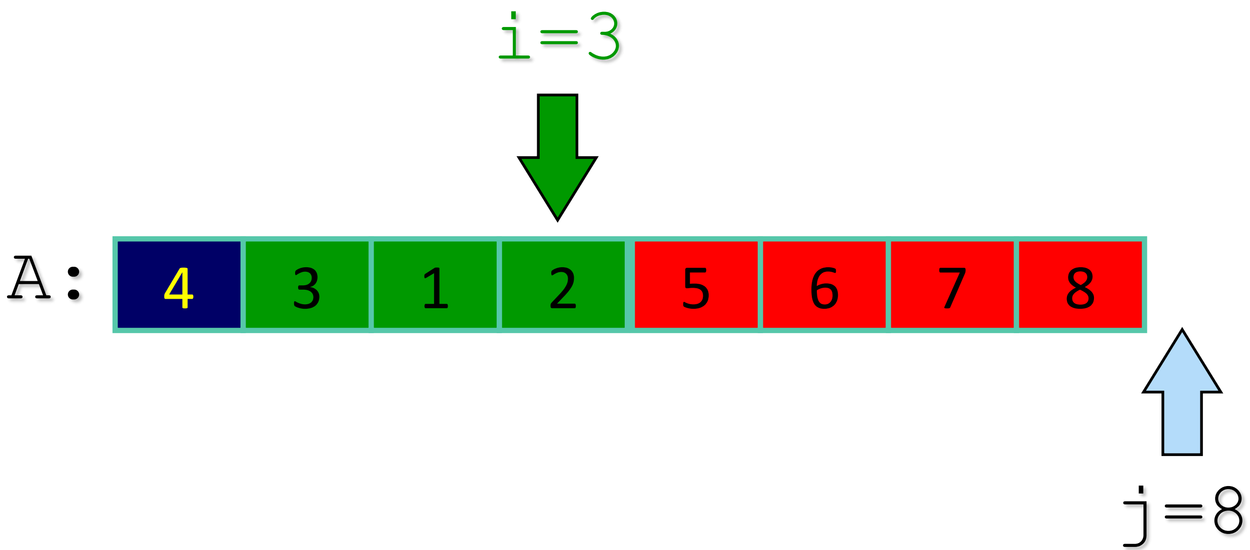
Partition Example



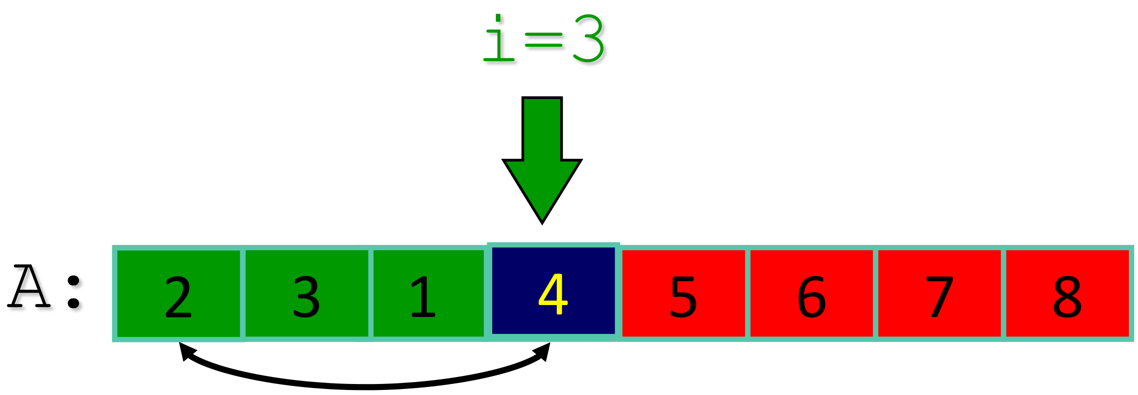
Partition Example



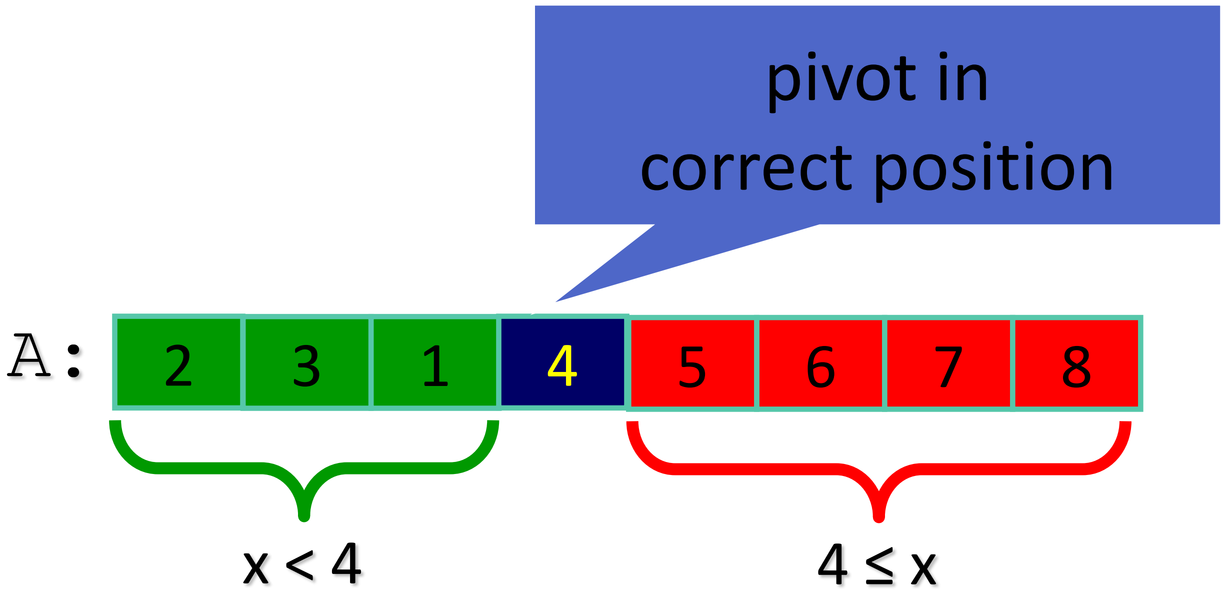
Partition Example



Partition Example



Partition Example



```
Partition(A, left, right)
    x ← A[left]
    i ← left
    for j ← left+1 to right
            if A[j] < x then
                    i ← i + 1
                    swap(A[i], A[j])
            end if
    end for j
    swap(A[i], A[left])
    return i
```

n = right – left +1
Time: cn for some constant c
Space: constant

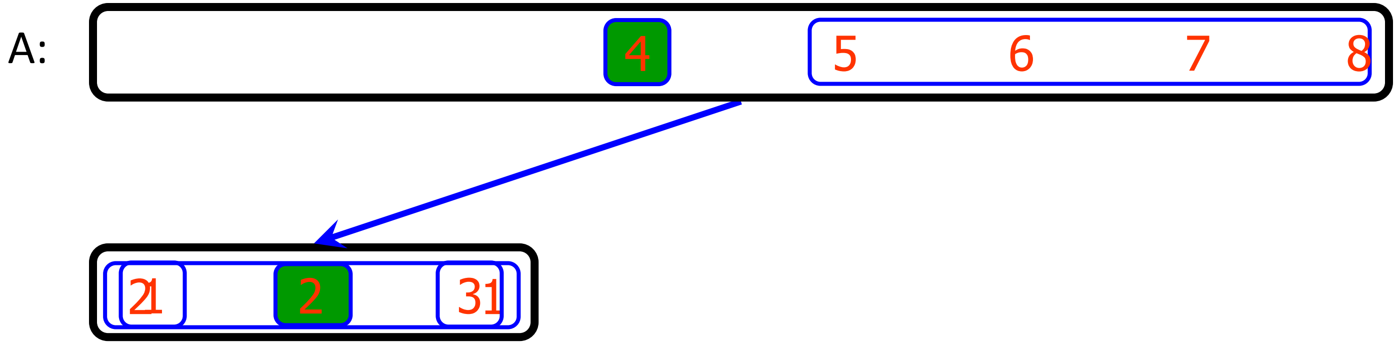
Quick-Sort(A, 0, 7)

Partition

A:								
	2 4	8 3	6 1	4	5 5	1 6	7 7	8 2

Quick-Sort(A, 0, 7)

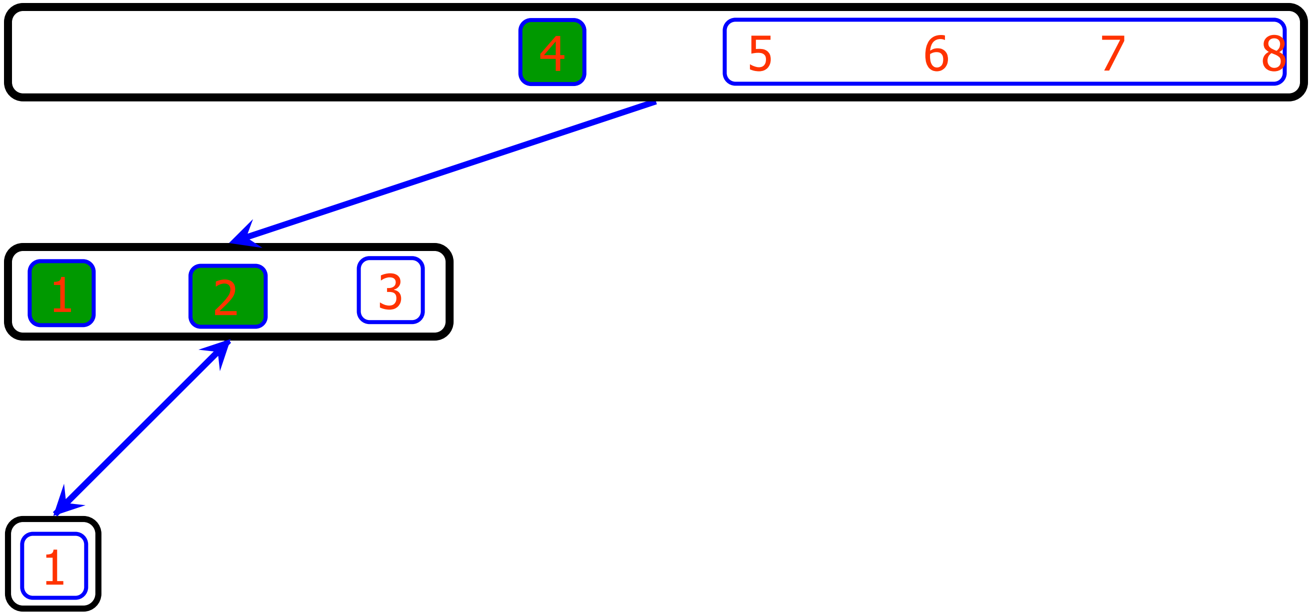
Quick-Sort(A, 0, 2) , partition



Quick-Sort(A, 0, 7)

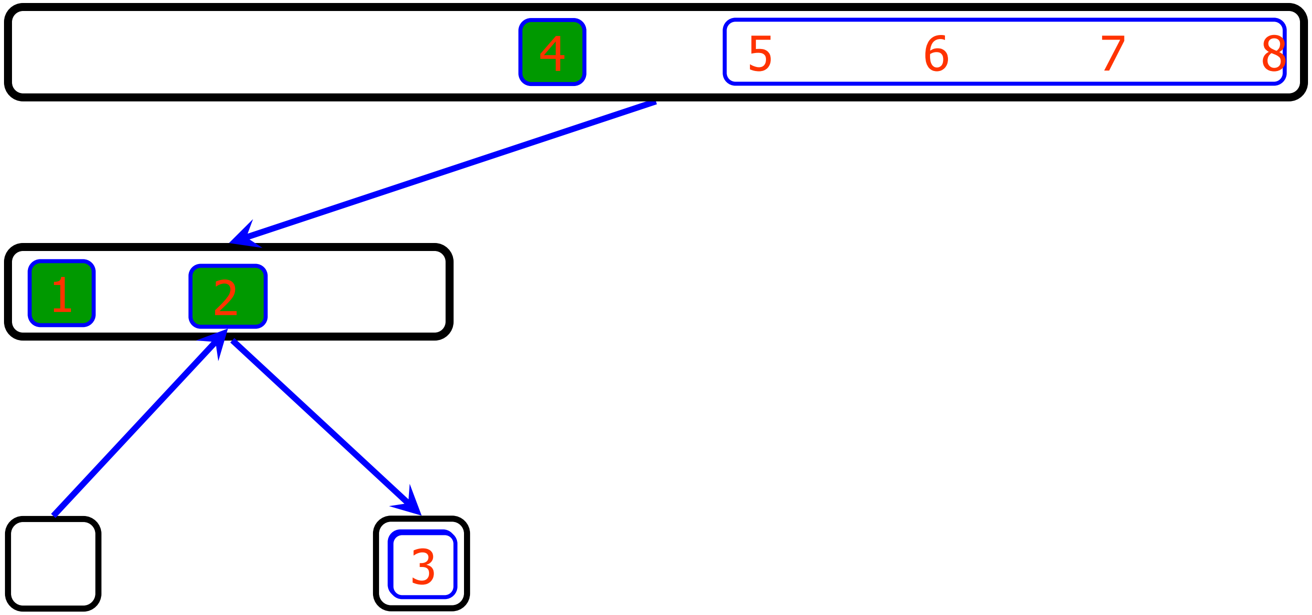
Quick-Sort(A, 0, 0) , base case

return



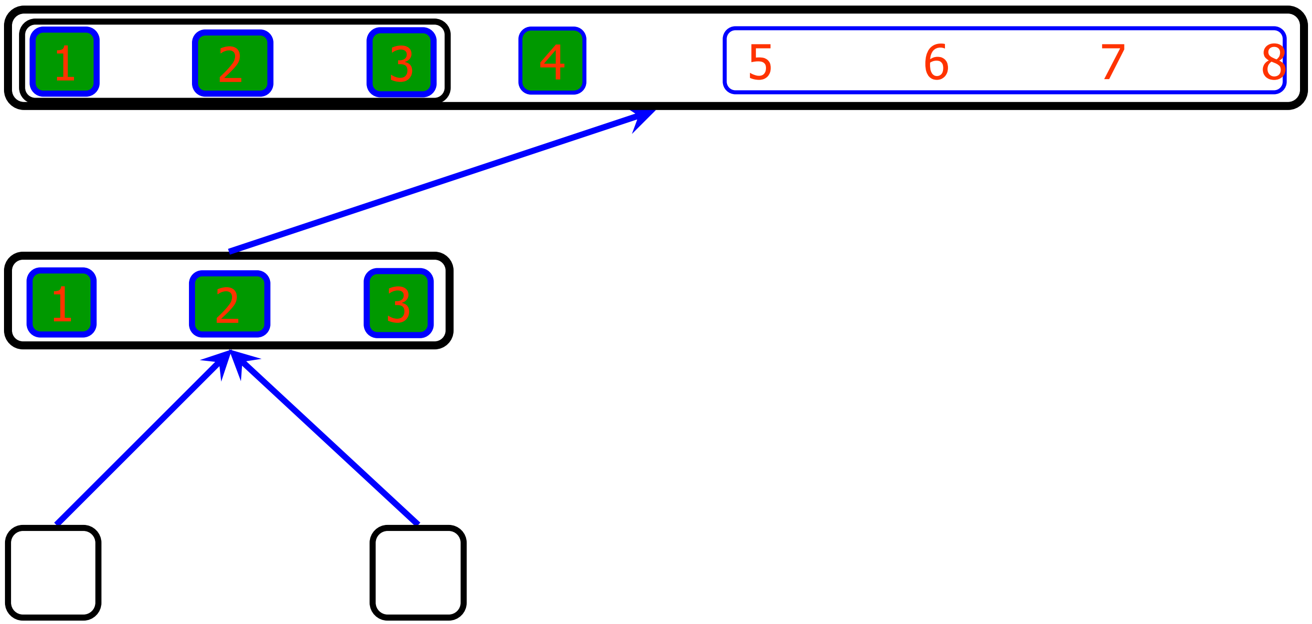
Quick-Sort(A, 0, 7)

Quick-Sort(A, 1, 1) , base case



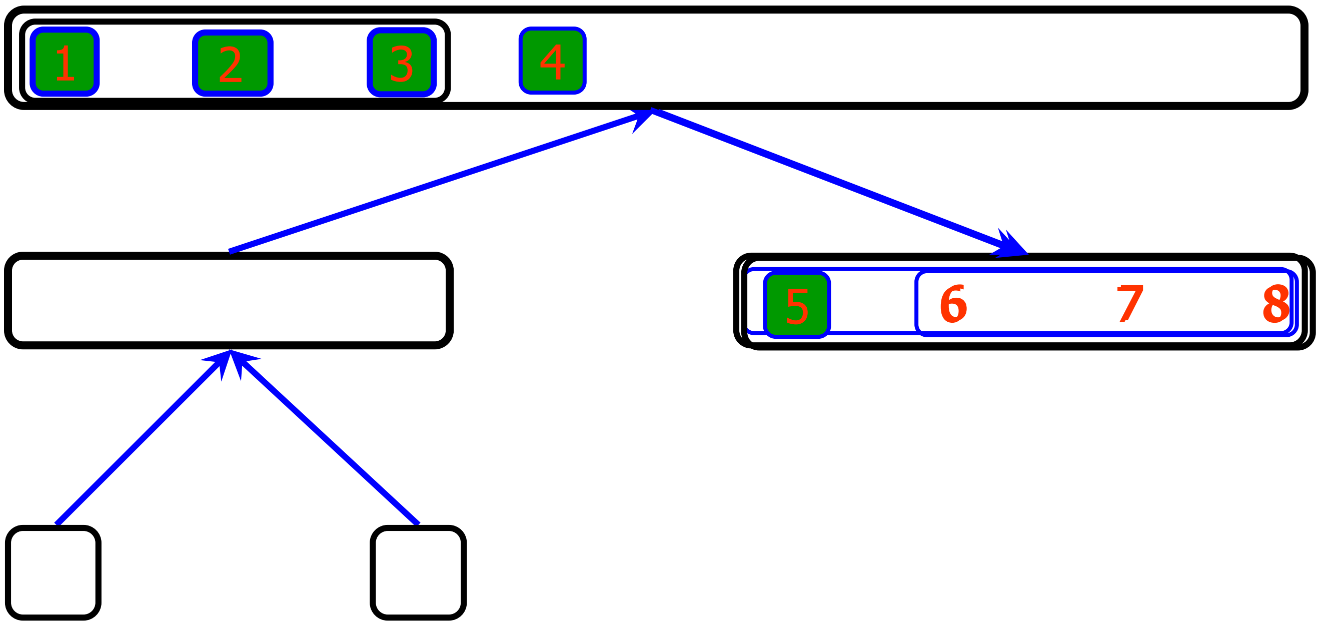
Quick-Sort(A, 0, 7)

Quick-Sort(A, 0, 2), return



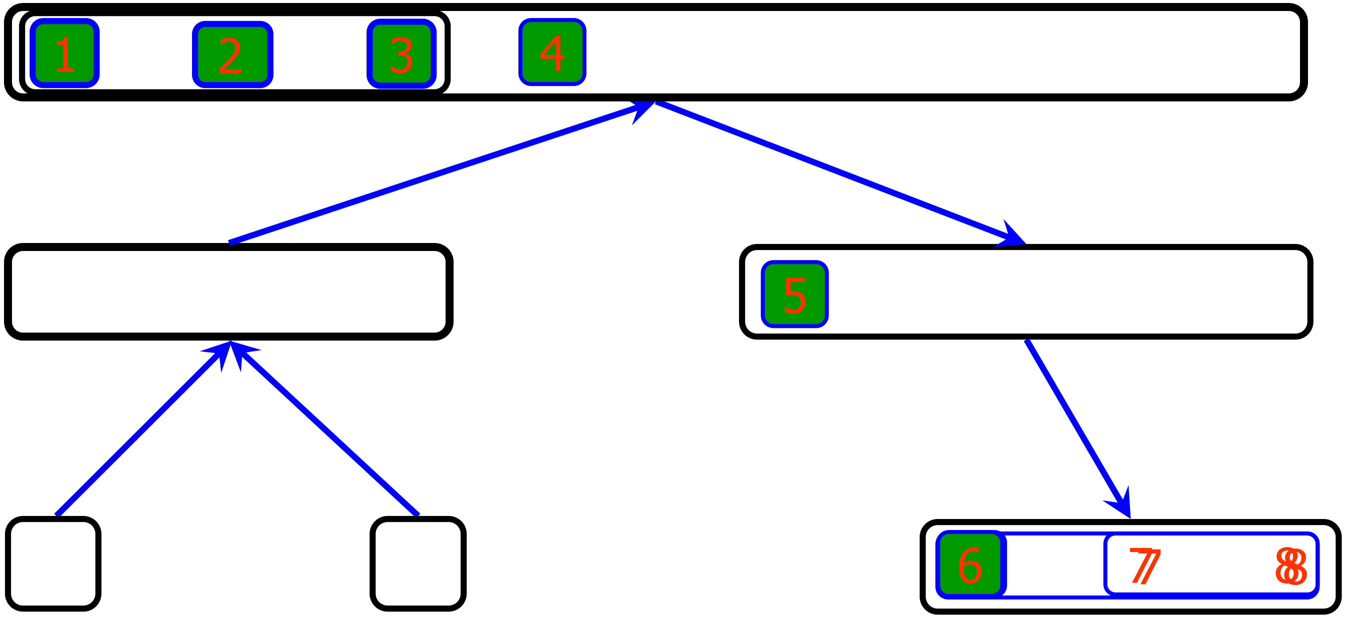
Quick-Sort(A, 0, 7)

Quick-Sort(A, 4, 7) , partition



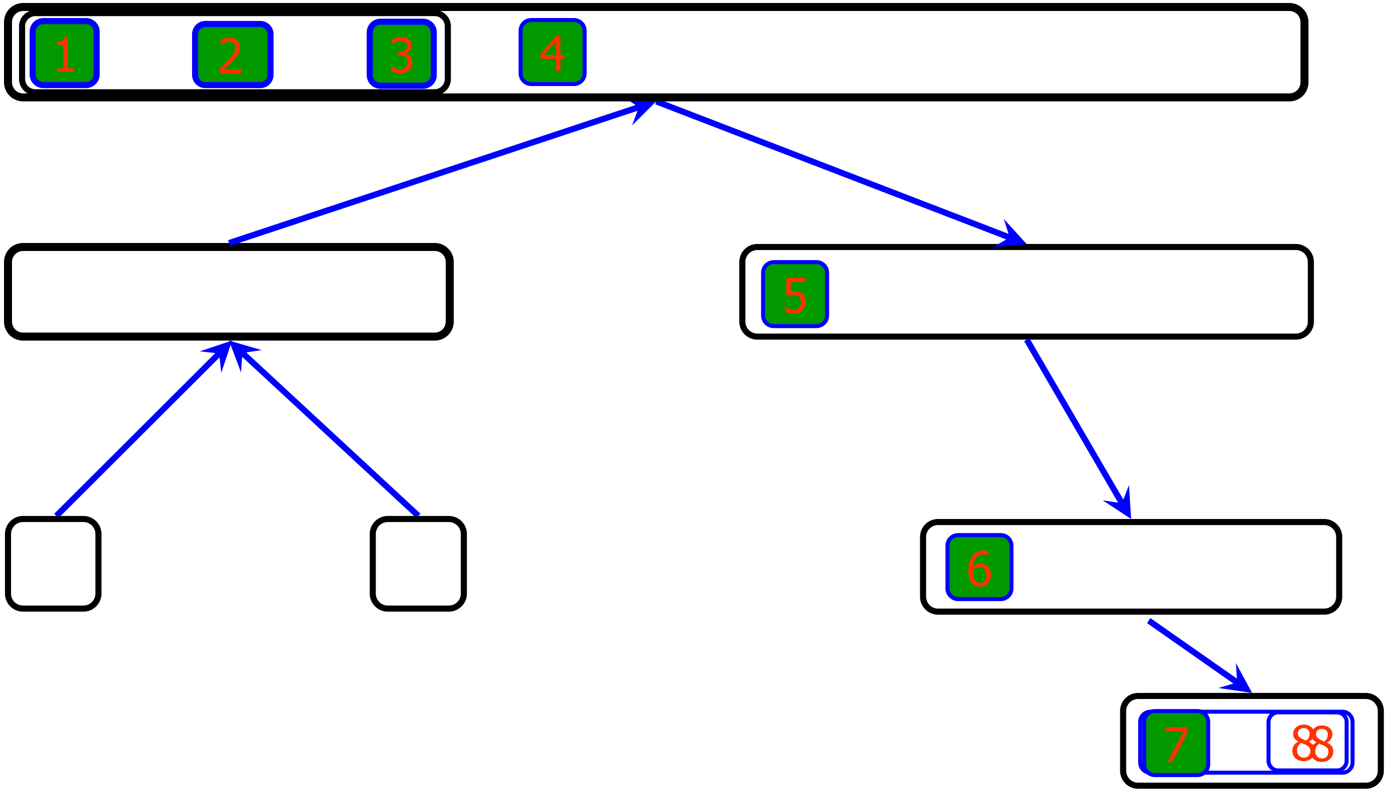
Quick-Sort(A, 0, 7)

Quick-Sort(A, 5, 7) , partition



Quick-Sort(A, 0, 7)

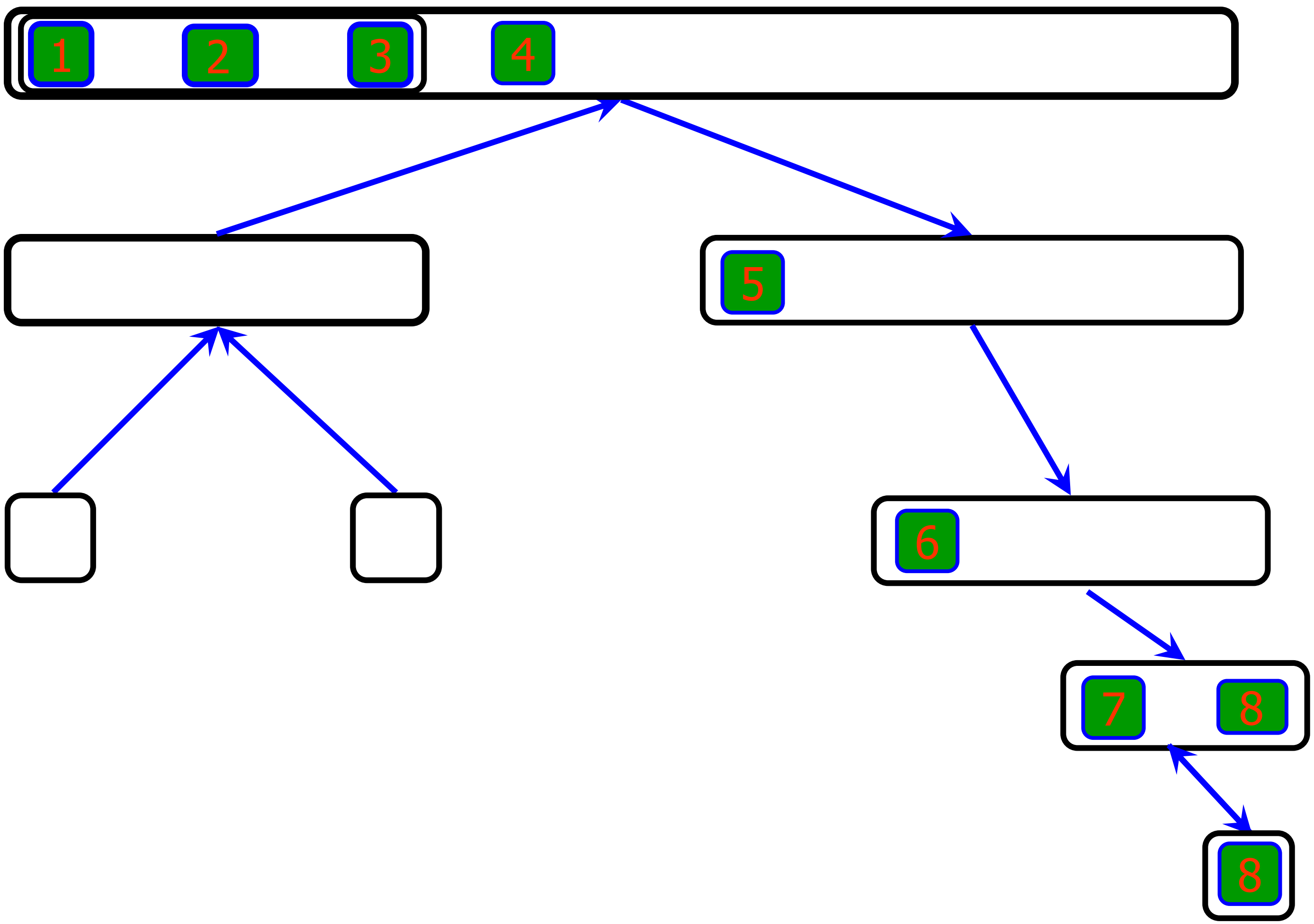
Quick-Sort(A, 6, 7) , partition



Quick-Sort(A, 0, 7)

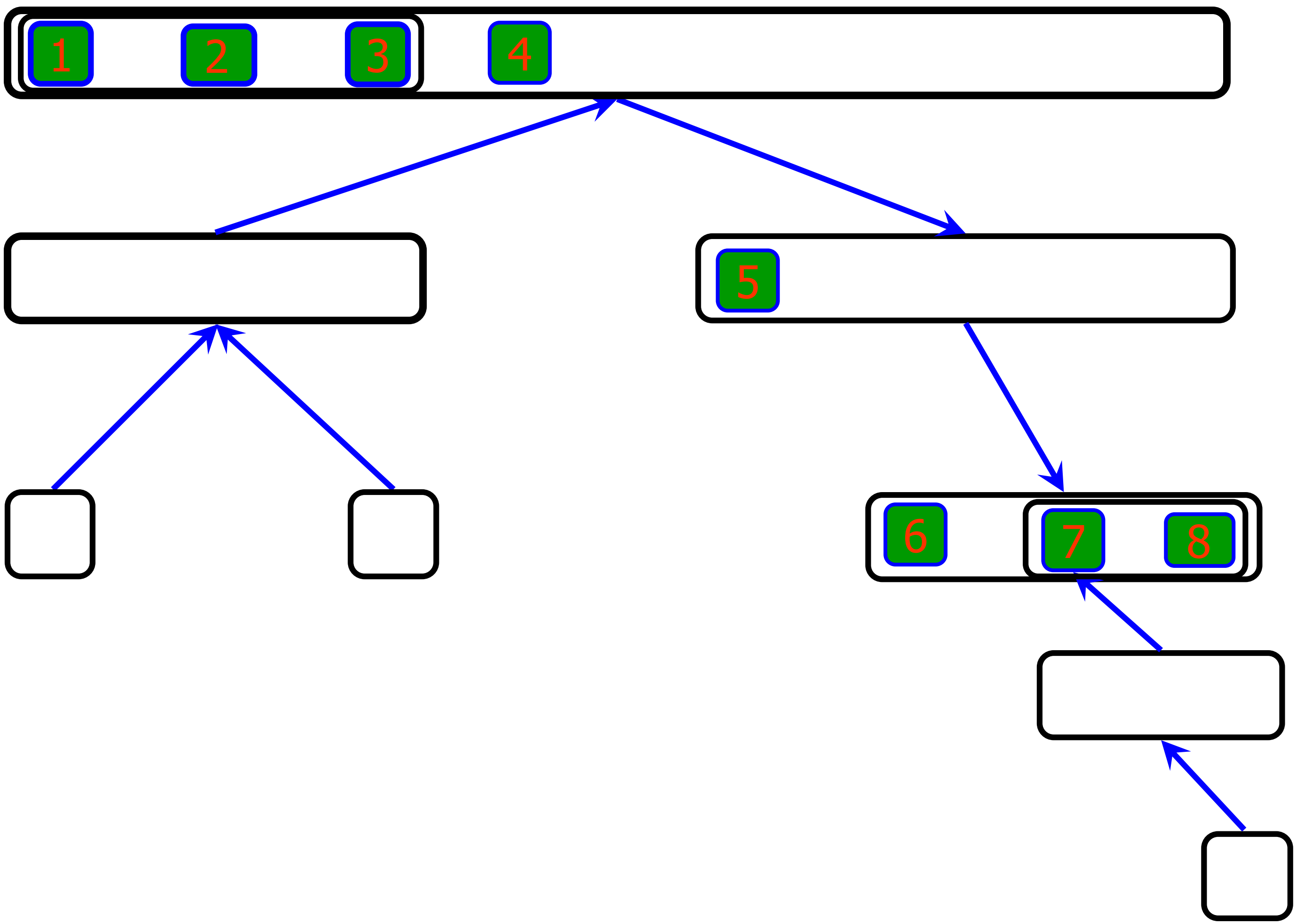
Quick-Sort(A, 7, 7) , base case

return



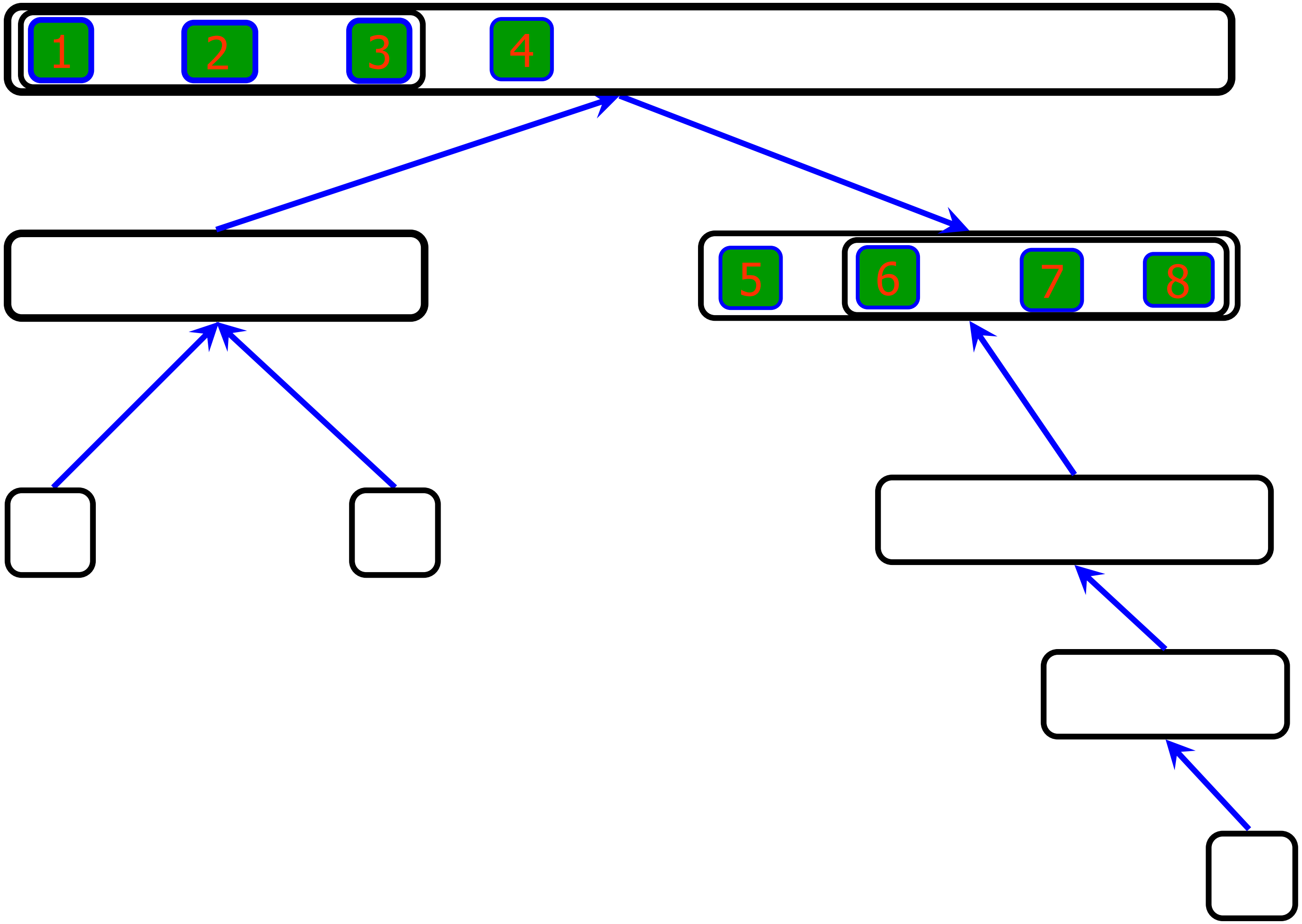
Quick-Sort(A, 0, 7)

Quick-Sort(A, 6, 7) , return



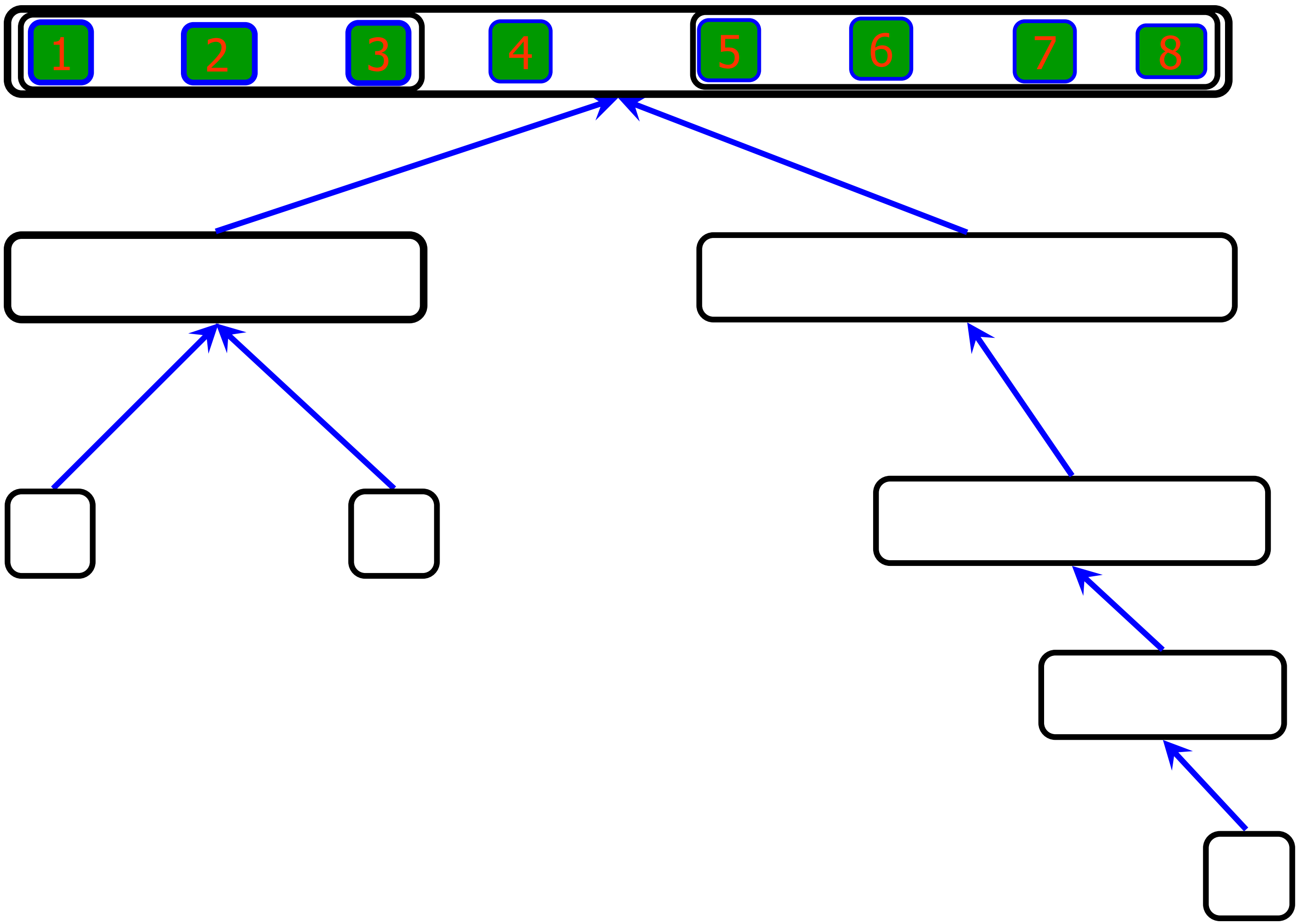
Quick-Sort(A, 0, 7)

Quick-Sort(A, 5, 7) , return



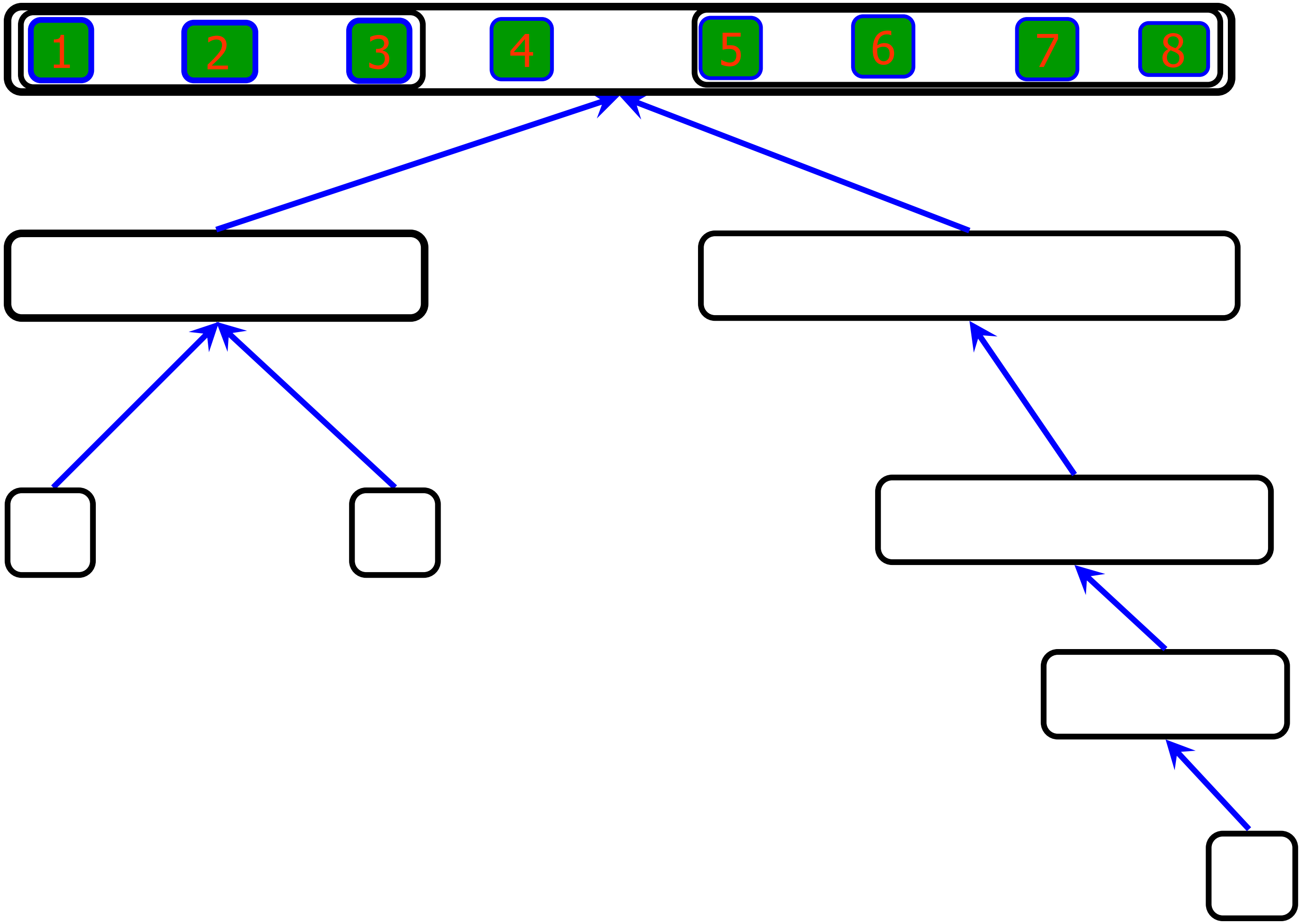
Quick-Sort(A, 0, 7)

Quick-Sort(A, 4, 7) , return



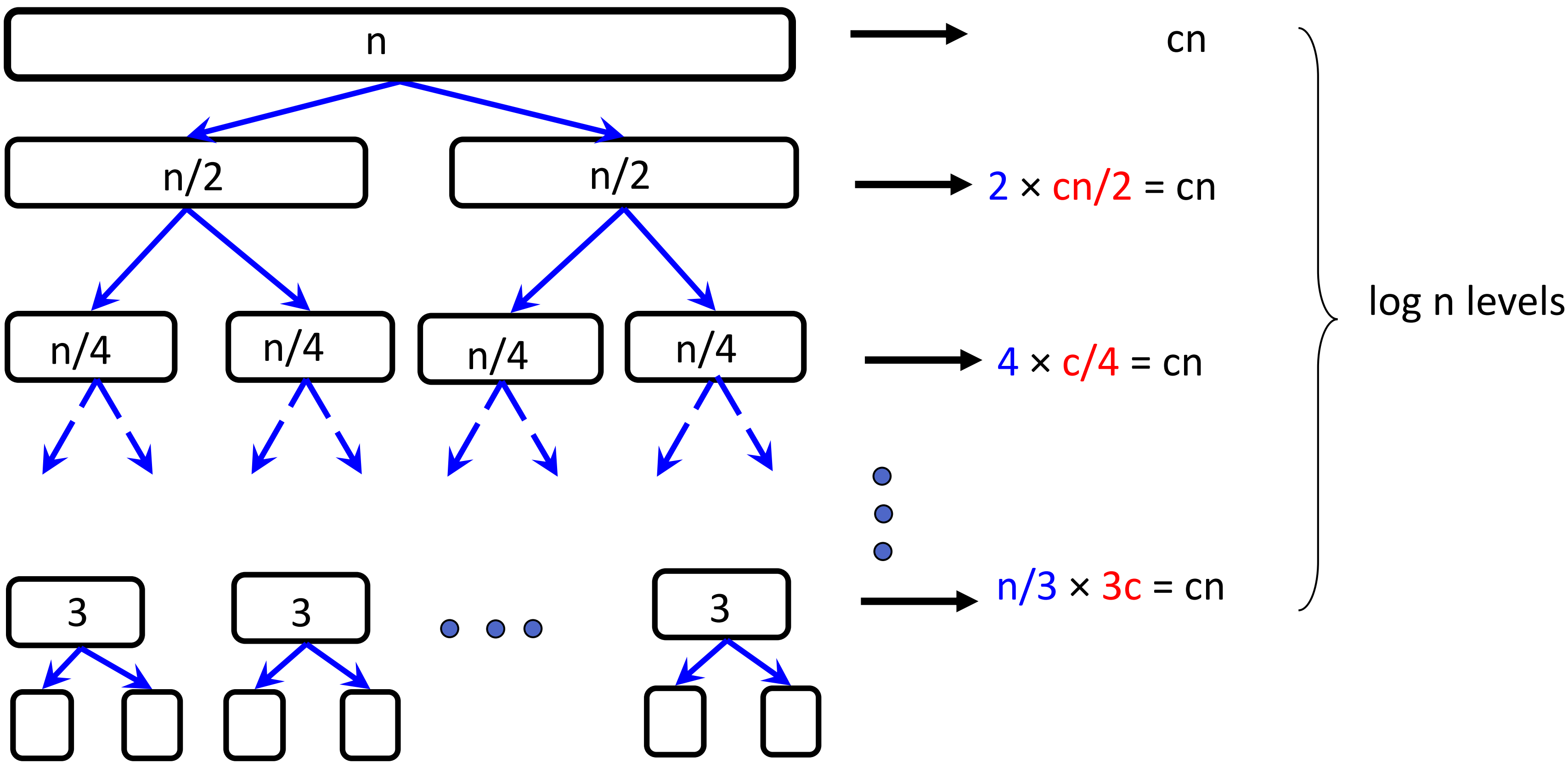
Quick-Sort(A, 0, 7)

Quick-Sort(A, 0, 7) , done!



Quick-Sort: Best Case

- Balanced Partitions

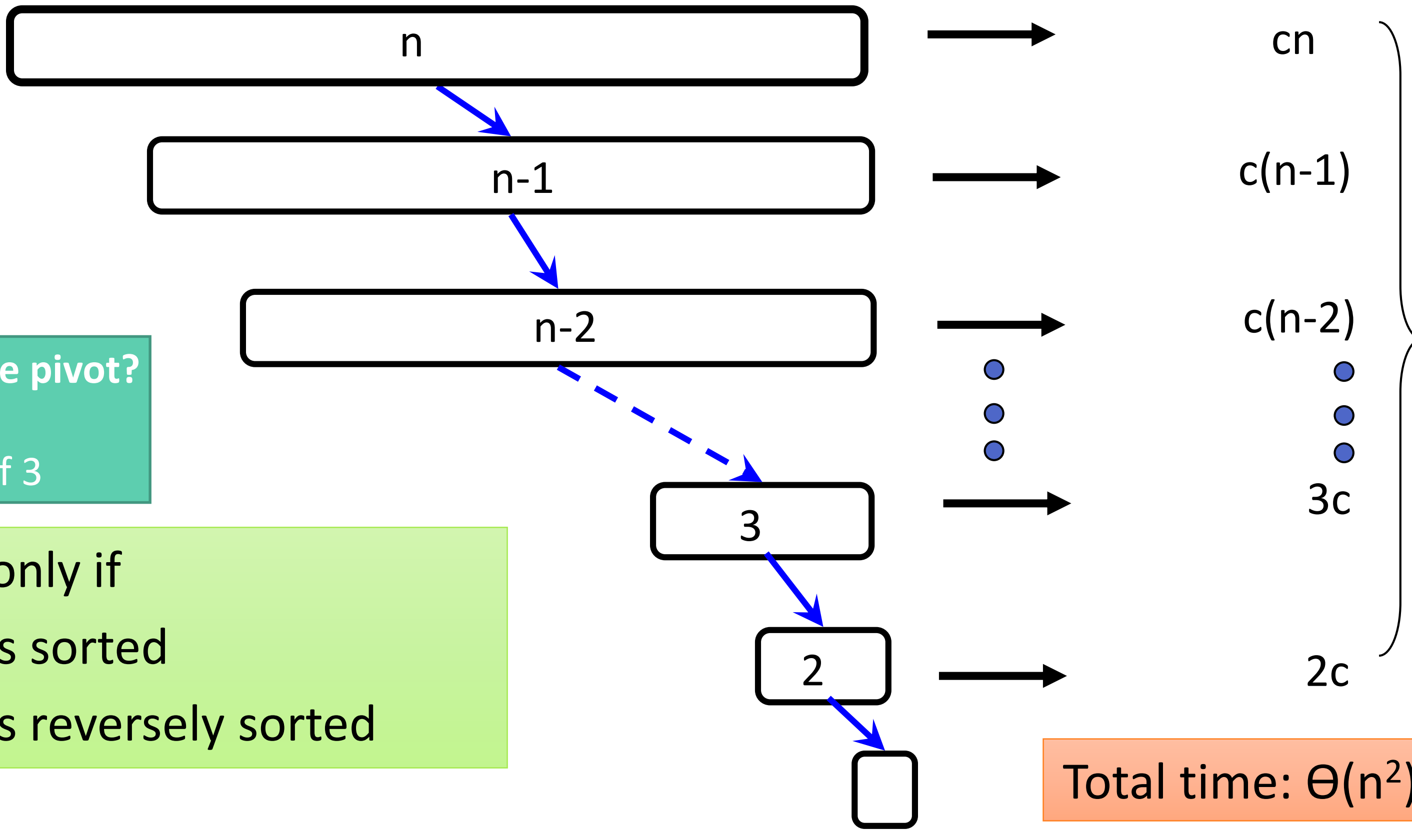


Total time: Θ(n.log n)

https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Quick-Sort: Worst Case

- Unbalanced Partition



How to choose the pivot?

- Random
- Median of 3

Happens only if

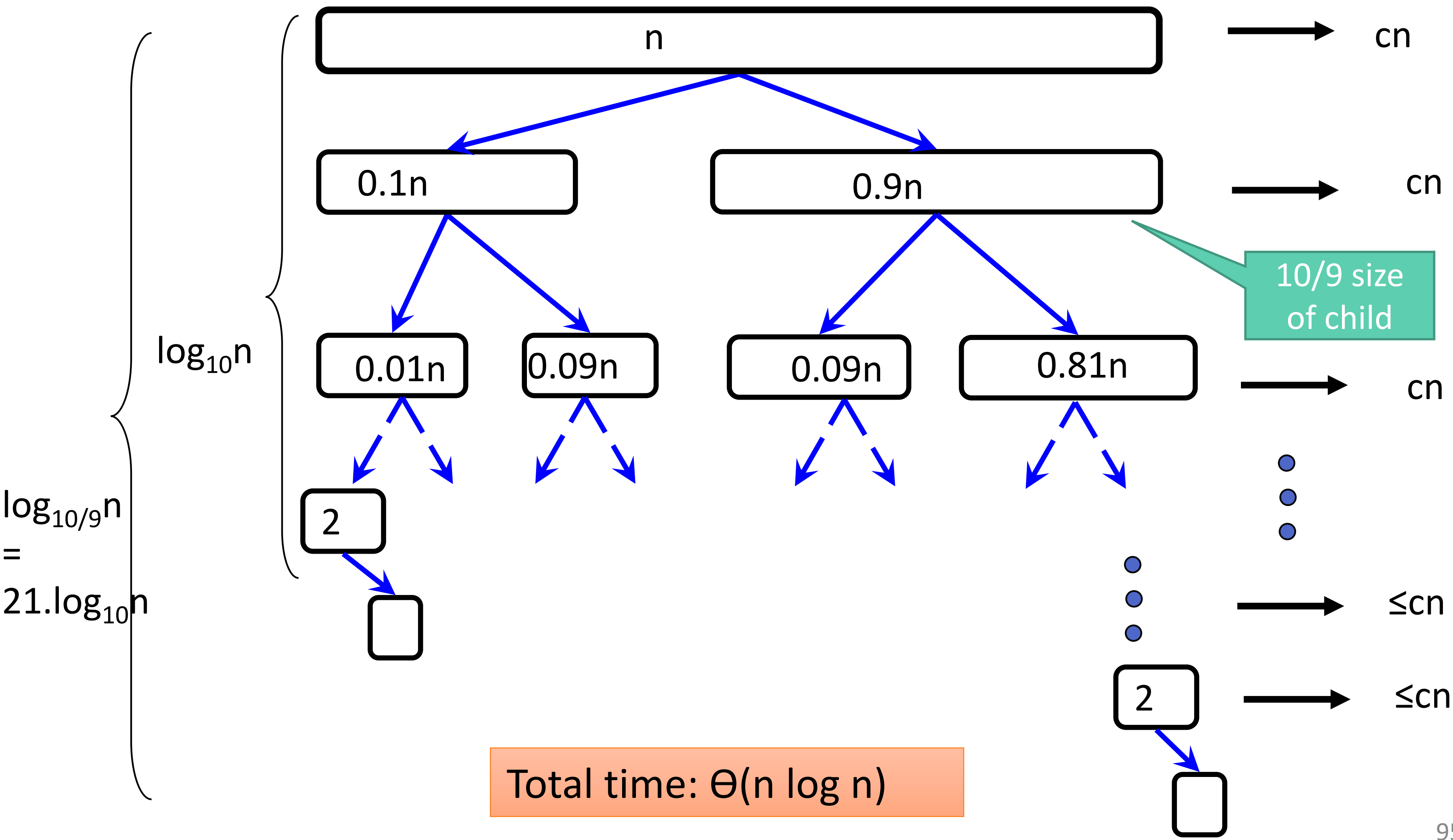
- input is sorted
- input is reversely sorted

Total time: $\Theta(n^2)$

https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Quick-Sort: an Average Case

- Suppose the split is 1/10 : 9/10



Total time: Θ(n log n)

https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Quick-Sort Summary

- Time
  - Most of the work done in partitioning.
  - Average case takes **Θ(n log(n))** time.
  - Worst case takes **$\Theta(n^2)$** time

Space
  - Sorts in-place, i.e., does not require additional space

Summary

- Divide and Conquer
- Merge-Sort
  - Most of the work done in Merging
  - **Θ(n log(n))** time
  - **Θ(n) space**
- Quick-Sort
  - Most of the work done in partitioning
  - Average case takes **Θ(n log(n))** time
  - Worst case takes **$\Theta(n^2)$** time
  - **Θ(1) space**

Tasks

- Self study
  - **Read**: Sorting Algorithms from Khan Academy
    https://www.khanacademy.org/computing/computer-science/algorithms#sorting-algorithms
- Finish Assignment 4 by Wed Oct 26 *(75 points)*
- Make progress on CodeChef (100 points)

Questions?