CSC 172– Data Structures and Algorithms

Lecture #15

Spring 2018

Please put away all electronic devices

Announcement

- Lab 7 (Sorting) will be released tonight
- Due on Sunday (1st April).

Agenda

- Sorting
- And More Sorting

Outline

In this topic, we will introduce sorting, including:

- Definitions
- Assumptions
- *In-place* sorting
- Stable sorting
- Sorting techniques and strategies
- Overview of run times

Why Sorting ?

- Why do we really need to sort the data?
  - Mostly to facilitate faster searching
  - Visualization

- Is it sometimes better not to sort the data?

Linear search vs. binary search

- Binary search
  - is much faster
  - but takes time to sort

- Pre-sort + binary search if lots of searches are expected

- m searches, T(n) sort time
  - O(mn) vs O(T(n) + m log n)

Linear Search

Sorting + Binary Search

Definition

8.1

Sorting is the process of:

– Taking a list of objects which could be stored in a linear order

$$(a_0, a_1, \ldots, a_{n-1})$$

e.g., numbers, and returning an reordering

$$(a'_0, a'_1, \ldots, a'_{n-1})$$

such that

$$a'_0 \leq a'_1 \leq \cdots \leq a'_{n-1}$$

Definition

Seldom will we sort isolated values

– Usually we will sort a number of records containing a number of fields based on a *key*:

19991532	Stevenson	Monica	3 Glendridge Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19985832	Kilji	Islam	37 Masterson Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.
19981932	Carol	Ann	81 Oakridge Ave.
20003287	Redpath	David	5 Glendale Ave.

Numerically by ID Number

19981932	Carol	Ann	81 Oakridge Ave.
19985832	Khilji	Islam	37 Masterson Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19991532	Stevenson	Monica	3 Glendridge Ave.
20003287	Redpath	David	5 Glendale Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.

Lexicographically by surname, then given name

19981932	Carol	Ann	81 Oakridge Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.
19985832	Kilji	Islam	37 Masterson Ave.
20003287	Redpath	David	5 Glendale Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19991532	Stevenson	Monica	3 Glendridge Ave.

Definition

In these topics, we will assume that:

- Arrays are to be used for both input and output,

- We will focus on sorting integers and leave the more general case of sorting records based on one or more fields as an implementation detail

In-place Sorting

Sorting algorithms may be performed *in-place*, that is, with the allocation of at most $\Theta(1)$ additional memory (*e.g.*, fixed number of local variables)

Other sorting algorithms require the allocation of second array of equal size

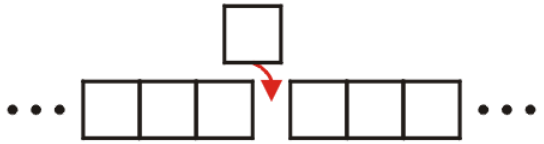
– Requires $\Theta(n)$ additional memory

We will prefer in-place sorting algorithms

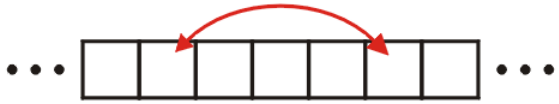
Classifications

The operations of a sorting algorithm are based on the actions performed:

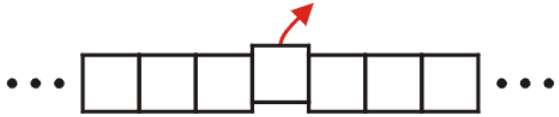
Insertion



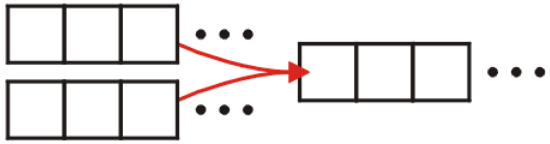
Swapping/ Exchanging



Selection



Merging



Run-time

The run time of the sorting algorithms we will look at fall into one of three categories:

$\Theta(n)$ $\Theta(n \log n)$ $\mathbf{O}(n^2)$

We will examine average- and worst-case scenarios for each algorithm

The run-time may change significantly based on the scenario

Run-time

We will review the more traditional $\mathbf{O}(n^2)$ sorting algorithms:

– Bubble sort, Selection sort, Insertion sort

Some of the faster $\mathbf{\Theta}(n \log n)$ sorting algorithms:

– Quicksort, Merge sort, and Heap sort

Lower-bound Run-time

Any sorting algorithm must examine each entry in the array at least once

– Consequently, all sorting algorithms must be $\Omega(n)$

We will not be able to achieve $\Theta(n)$ behavior without additional assumptions

Lower-bound Run-time

The general run time is $\Omega(n \log n)$

The proof depends on:

- The number of permutations of n objects is $n!$,
- A binary tree with 2^h leaf nodes has height **at least** h,
- Each permutation is a leaf node in a *comparison tree*, and
- The property that $\lg(n!) = \Theta(n \log \mathrm{n})$

https://www.mcs.sdsmt.edu/ecorwin/cs372/handouts/theta_n_factorial.htm
http://lti.cs.vt.edu/LTI_ruby/Books/CS172/html/SortingLowerBound.html

Summary

Introduction to sorting, including:

- Assumptions
- In-place sorting ($\mathbf{O}(1)$ additional memory)
- Sorting techniques
  - insertion, swapping, selection, merging
- Run-time classification: $\mathbf{O}(n)$, $\mathbf{O}(n \log \mathrm{n})$, $\mathbf{O}(n^2)$

Simple Sorting Algorithms

Sorting Algorithms Covered

- Bubble Sort
- Selection Sort
- Insertion Sort

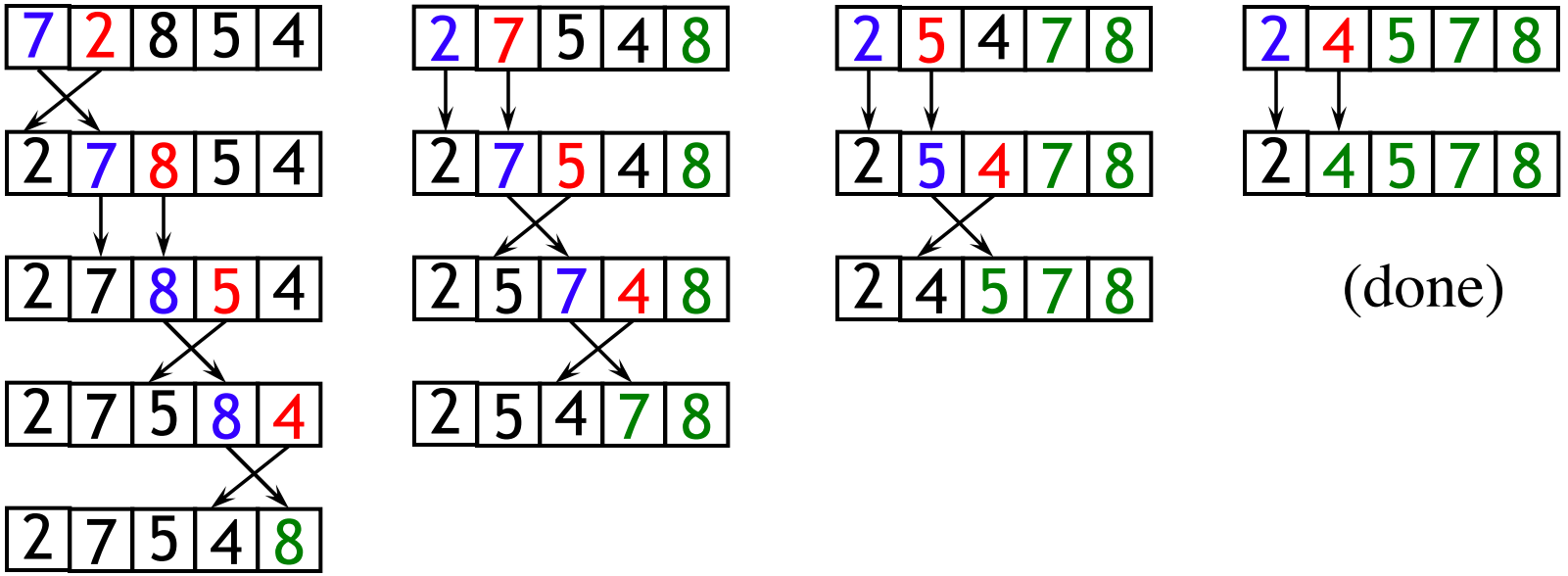
Visualization

- https://visualgo.net/bn/sorting
- http://sorting.at/#

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--) {  // counting down
      for (inner = 0; inner < outer; inner++) {        // bubbling up
         if (a[inner] > a[inner + 1]) {  // if out of order...
            int temp = a[inner];          // ...then swap
            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
  - In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time)
- Result is n * n/2 * c, that is, $O(n^2/2) = O(n^2)$

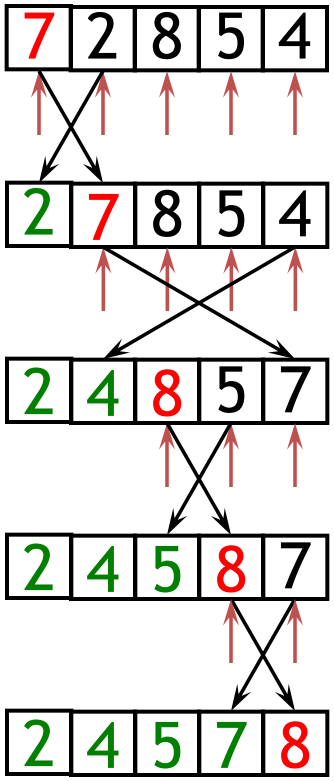
Loop invariants

- 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
  - 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 the loop exit test, outer == 0, we know that *all* elements of the array are in the correct place

Selection sort

- Given an array of length n,
  - Search elements 0 through n-1 and select the smallest
    - Swap it with the element in location 0
  - Search elements 1 through n-1 and select the smallest
    - Swap it with the element in location 1
  - Search elements 2 through n-1 and select the smallest
    - Swap it with the element in location 2
  - Search elements 3 through n-1 and select the smallest
    - Swap it with the element in location 3
  - Continue in this fashion until there's nothing left to search

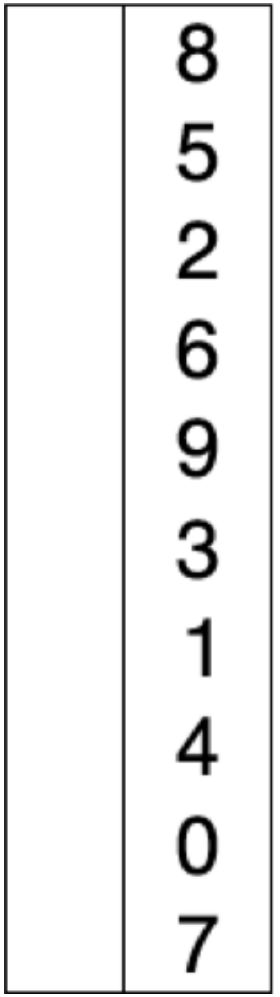
Example and analysis of selection sort



- Analysis:
  - The outer loop executes n-1 times
  - The inner loop executes about n/2 times on average (from n to 2 times)
  - Work done in the inner loop is constant (swap two array elements)
  - Time required is roughly (n-1)*(n/2)
  - You should recognize this as $O(n^2)$

Selection sort

```
void selection_sort(int[] array) {
    int i, j, k;
    for (i=0; i< array.length-1; i++) {
        j=i;
        for (k=i+1; k< array.length k++)
            if (array[k] < array[j]) j=k;
        if (j!=i) swap(array, i, j);
    }
}
```



Properties

- Number of comparisons & run-time $\Omega(n^2)$
  - Even when the input is already sorted, thus not adaptive
- Sorting is *in-place*, i.e. O(1)-extra storage
- Number of data movements is *always O(n)*: nice!
  - Important for some applications (relational database) where we want to move memory/disk blocks
-

Insertion sort

6 5 3 1 8 7 2 4

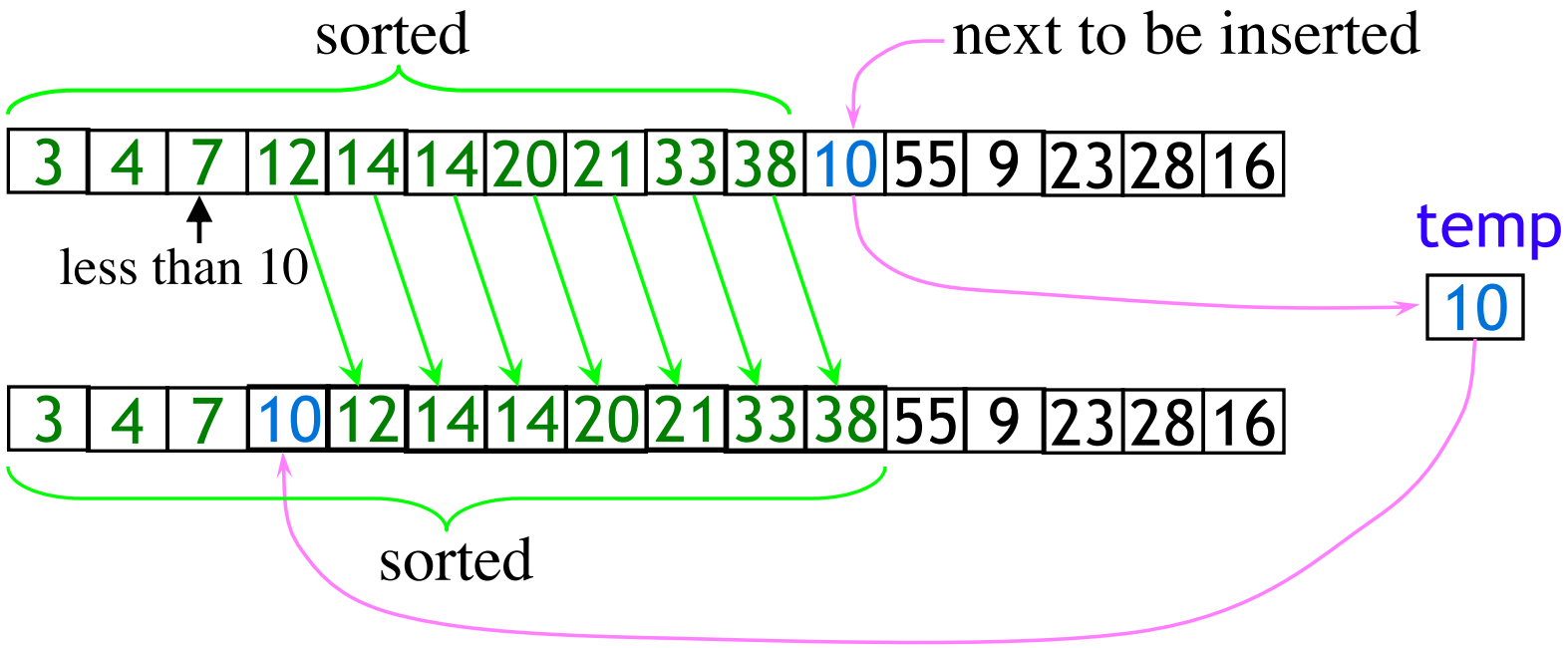
$$T(n) = O(\sum_{j=1}^{n-1} j) = O(n^2)$$

https://images-na.ssl-images-amazon.com/images/I/5164OWIqLqL.jpg

Insertion sort

```
void insertion_sort(int[] array) {
    int temp, j;
    for (int i=1; i<array.length; i++) {
        temp = array[i];
        j = i-1;
        while (j >= 0 && array[j] > temp) {
            array[j+1] = array[j];
            j--;
        }
        array[j+1] = temp;
    }
}
```

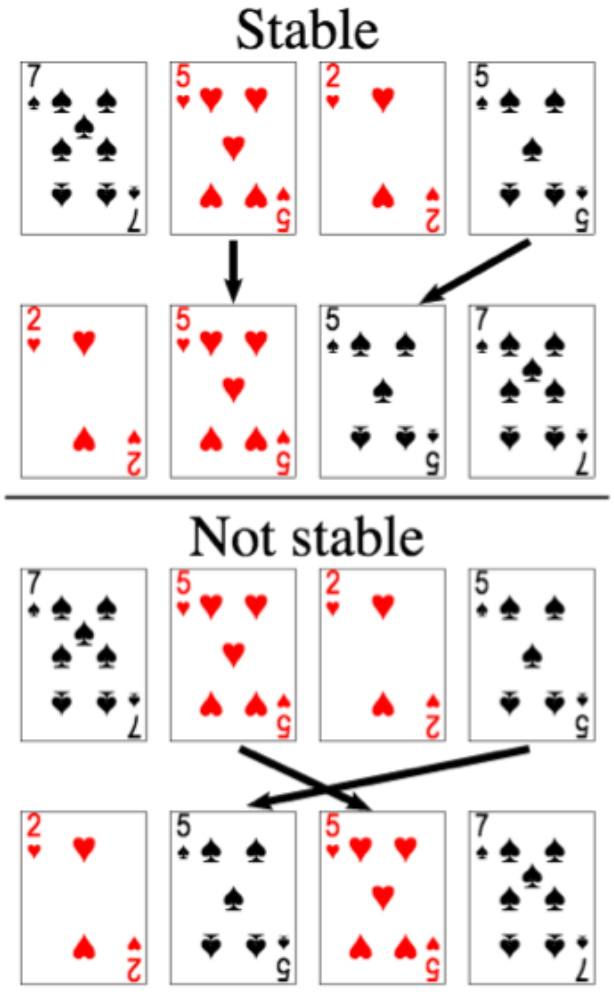
One step of insertion sort



Properties

- Worst case run time is $O(n^2)$
  - Worst case input: inversely sorted vector
- Sorting is *in-place*, i.e. O(1)-extra storage
- Number of comparisons $\Omega(n^2)$ at worst
- Number of item moves is also $\Omega(n^2)$ at worst
- ***Adaptive**: $O(n)$* time for nearly sorted input
- It is a *stable sort* algorithm

Stable vs. Not Stable



More efficient sorting algorithms

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 (left ≥ right), 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

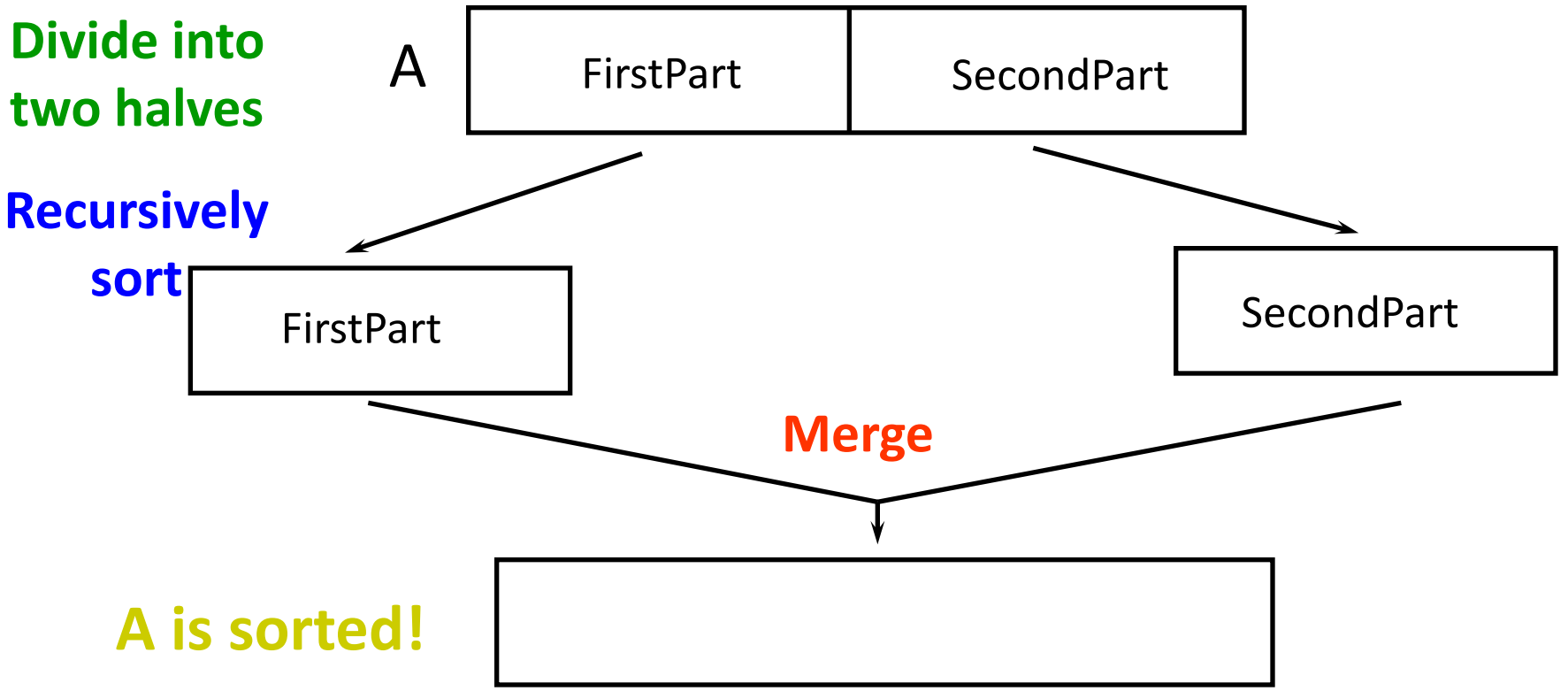
Merge sort

6 5 3 1 8 7 2 4

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

A classic divide-and-conquer algorithm

Merge Sort: Idea



Merge Sort: Algorithm

```
Merge-Sort (A, left, right)
  if    left ≥ right   return
  else
        middle ← b(left+right)/2⌋
        Merge-Sort(A, left, middle)
        Merge-Sort(A, middle+1, right)
        Merge(A, left, middle, right)
```

Recursive Call

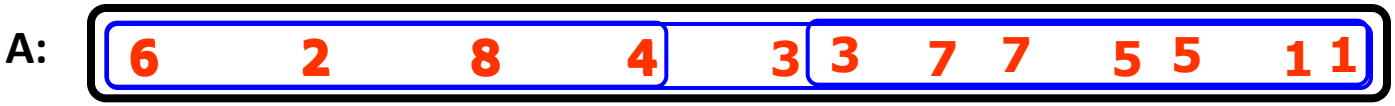
Merge sort

```
public static void MergeSort(int [] arr)
{
    if (arr.length <= 1) return;
    int[] left = new int[arr.length/2];
    int [] right = new int[arr.length - arr.length/2];
    System.arraycopy(arr, 0, left, 0, arr.length/2);
    System.arraycopy(arr, arr.length/2, right, 0,
arr.length - arr.length/2);

    MergeSort(left);
    MergeSort(right);

     merge(arr, left, right);
}
```

An important idea!

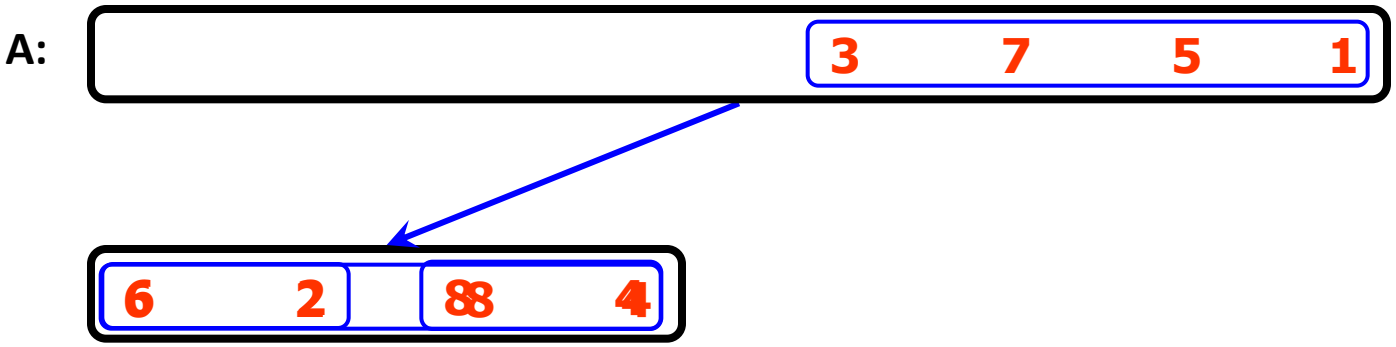
Merge-Sort(A, 0, 7)

Divide



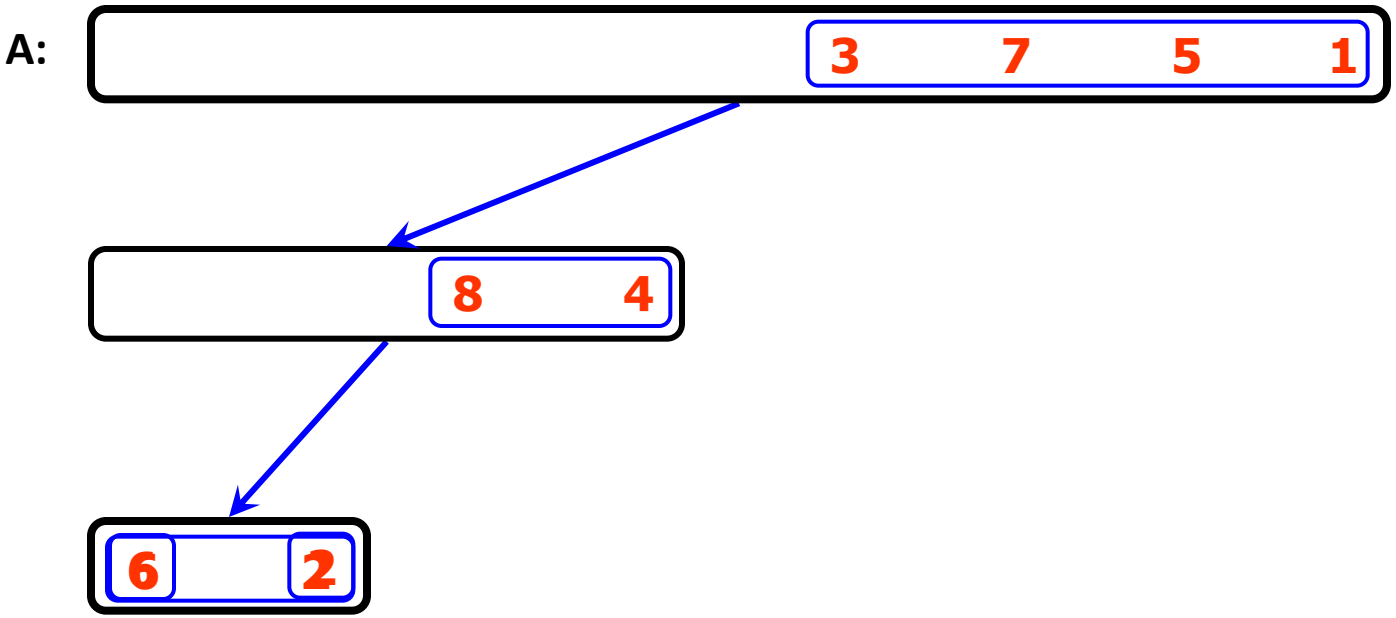
Merge-Sort(A, 0, 7)

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



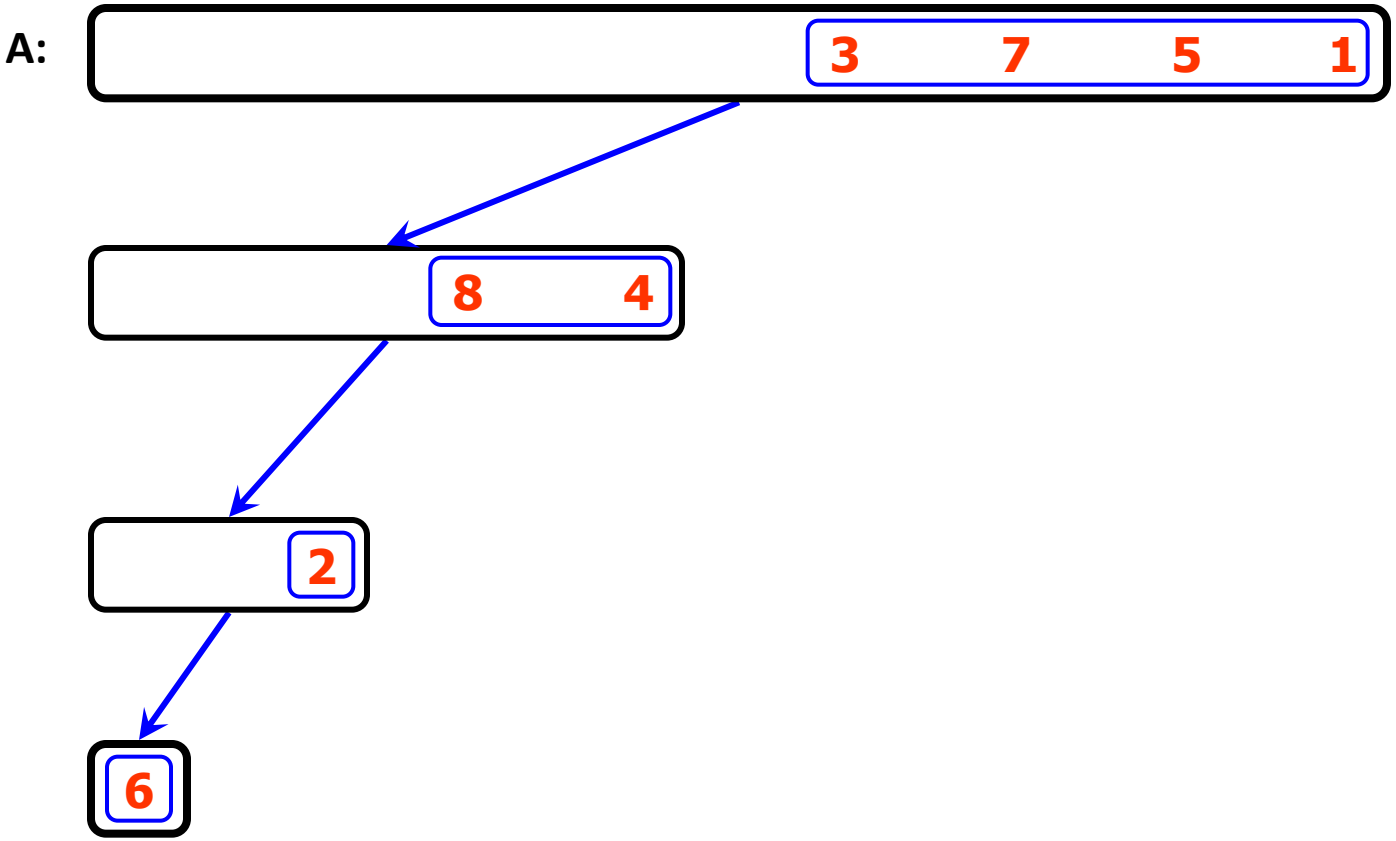
Merge-Sort(A, 0, 7)

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



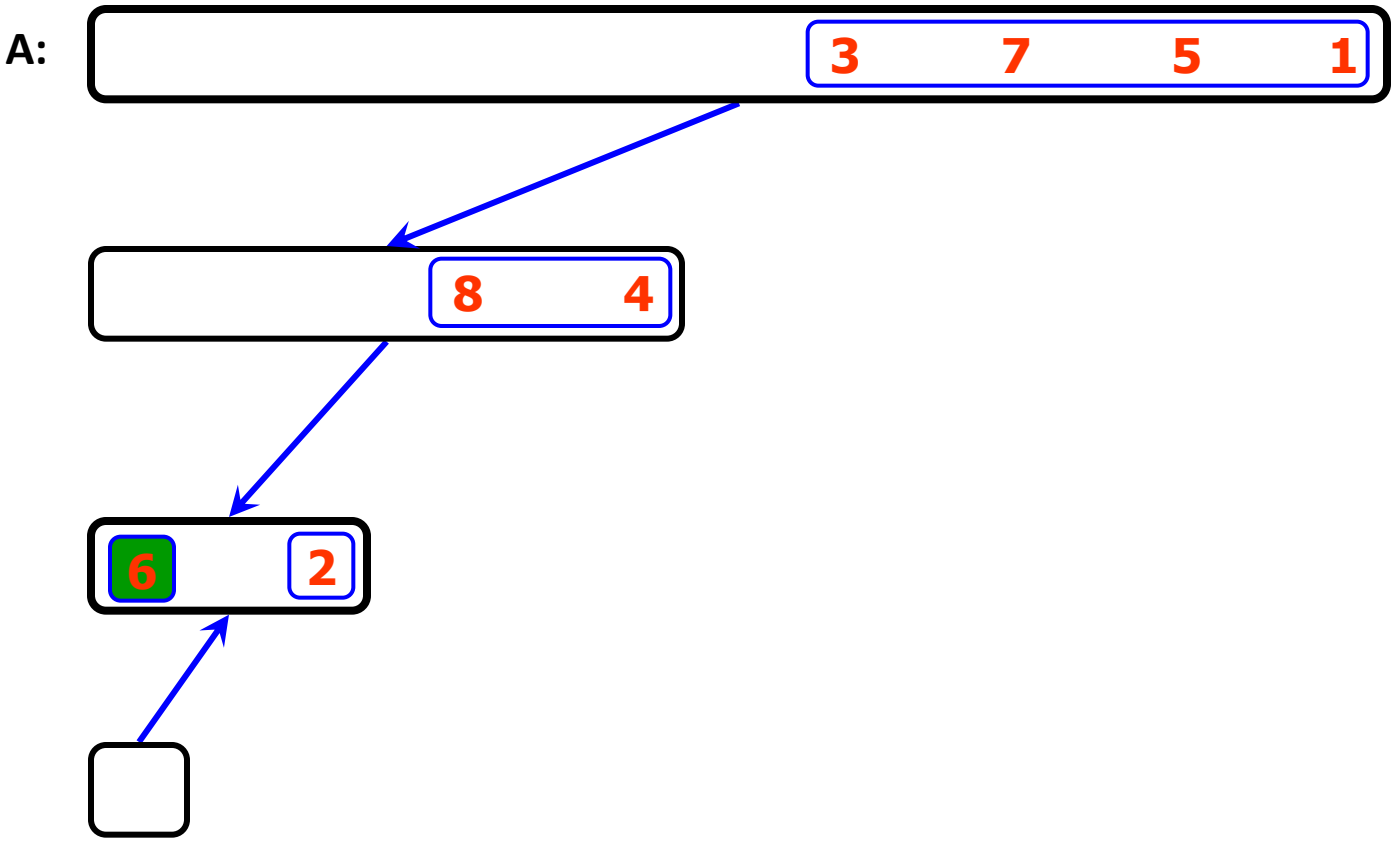
Merge-Sort(A, 0, 7)

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



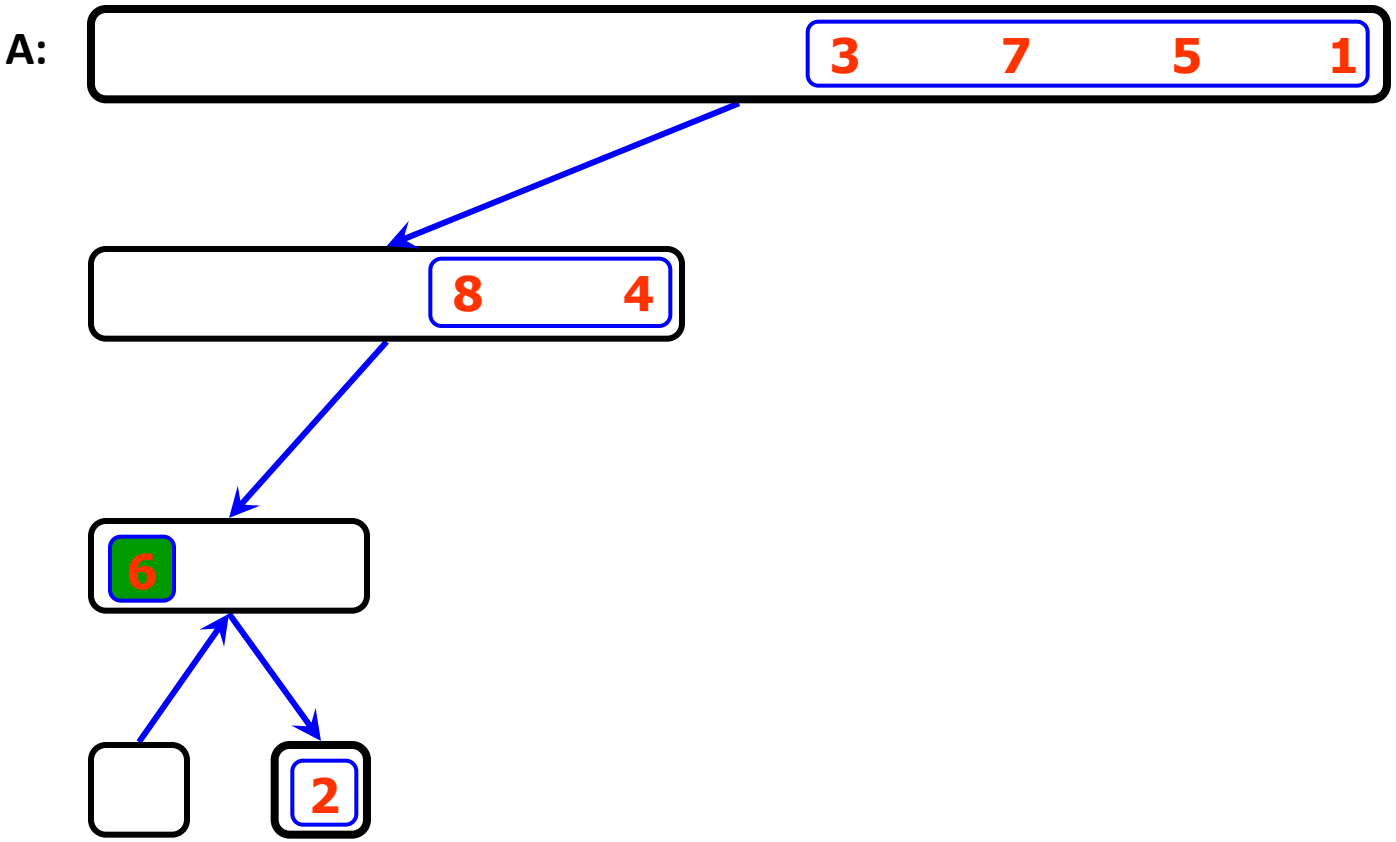
Merge-Sort(A, 0, 7)

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



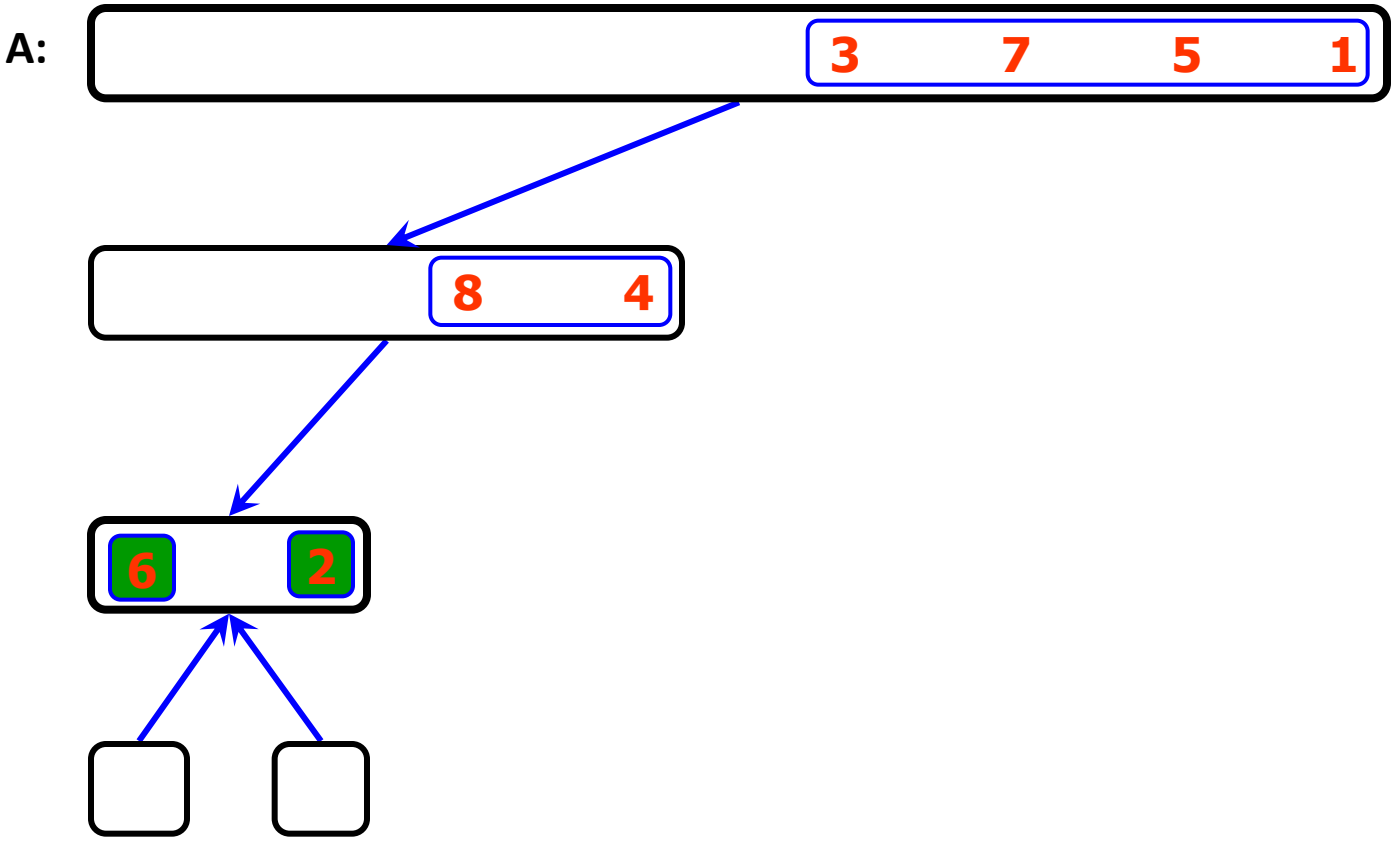
Merge-Sort(A, 0, 7)

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



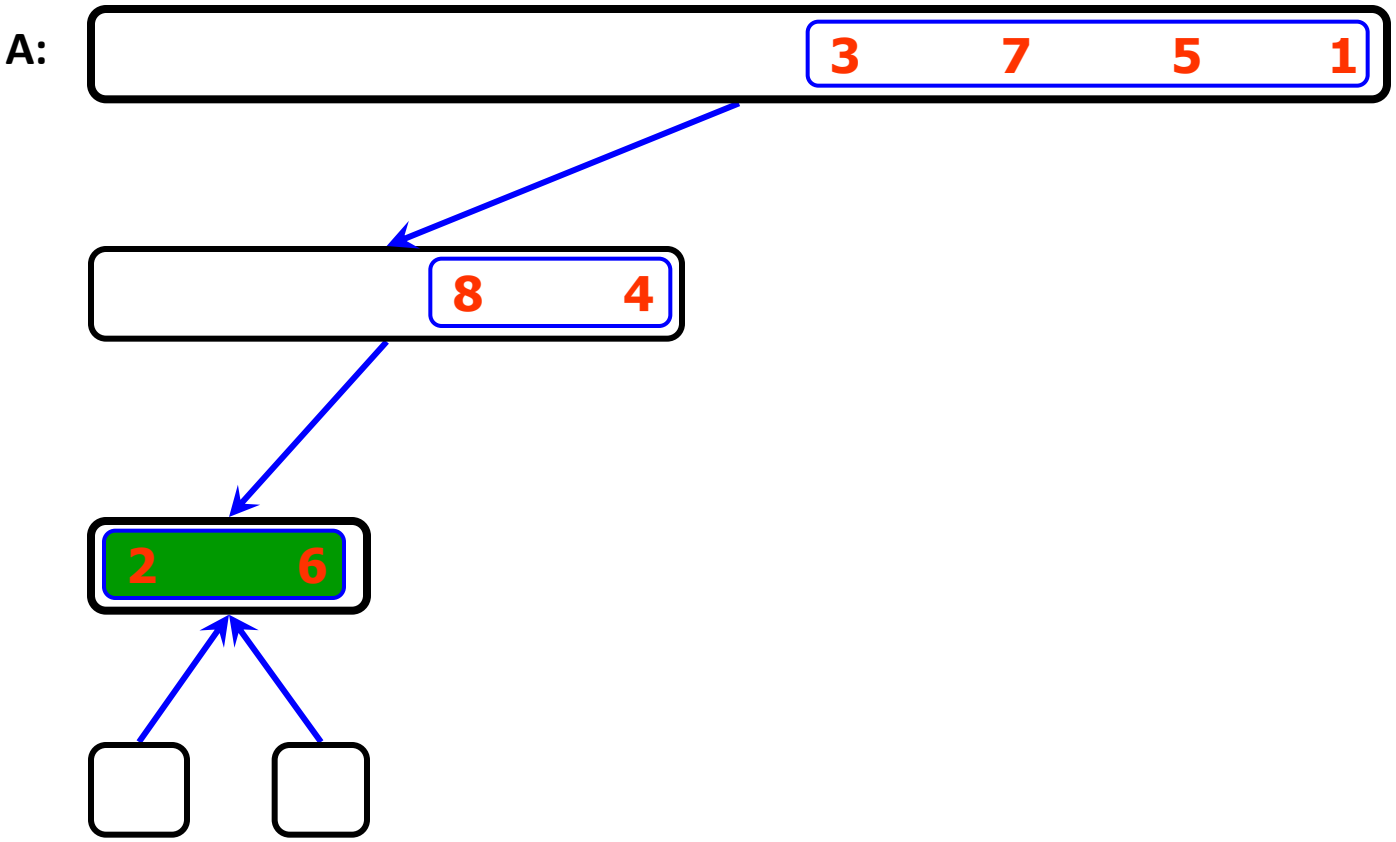
Merge-Sort(A, 0, 7)

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



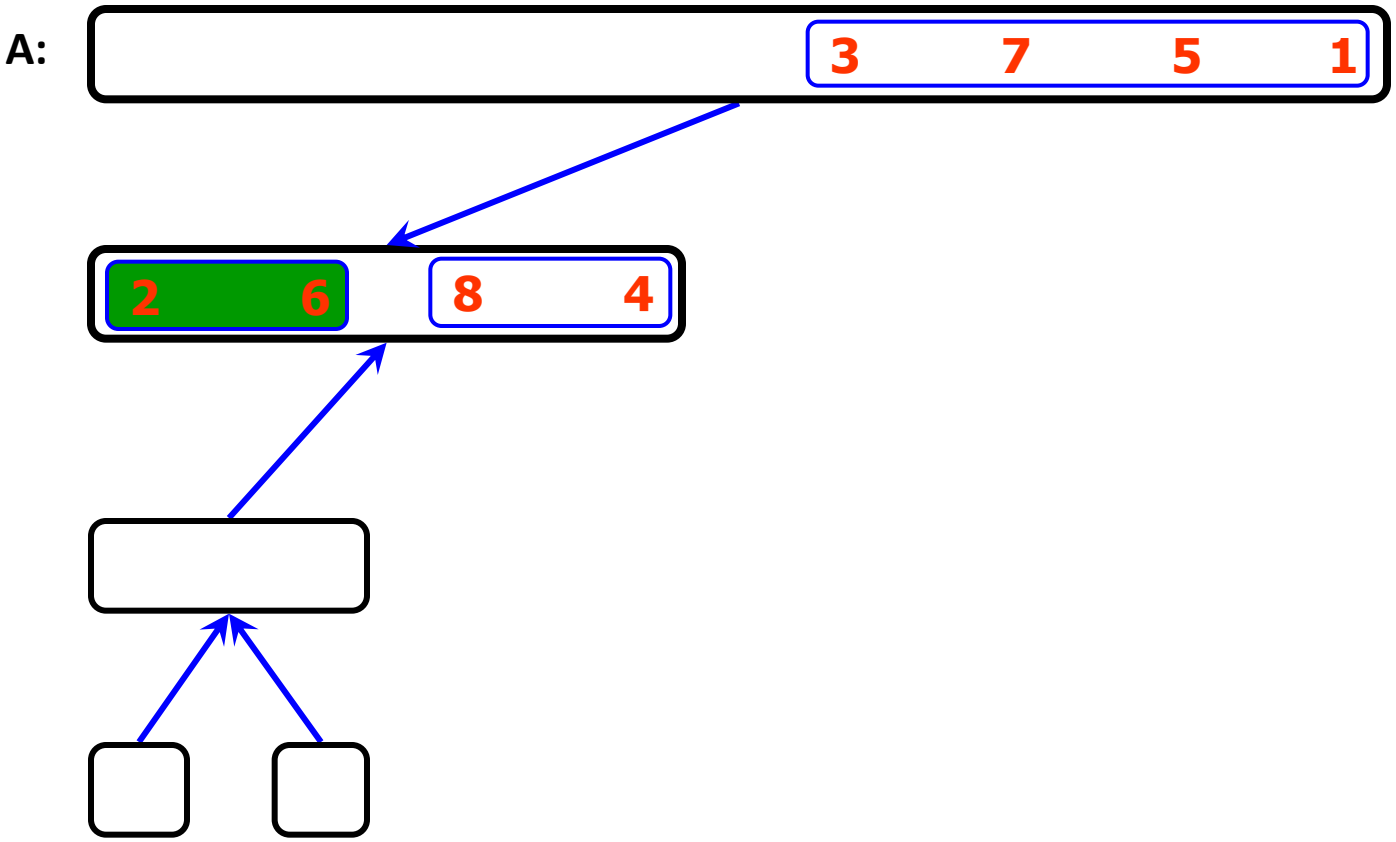
Merge-Sort(A, 0, 7)

Merge(A, 0, 0, 1)



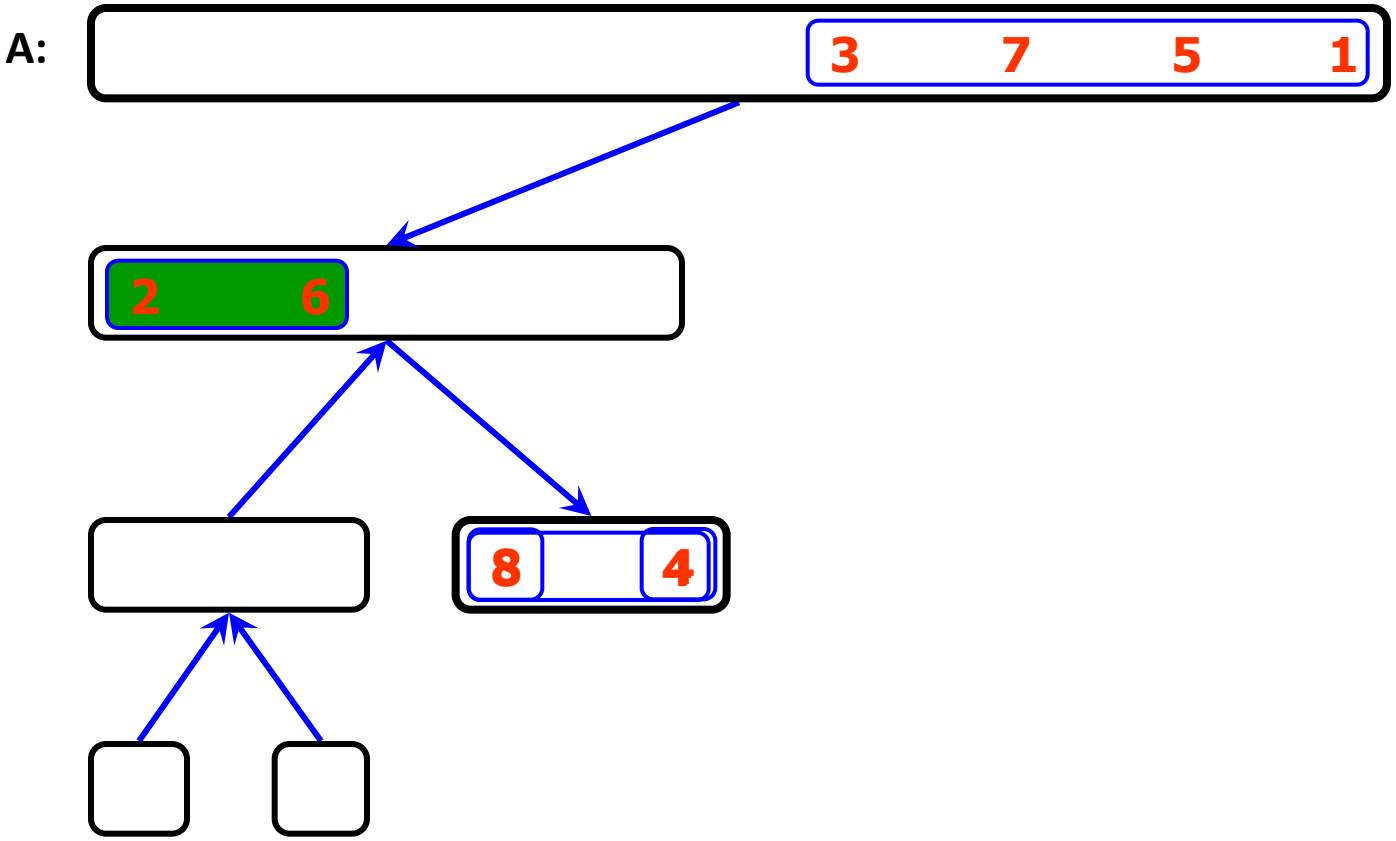
Merge-Sort(A, 0, 7)

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



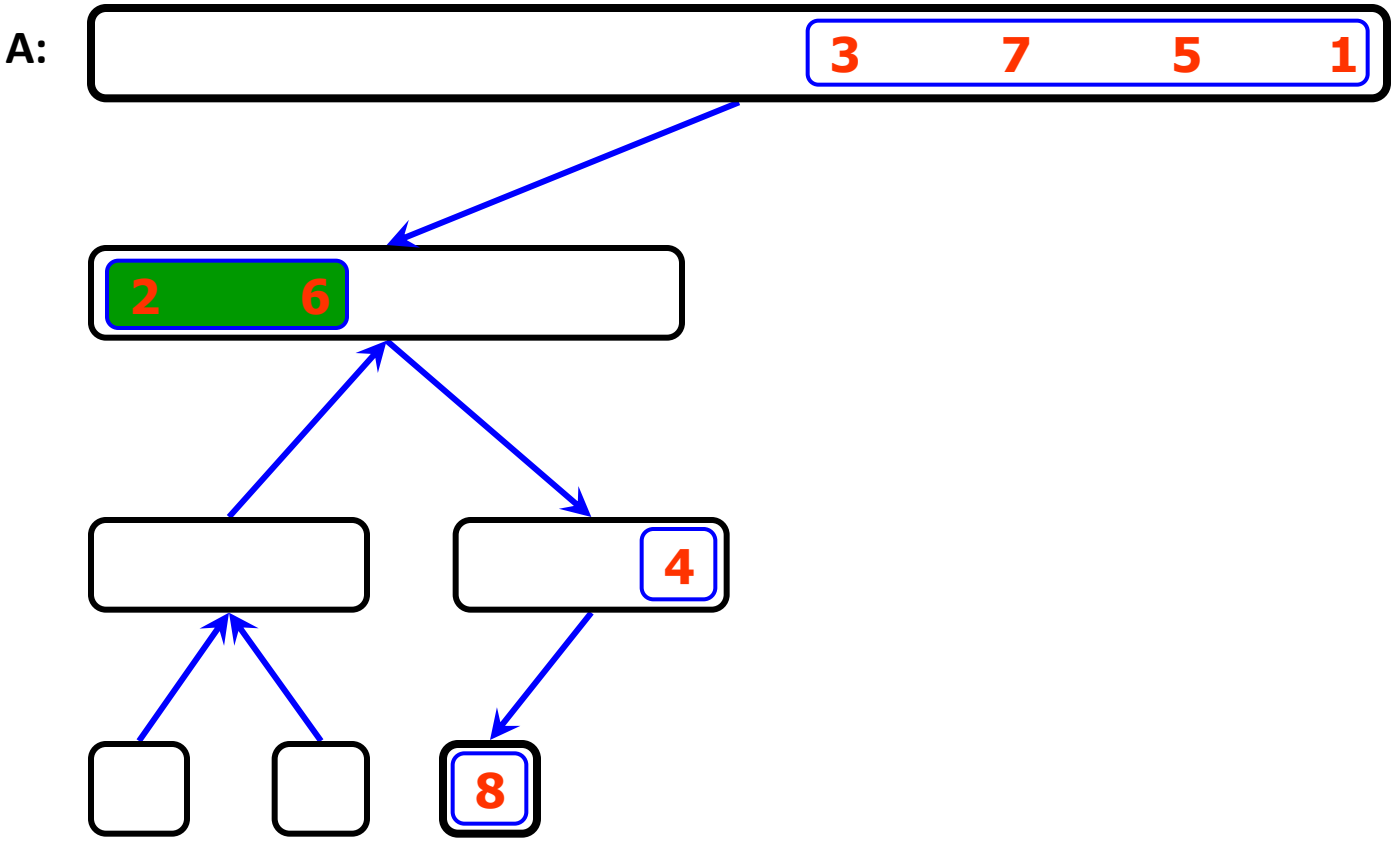
Merge-Sort(A, 0, 7)

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



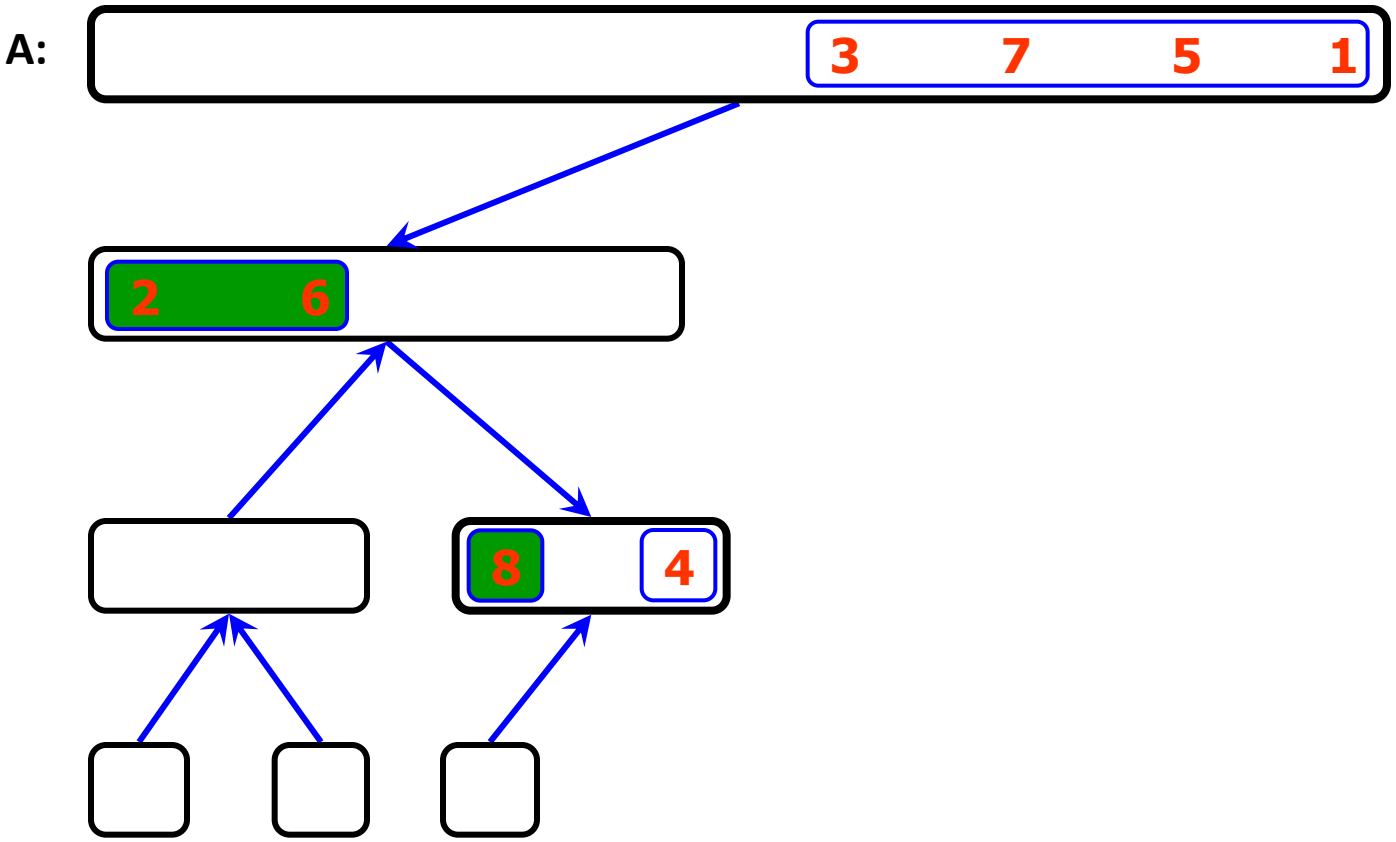
Merge-Sort(A, 0, 7)

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



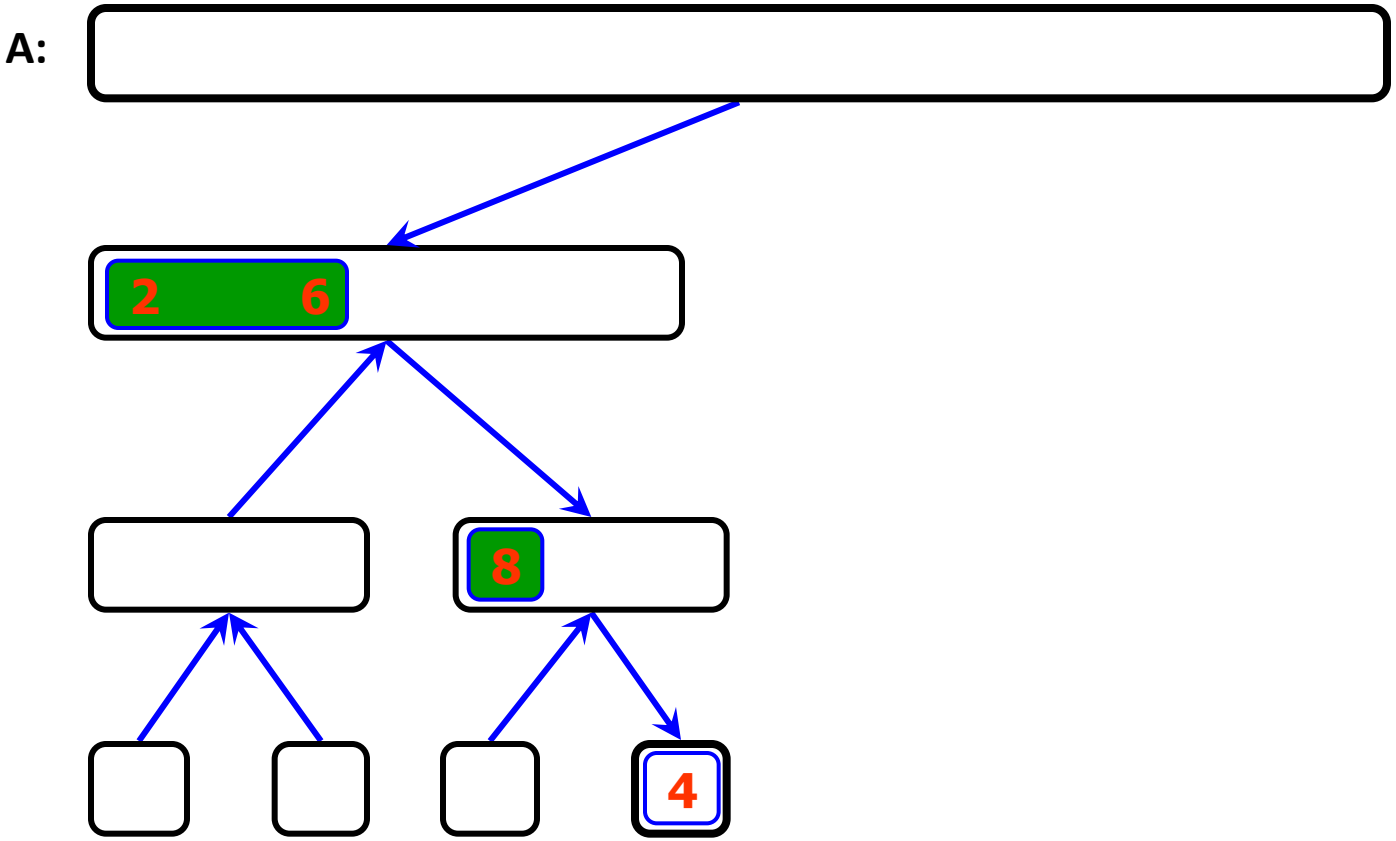
Merge-Sort(A, 0, 7)

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



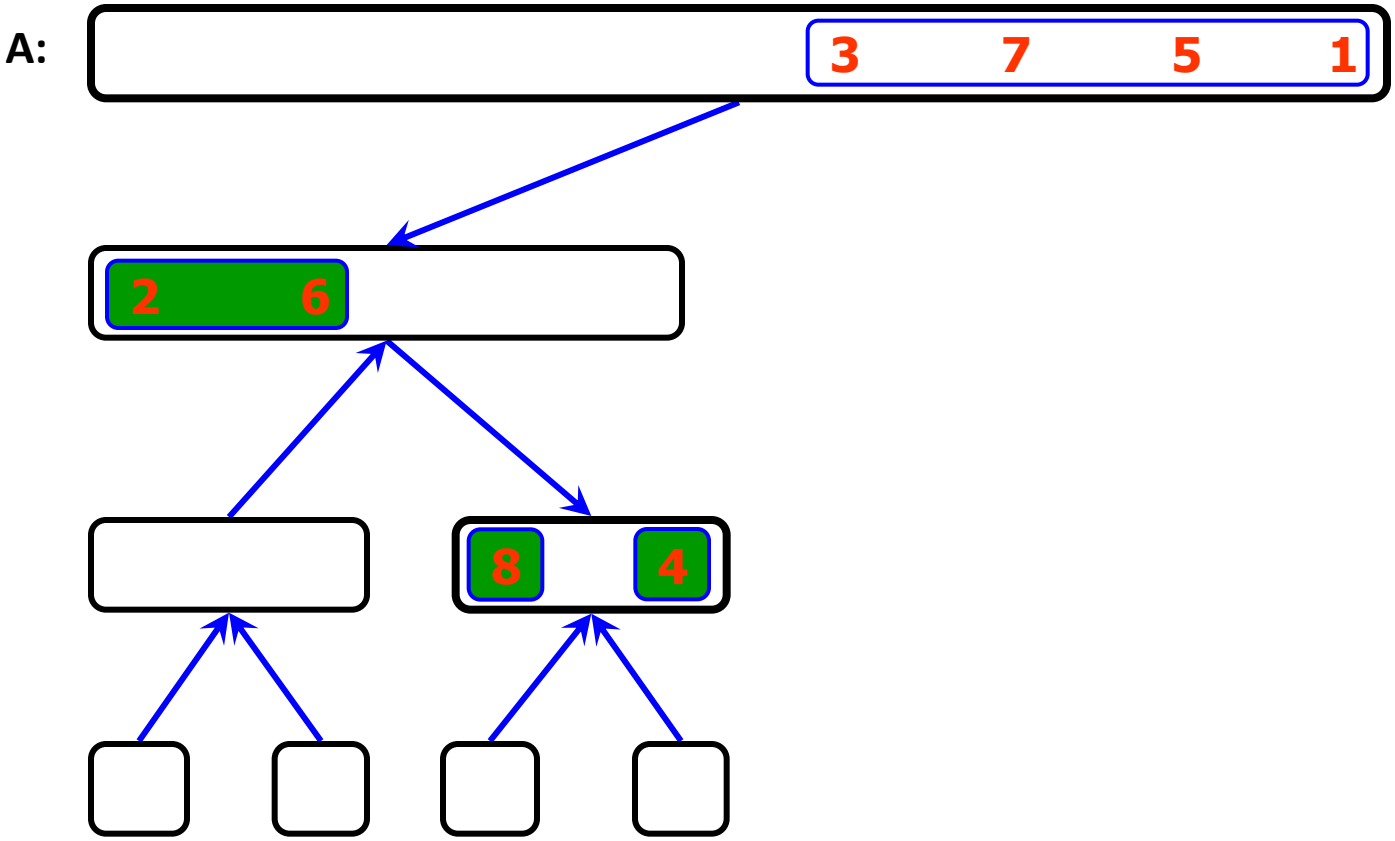
Merge-Sort(A, 0, 7)

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



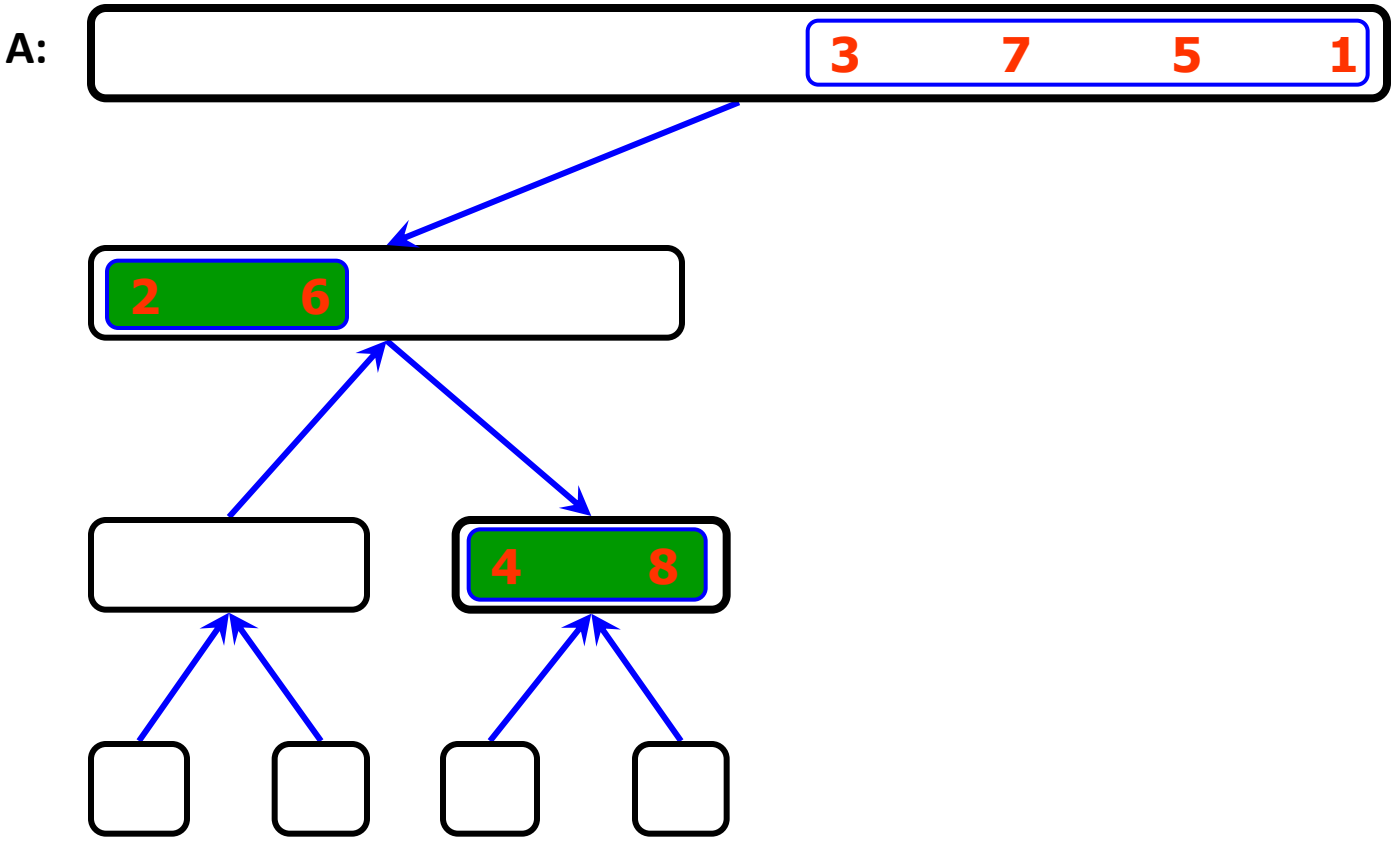
Merge-Sort(A, 0, 7)

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



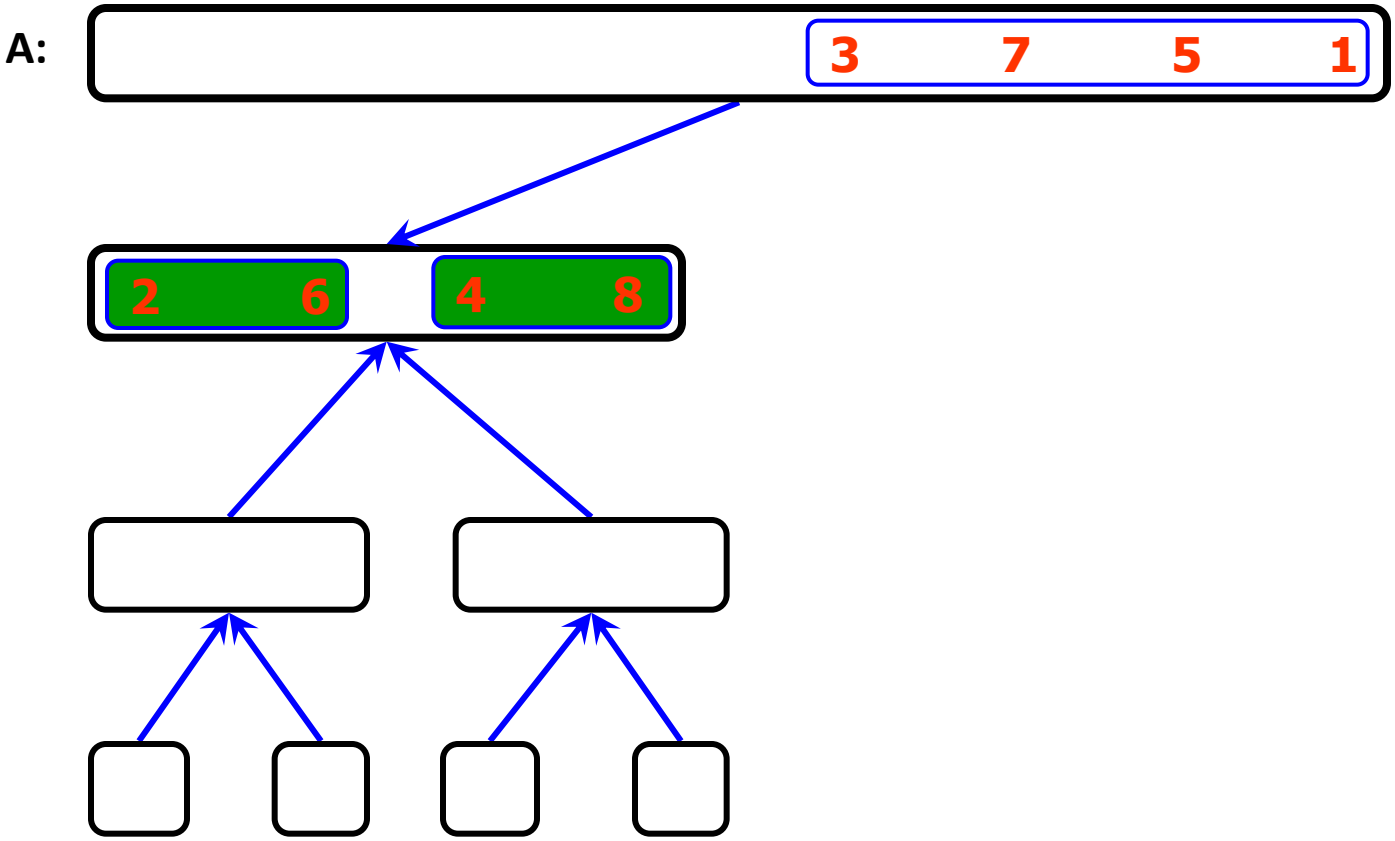
Merge-Sort(A, 0, 7)

Merge(A, 2, 2, 3)



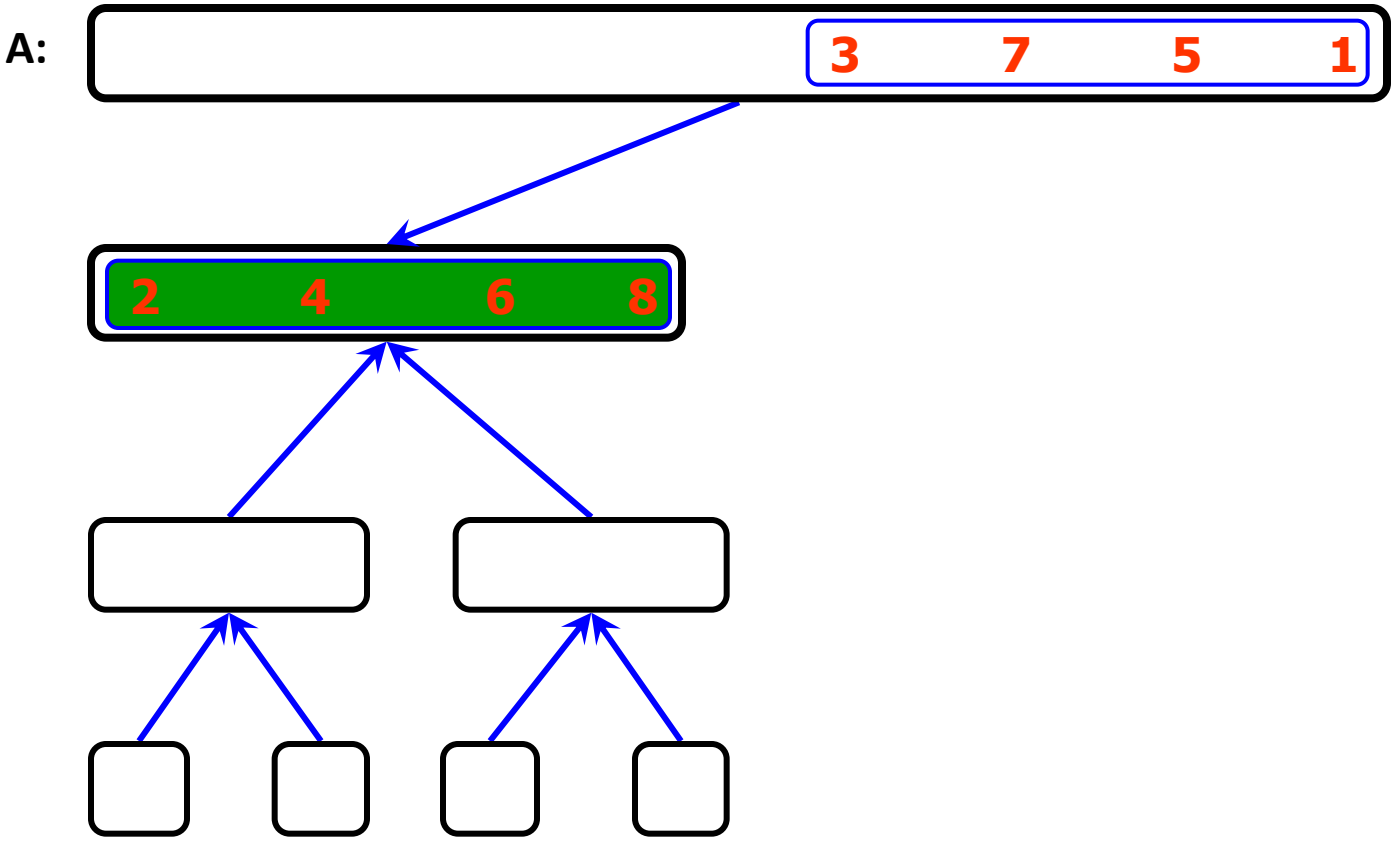
Merge-Sort(A, 0, 7)

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



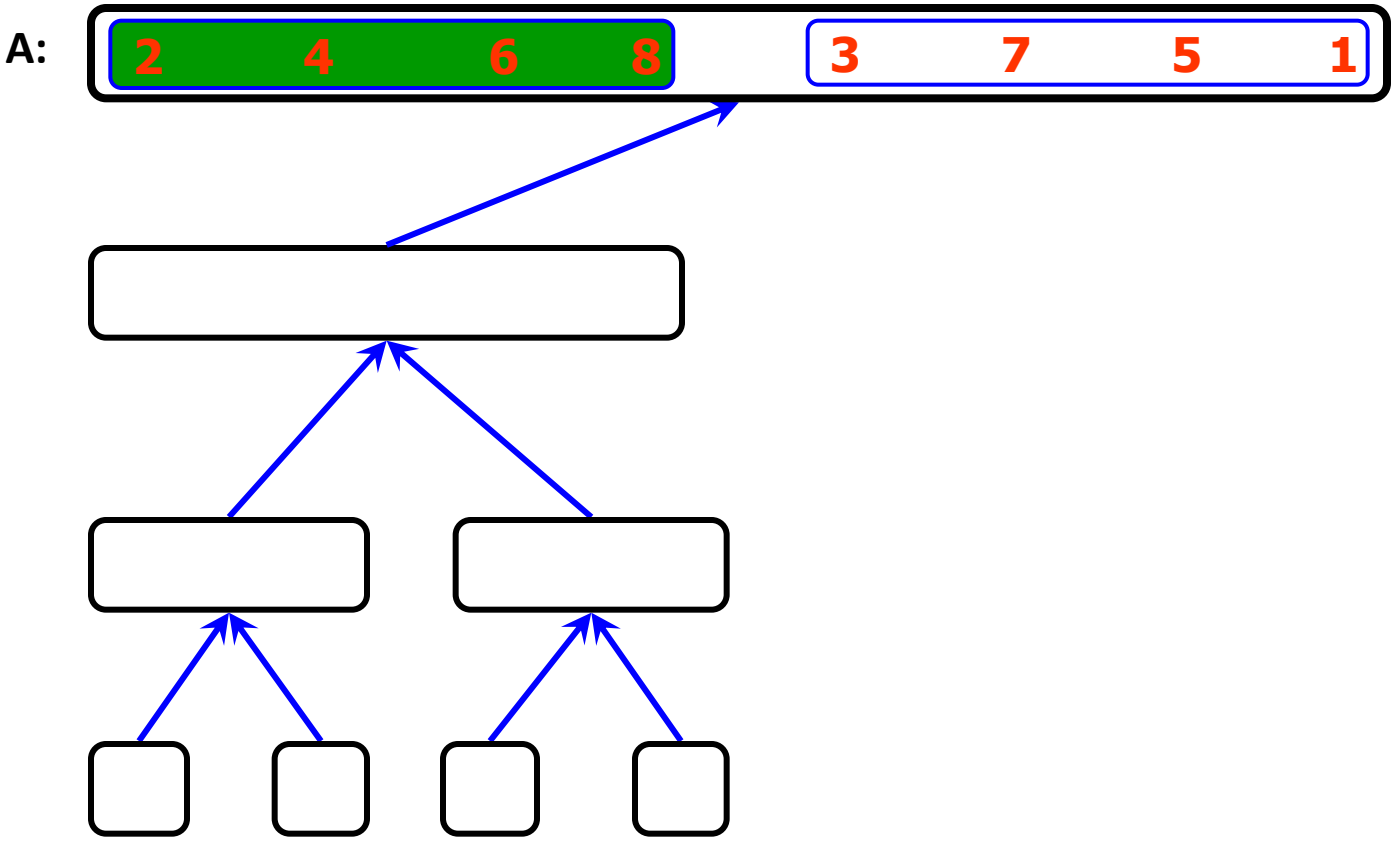
Merge-Sort(A, 0, 7)

Merge(A, 0, 1, 3)



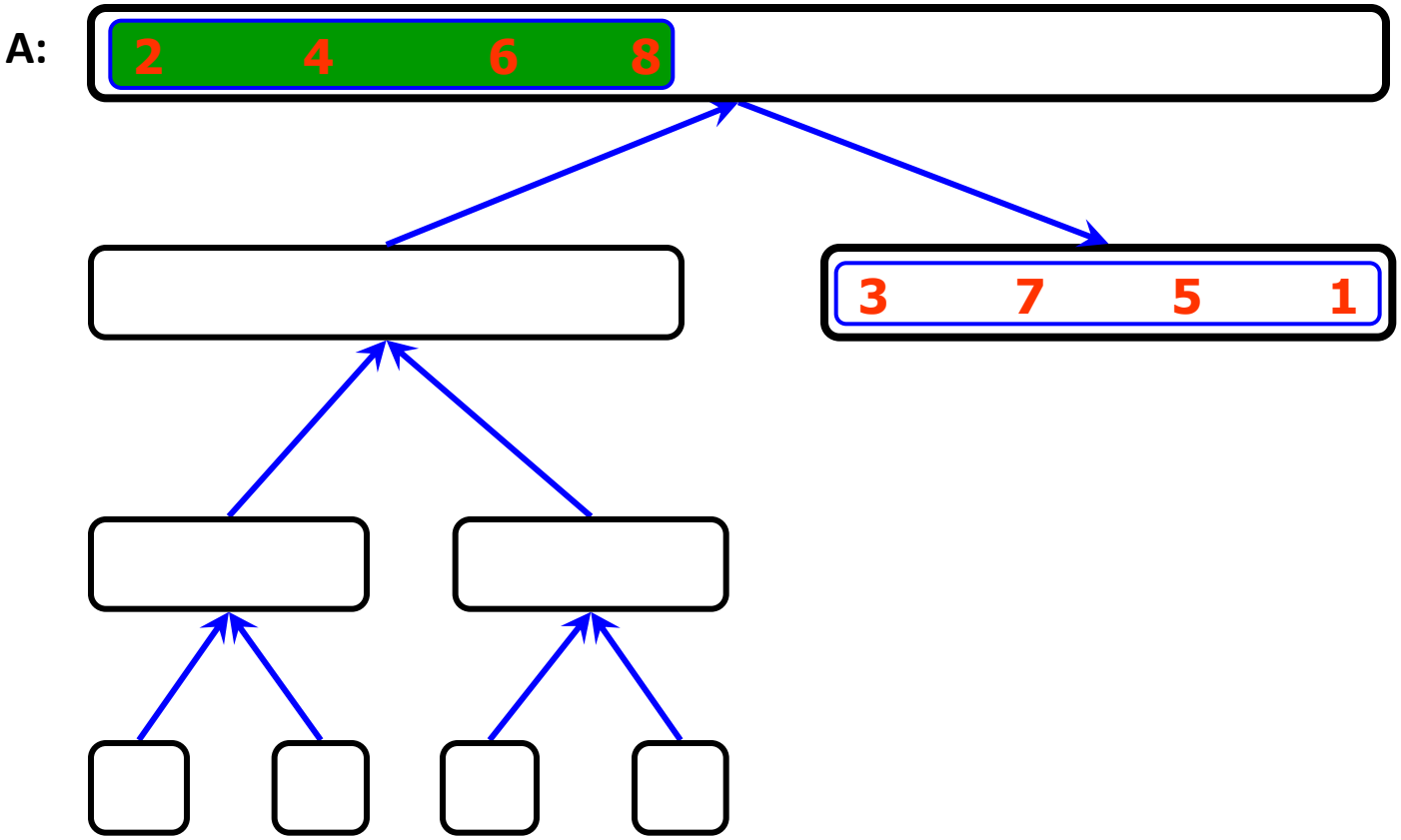
Merge-Sort(A, 0, 7)

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



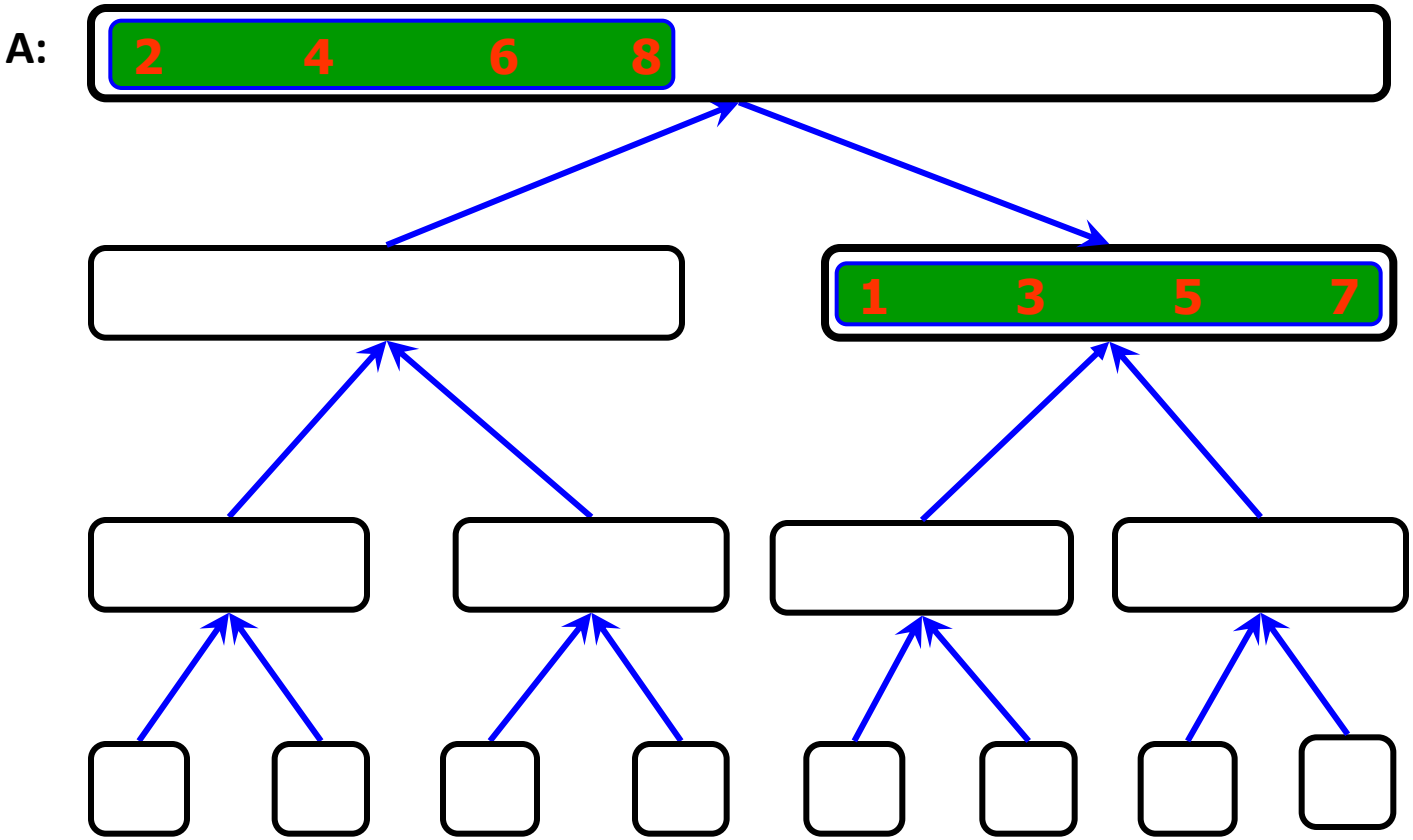
Merge-Sort(A, 0, 7)

Merge-Sort(A, 4, 7)



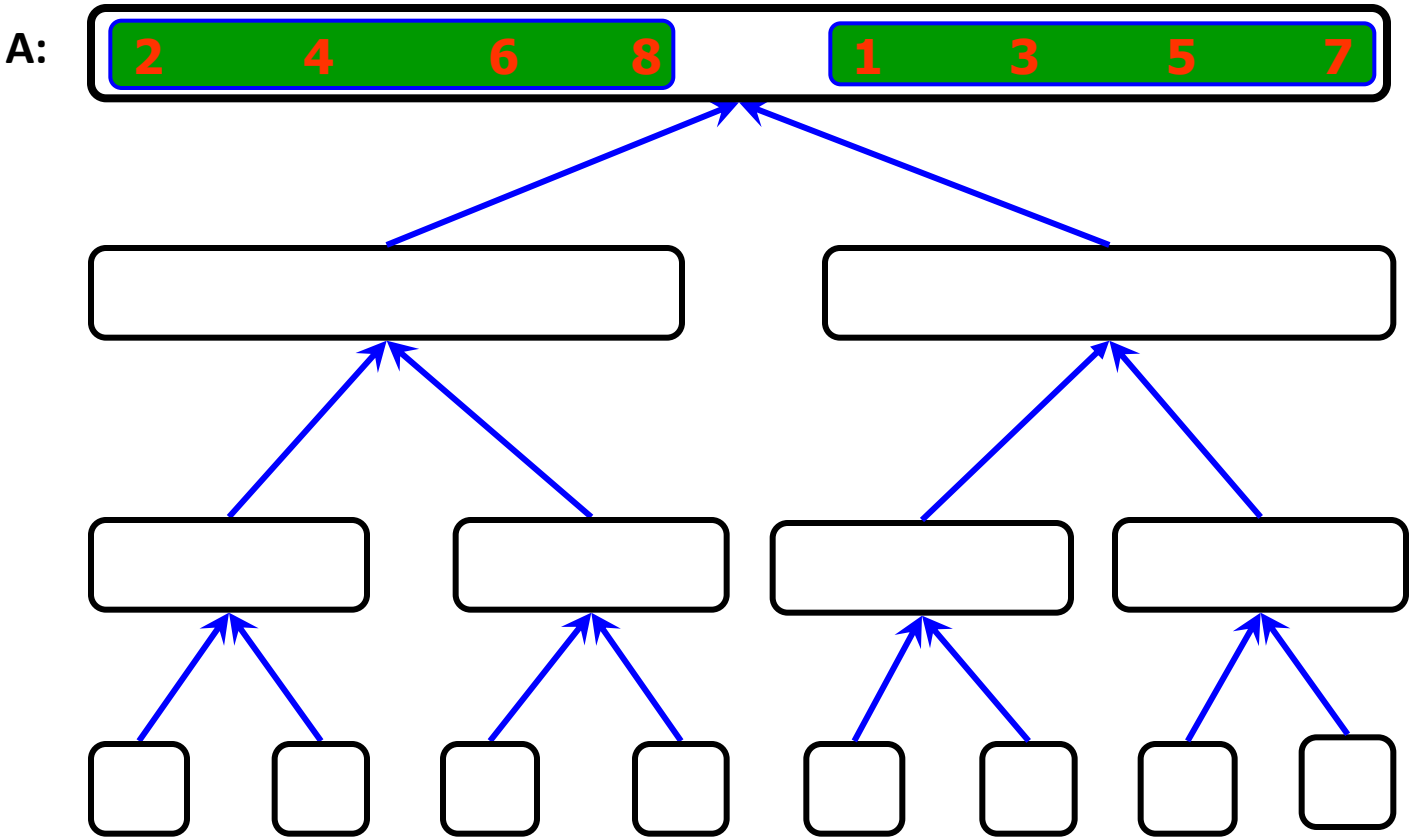
Merge-Sort(A, 0, 7)

Merge (A, 4, 5, 7)



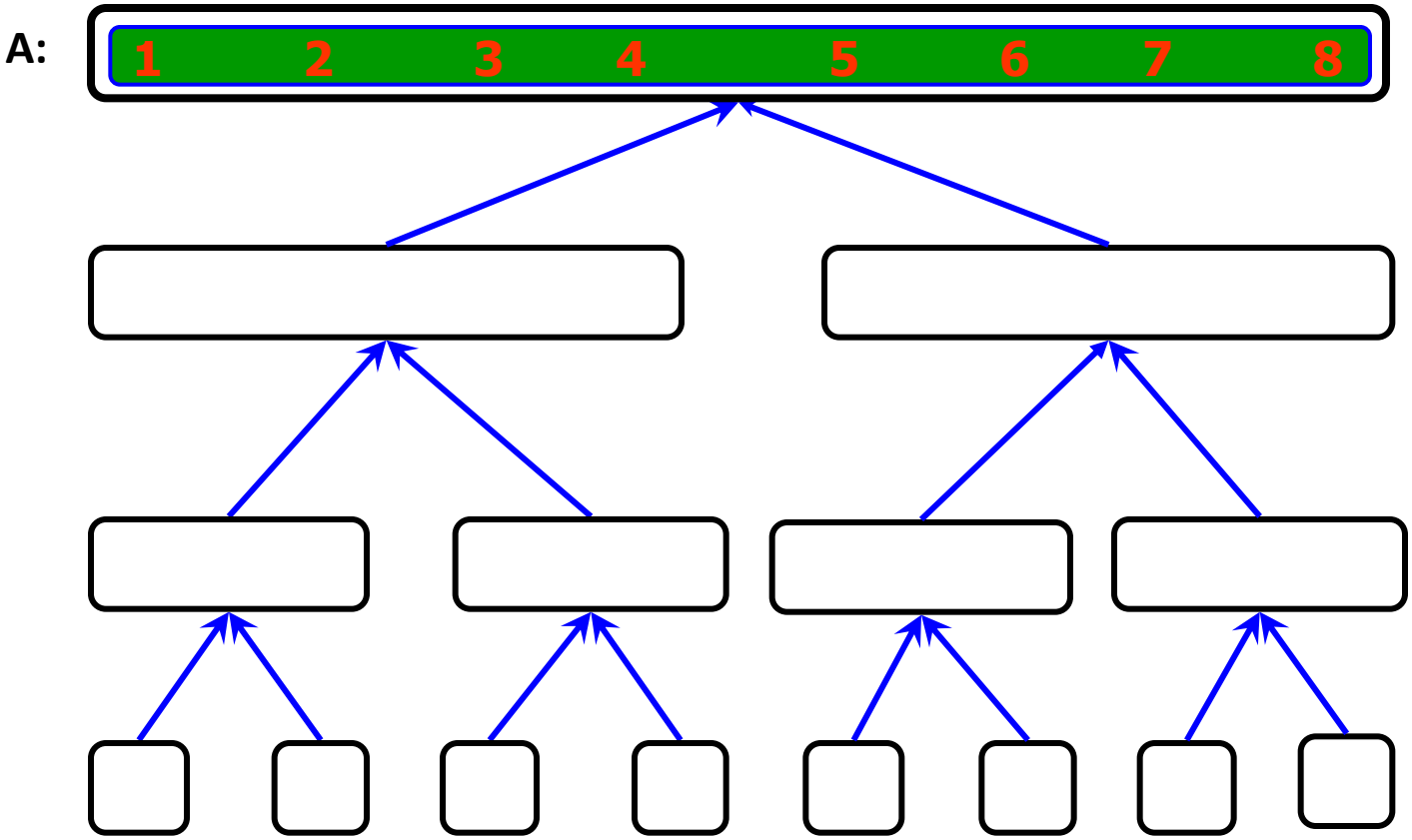
Merge-Sort(A, 0, 7)

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

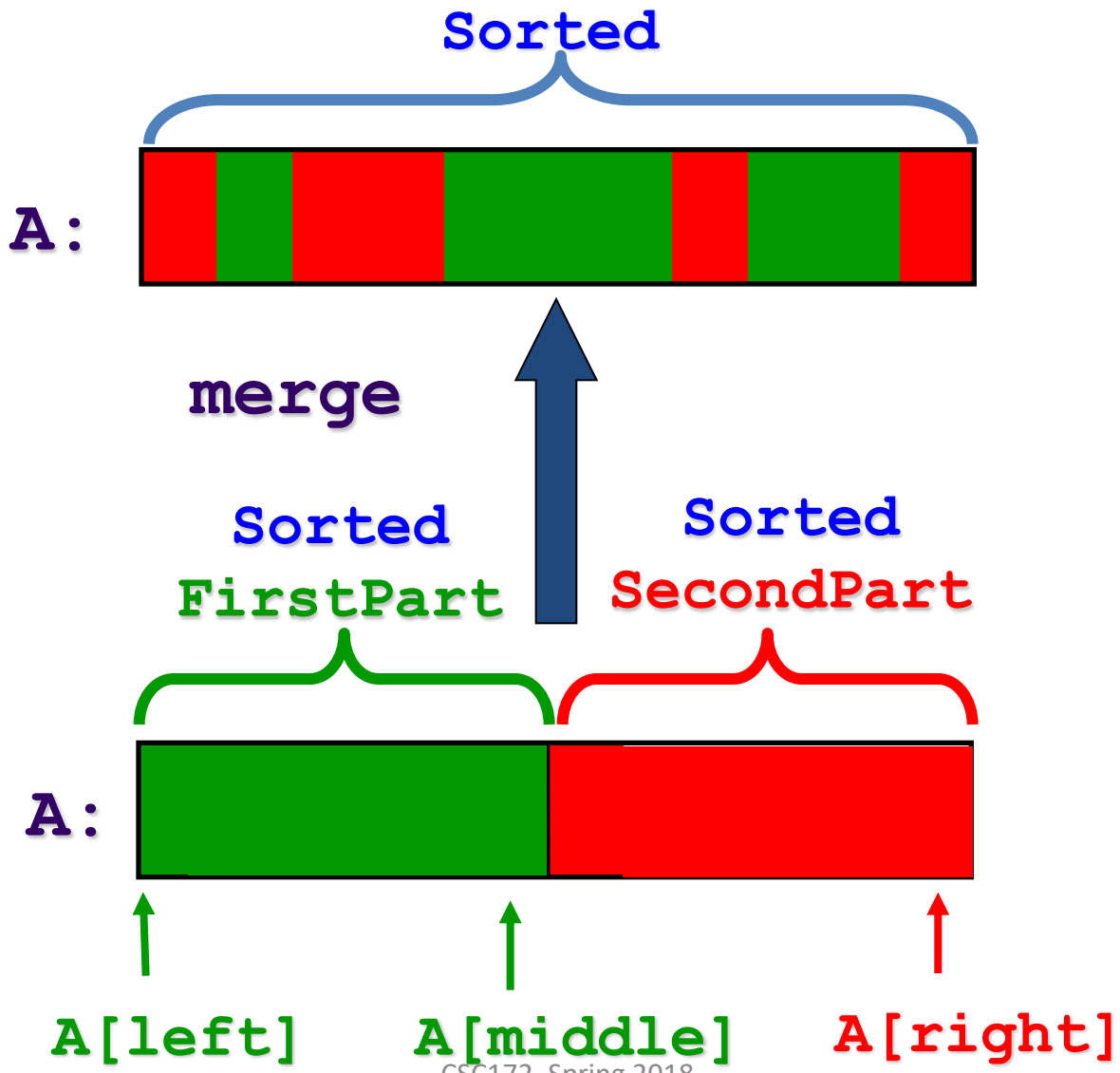


Merge-Sort(A, 0, 7)

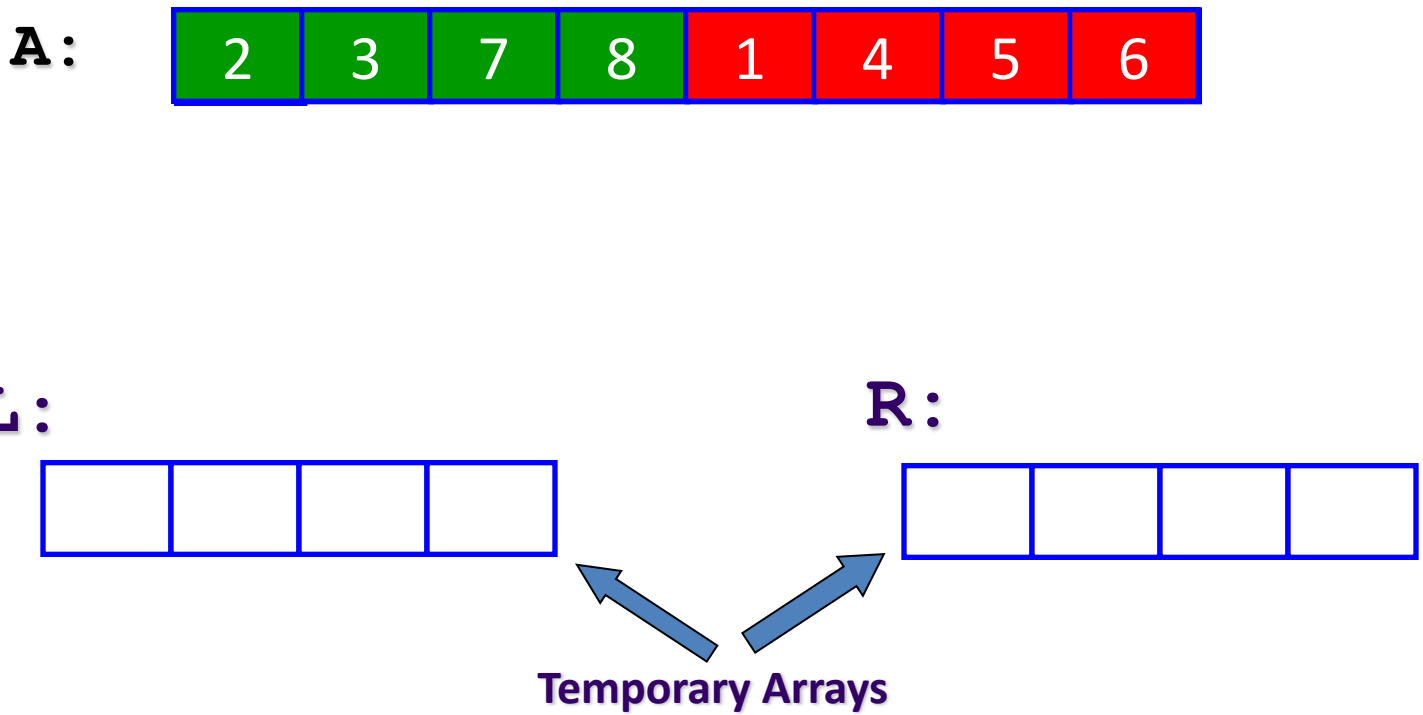
Merge-Sort(A, 0, 7), done!



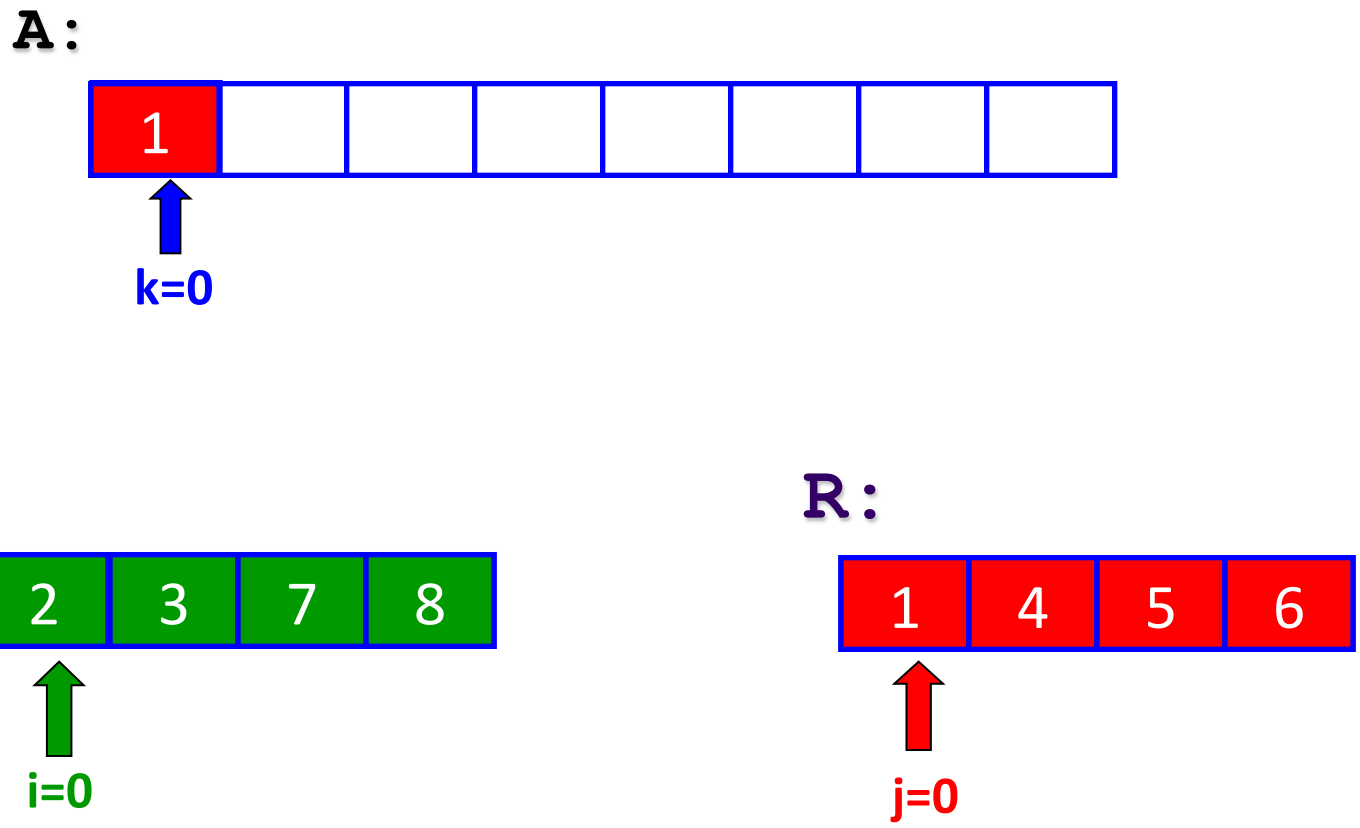
Merge-Sort: Merge



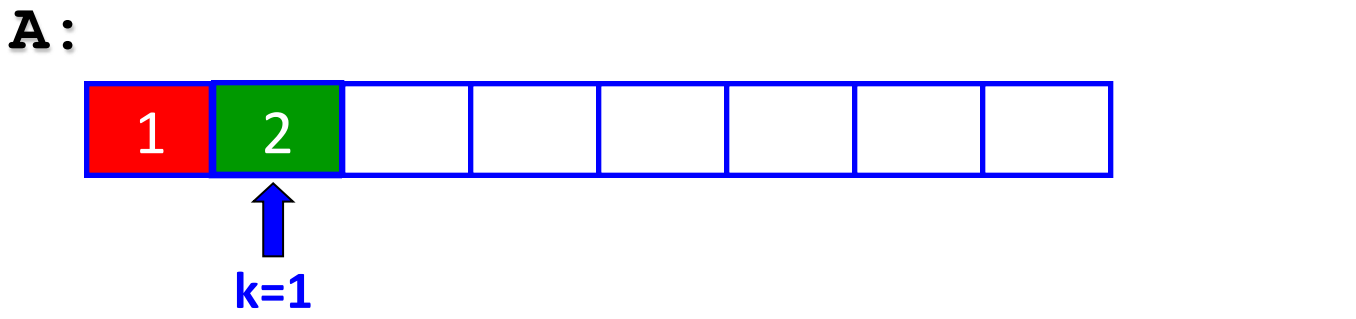
Merge-Sort: Merge Example

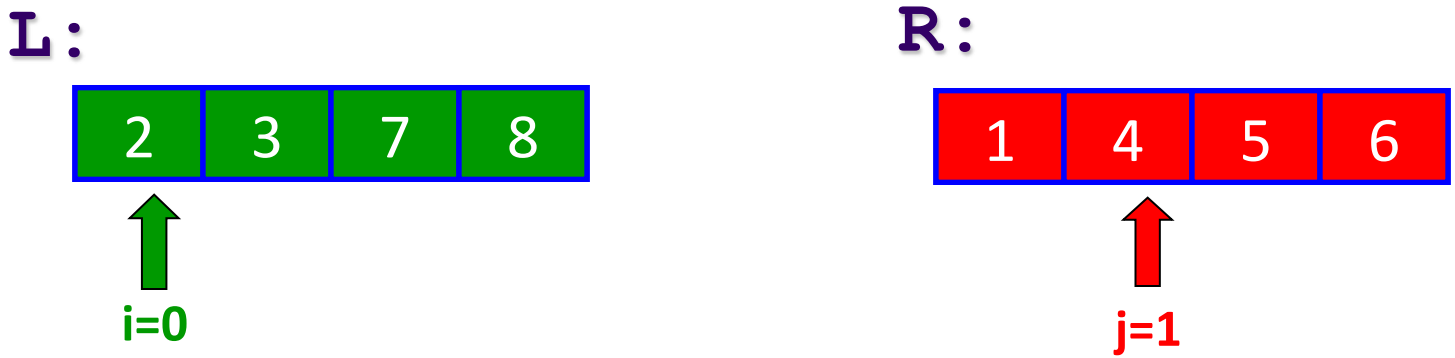


Merge-Sort: Merge Example

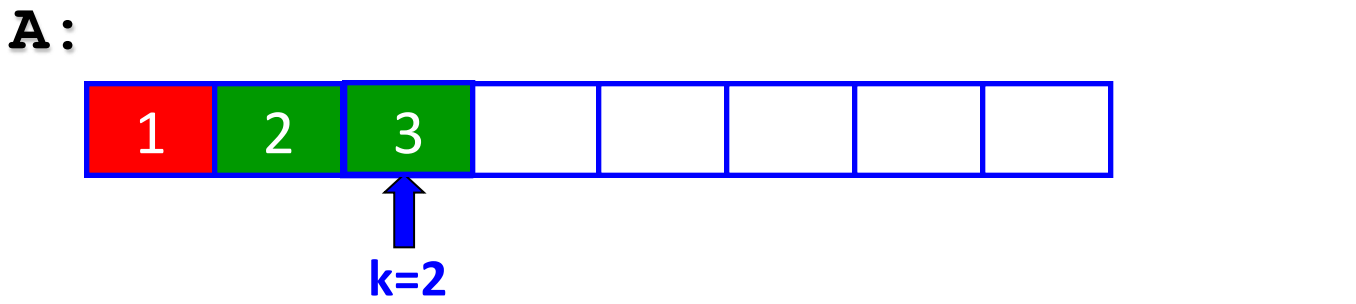


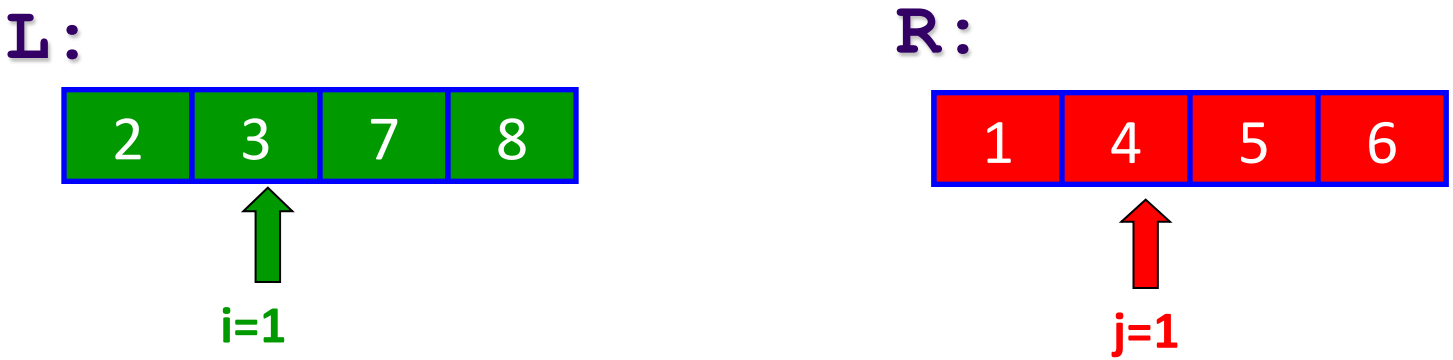
Merge-Sort: Merge Example



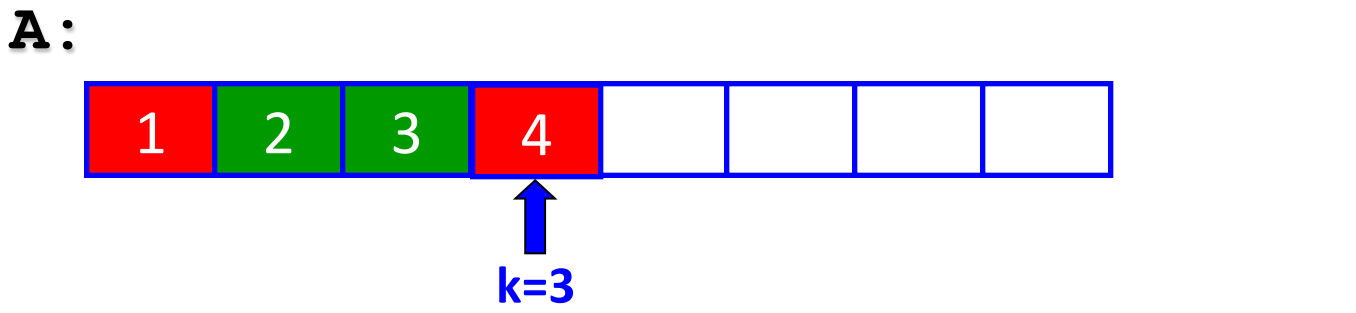


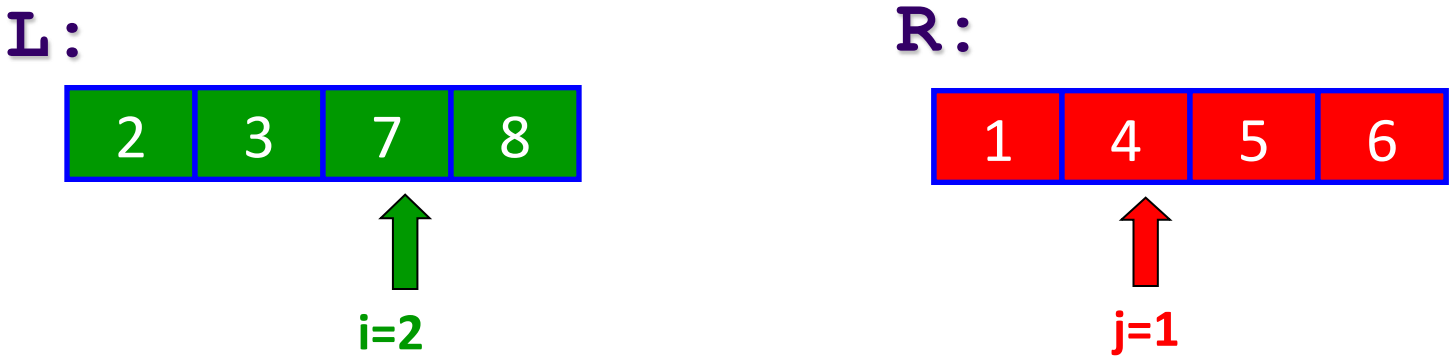
Merge-Sort: Merge Example



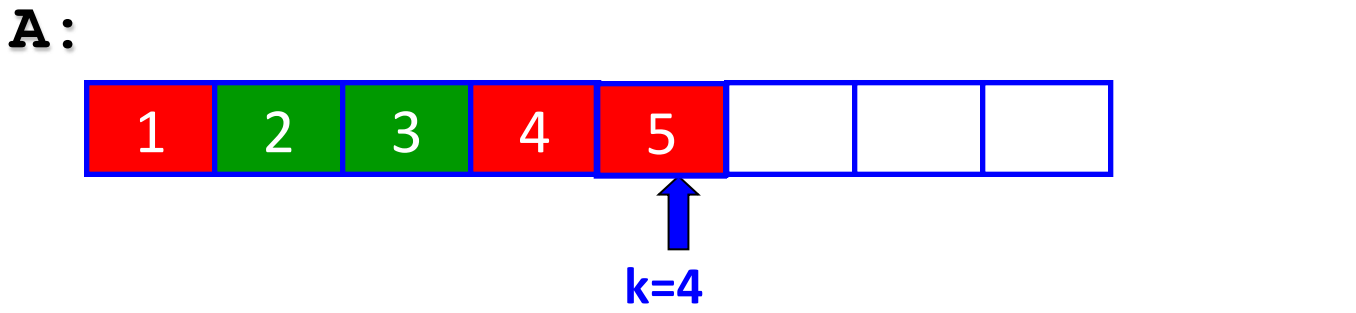


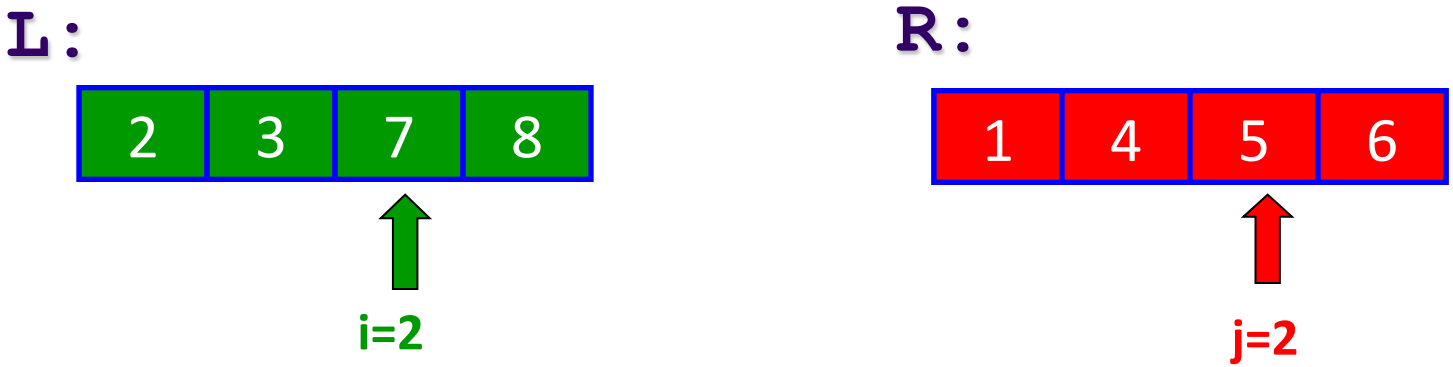
Merge-Sort: Merge Example



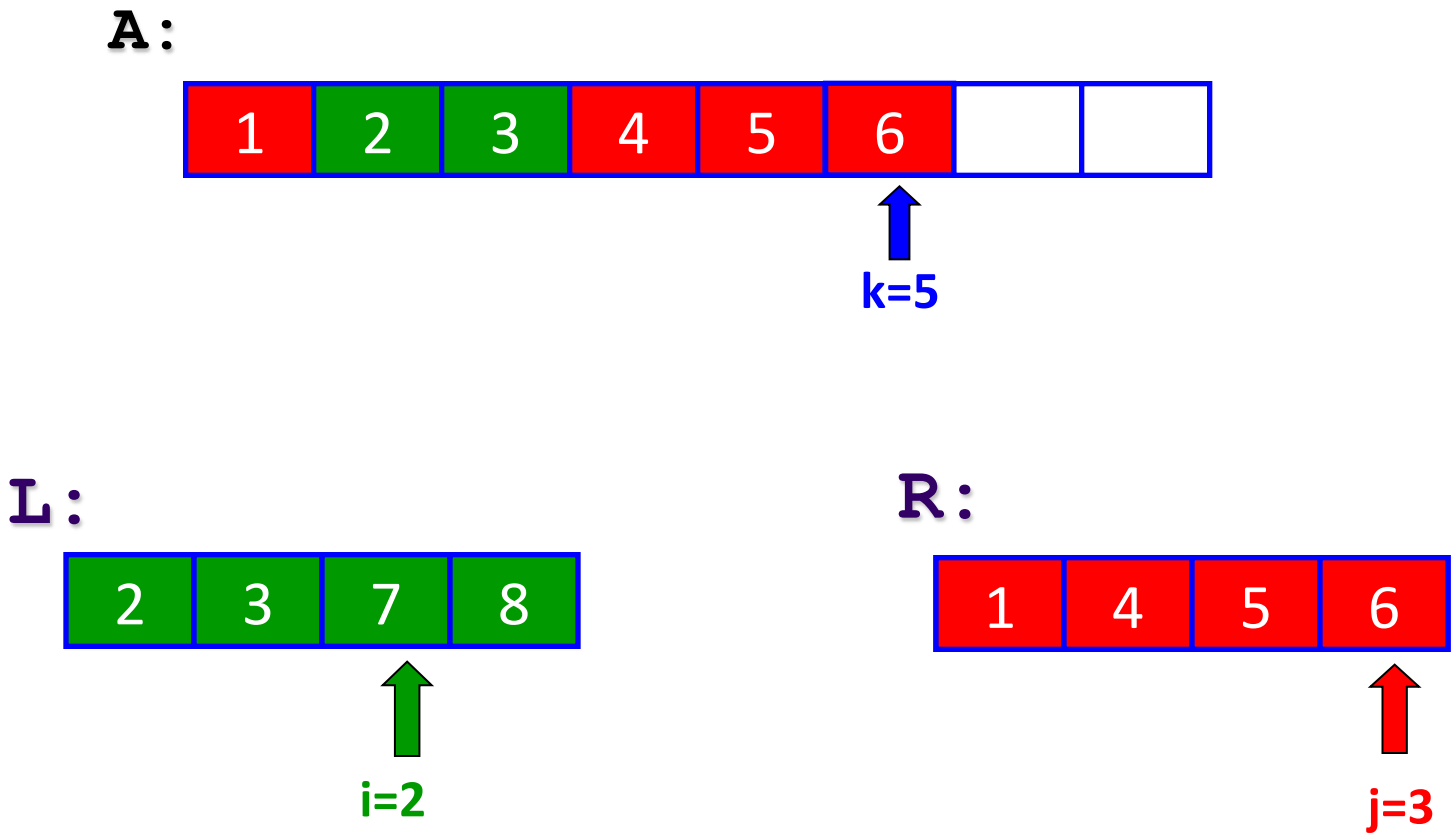


Merge-Sort: Merge Example

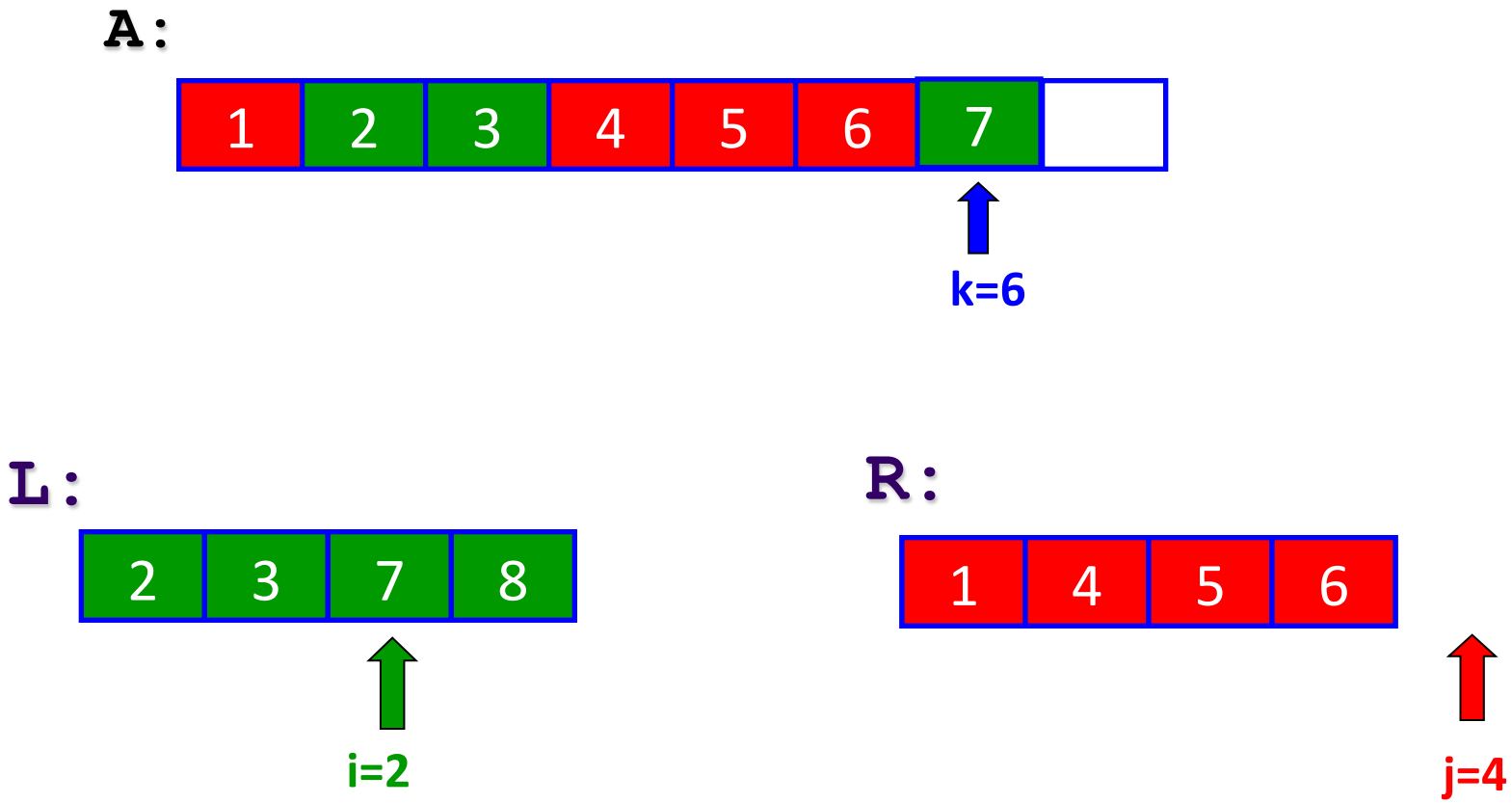




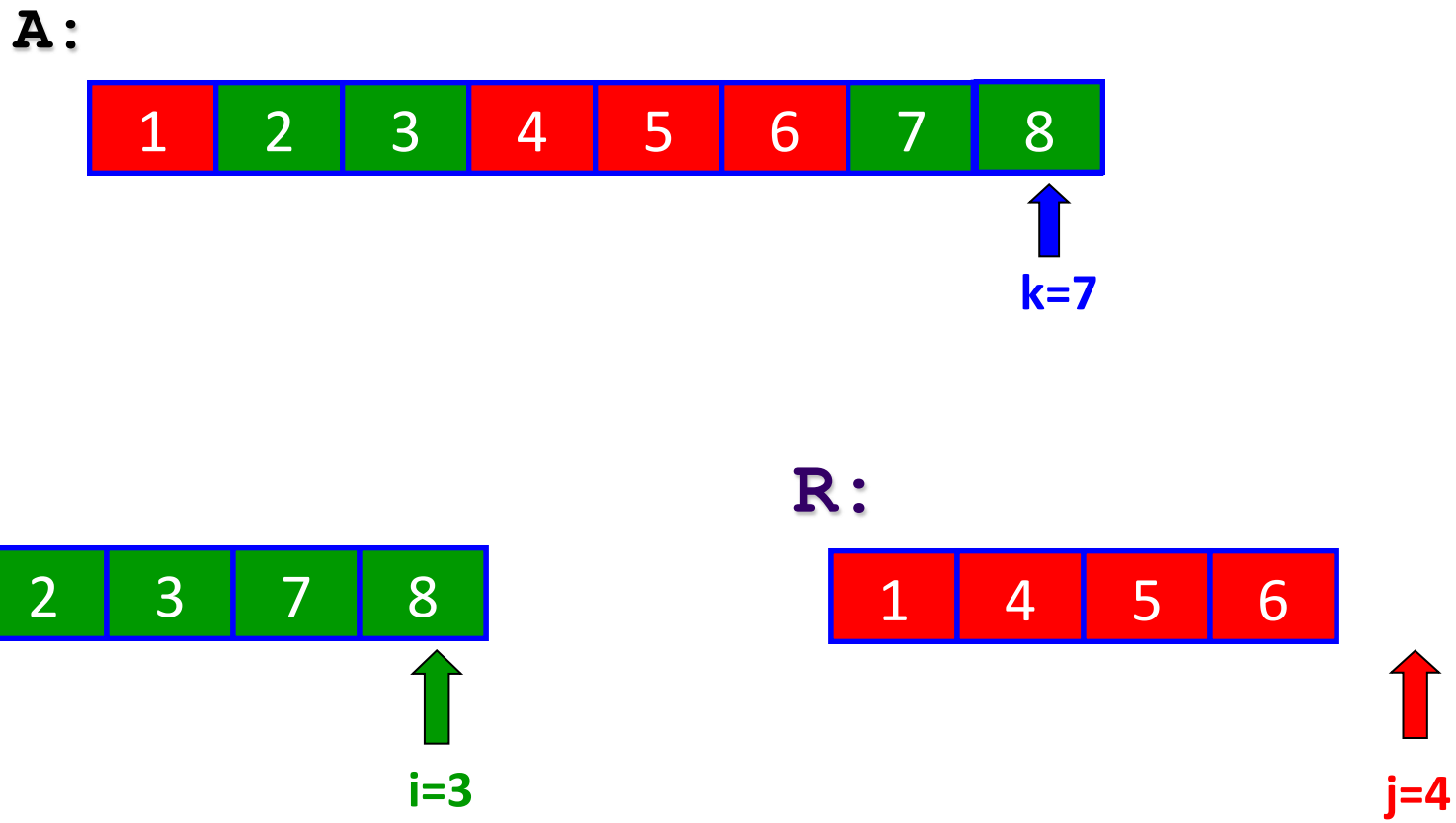
Merge-Sort: Merge Example



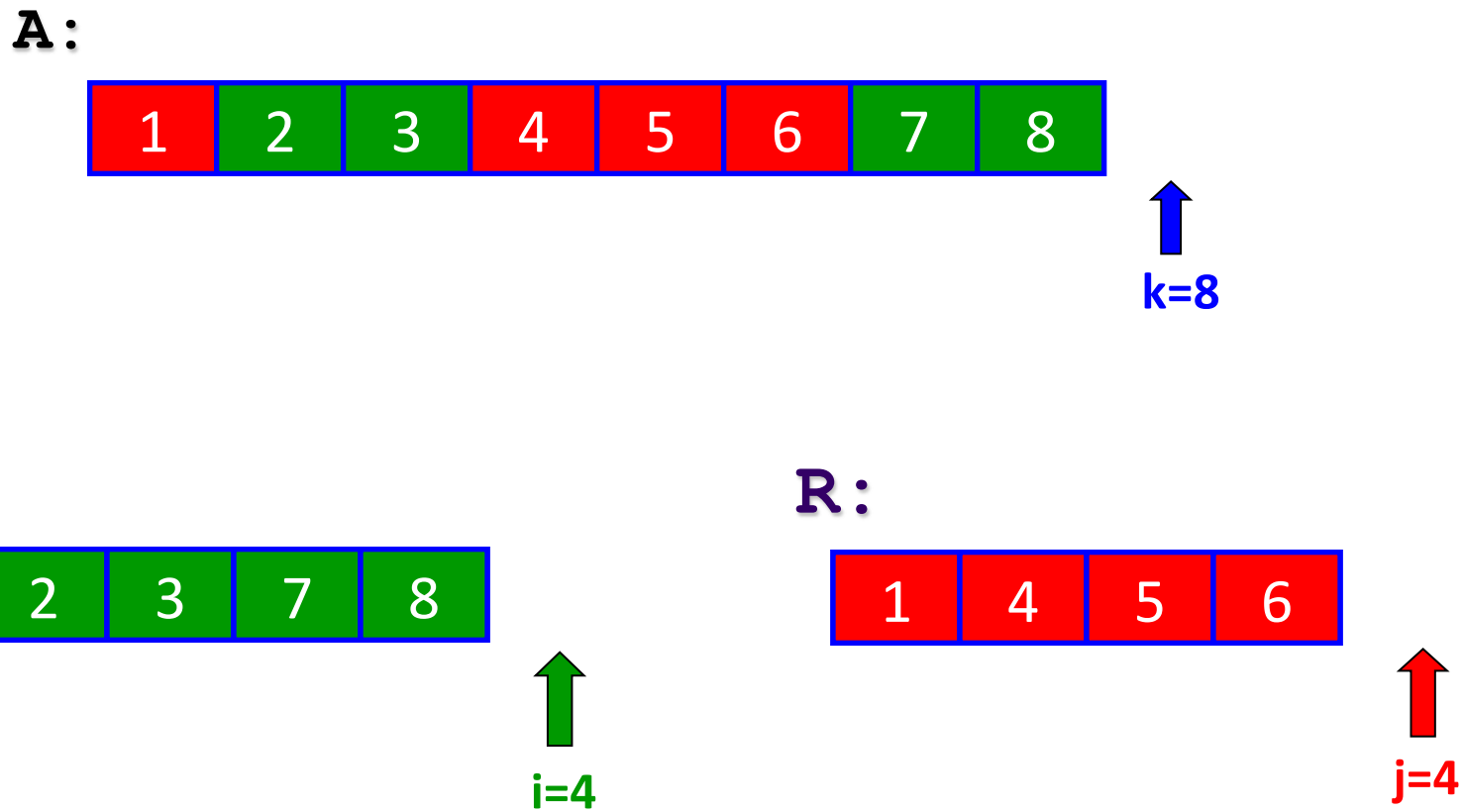
Merge-Sort: Merge Example



Merge-Sort: Merge Example



Merge-Sort: Merge Example



The merge procedure

```
public static void merge(int[] target,
          int[] left,
          int[] right)
{
    int i=0, j=0, k=0;

    while (i < left.length && j < right.length) {
        if (left[i] < right[j])
            target[k++] = left[i++];
        else
            target[k++] = right[j++];
    }


    while (i < left.length) target[k++] = left[i++];
    while (j < right.length) target[k++] = right[j++];
}
```

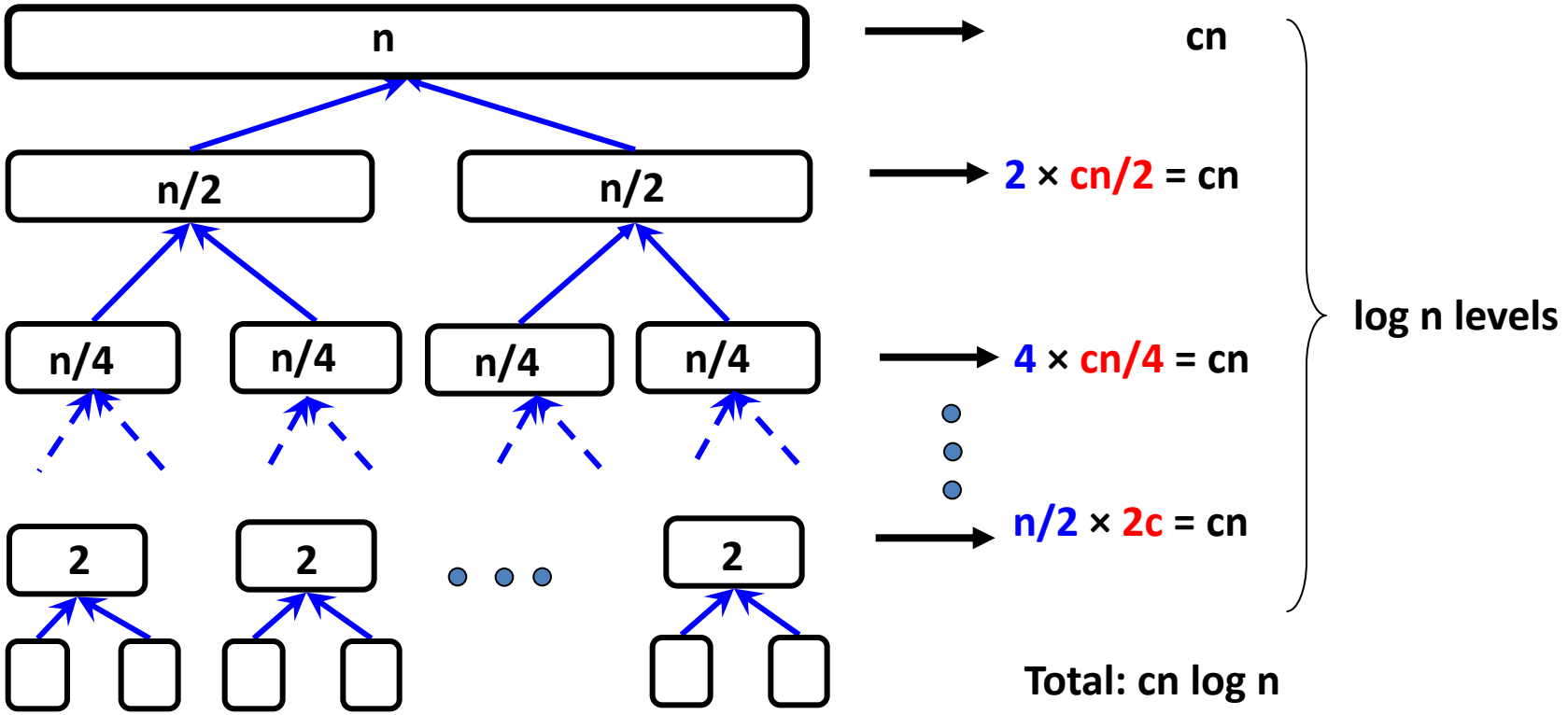
Running time of MergeSort

- The running time can be expressed as a recurrence:

$$T(n) = \begin{cases} \text{solving\_trivial\_problem} & \text{if } n = 1 \\ \text{num\_pieces } T(n/\text{subproblem\_size\_factor}) + \text{dividing} + \text{combining} & \text{if } n > 1 \end{cases}$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Merge-Sort Analysis



- Total running time: Θ(nlogn)
- Total Space: Θ (n)

Recurrence Tree / Repeated Substitution Method

$T(n) = 2T(n/2) + cn \quad n > 1$

$= 1 \quad n=1$

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

  $= 2\{2T(n/2^2) + c.n/2\} + cn$

  $= 2^2 T(n/2^2) + c.2n$

  $= 2^2\{2T(n/2^3) + c.n/2^2\} + c.2n$

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

  $= \ldots\ldots$

  $= 2^k T(n/2^k) + c.\mathbf{k}\cdot n$

  $= \ldots.$

  $= 2^{\log n} T(1) + c.(\mathbf{\log n})\cdot n$ when $n/2^k = 1 \Rightarrow k = \log_2 n$

  $= 2^{\log n} \cdot 1 + c.(\mathbf{\log n})\cdot n$

  $= \mathbf{n + c.n \log n}$ where $2^{\log n} = n$

- Therefore, $\mathbf{T(n) = O(n \log n)}$

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.

Properties of Merge Sort

- Worst case run time O(n log n) is *optimal* among comparison-based sorting algorithms
  - O(n log n) comparisons and item moves
- Space complexity $S(n) = S(n/2) + cn = \Theta(n)$
  - Big problem!
  - Can be made in-place, but too complex
  - Only O(log n) space (for recursion) when sorting linked list
- Is stable, not adaptive
- Question: write an iterative version of merge sort
  - Quite complicated (and not required for quiz/exam)
- Recursion > Iteration (slightly),

Why merge sort?

- Merge sort isn't an "in place" sort—it requires extra storage
- However, it doesn't require this storage "all at once"
- This means you can use merge sort to sort something that doesn't fit in memory—say, 300 million census records—then much of the data must be kept on backup media, such as a hard drive
- Merge sort is a good way to do this

Using merge sort for large data sets

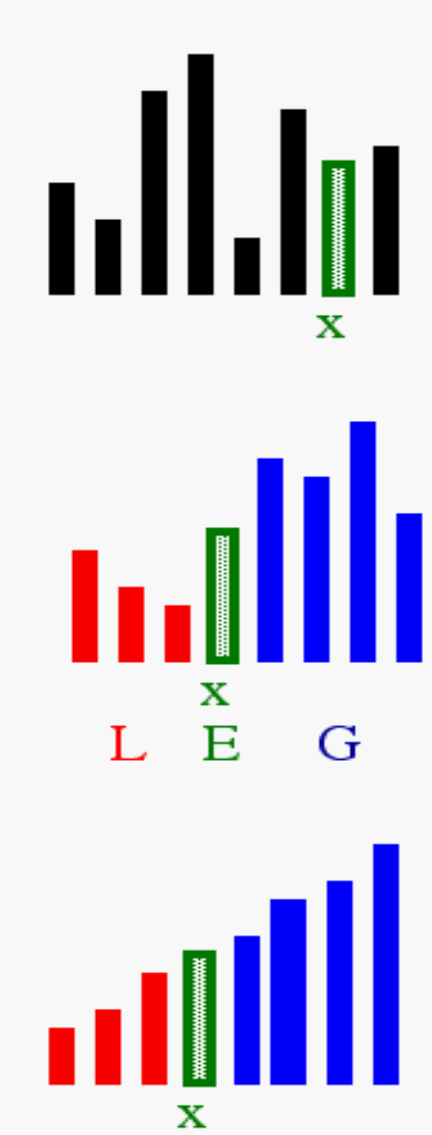
- *Very roughly,* here's how to sort large amounts of data:
  - Repeat:
    - Read in as much data as fits in memory
    - Sort it, using a fast sorting algorithm (quicksort may be a good choice)
    - Write out the sorted data to a new file
  - After all the data has been written into smaller, individually sorted files:
    - Read in the initial portion of each sorted file into individual arrays
    - Start merging the arrays
    - Whenever an array becomes empty, read in more data from its file
    - Every so often, write the destination array to the (one) final output file
- When you are done, you will have one (large) sorted file

Quick Sort: Idea

1) **Select**: pick an element

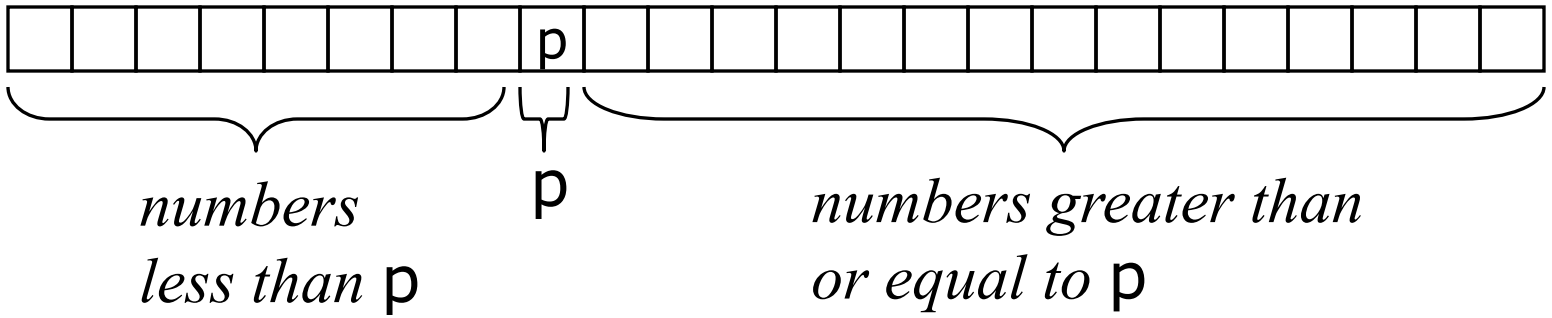
2) **Divide**: partition elements so that x goes to its final position E

3) **Conquer**: recursively sort left and right partitions

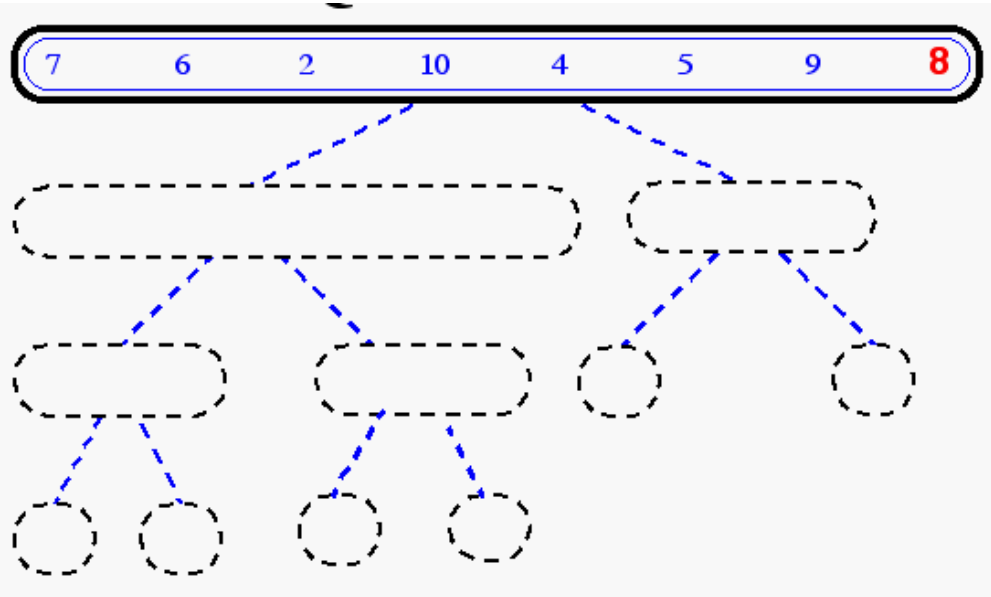


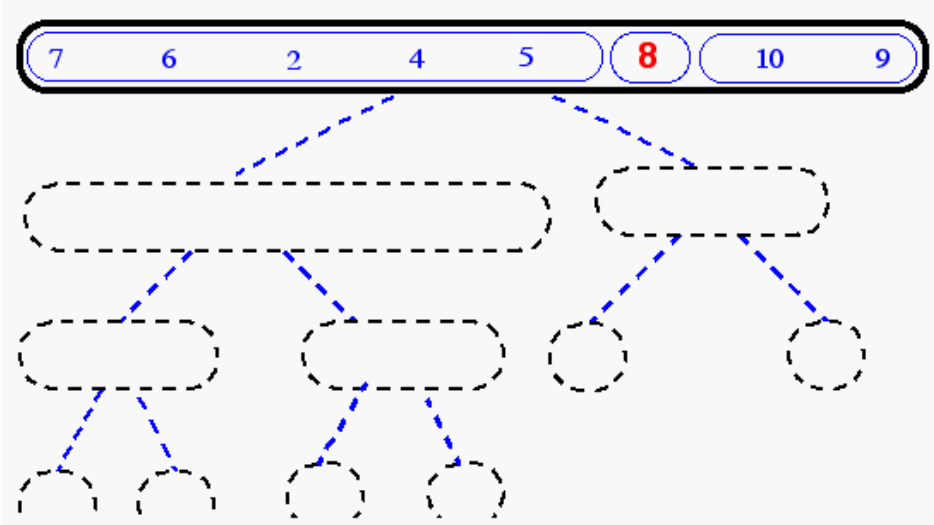
Quick Sort - Partitioning

- A key step in the Quick sort algorithm is partitioning the array
  - We choose some (any) number p in the array to use as a pivot
  - We partition the array into three parts:



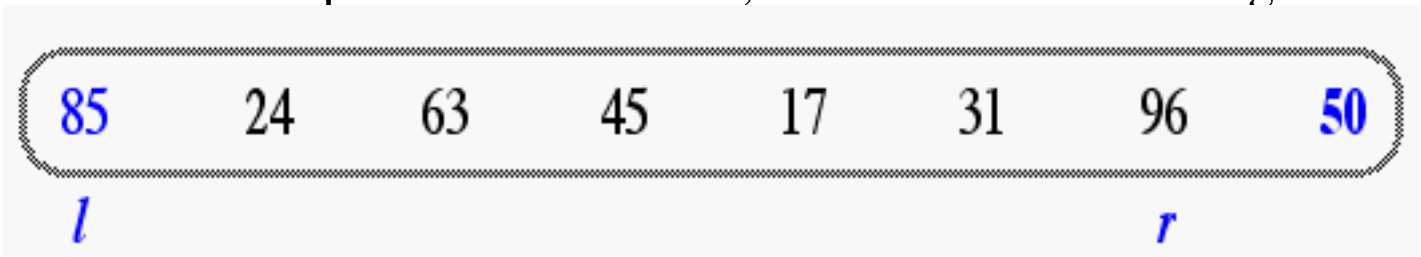
Quick Sort – Partitioning



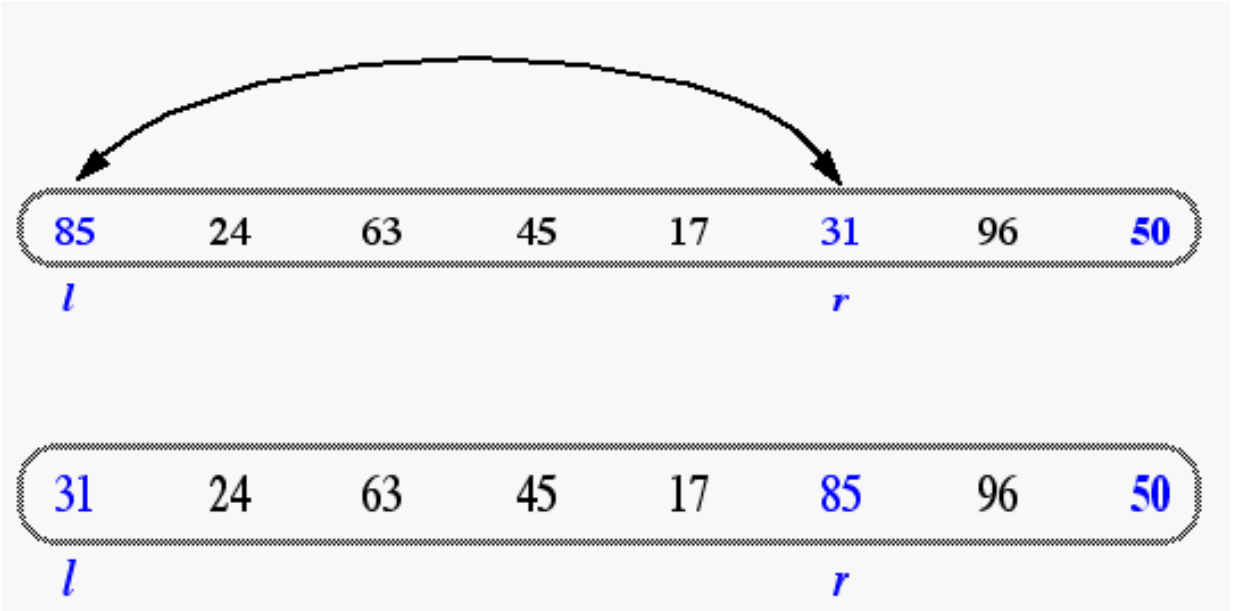


Quick Sort – Partitioning – algorithm

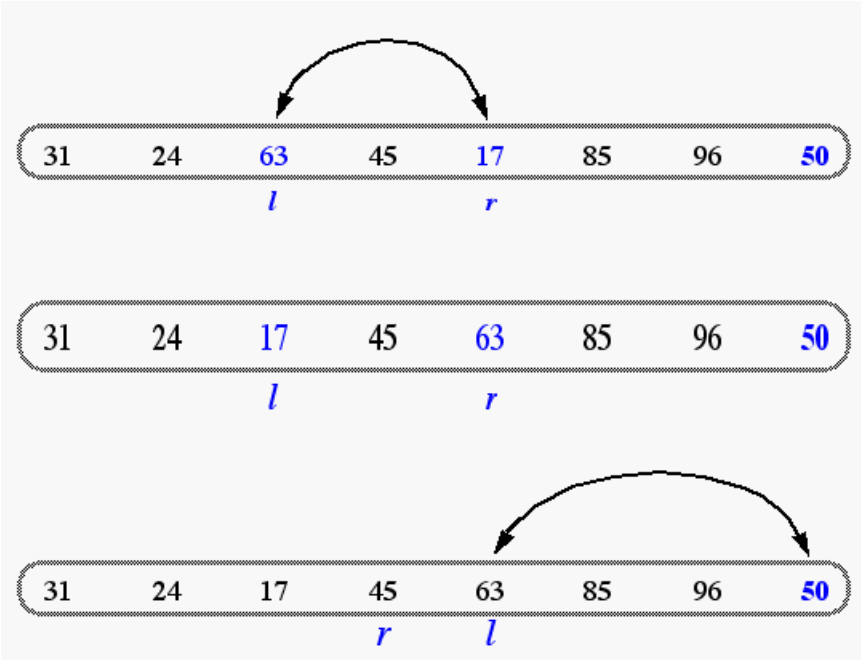
Index *l* scans the sequence from the left, and index *r* from the right.



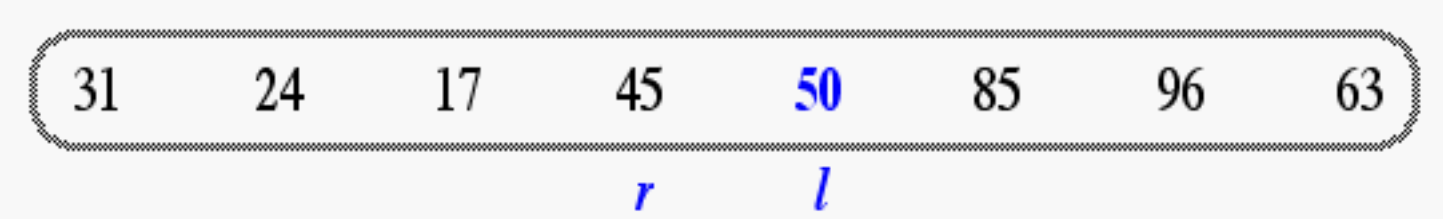
Increment *l* until *arr[l]* is larger than the pivot; and decrement *r* until *arr[r]* is smaller than the pivot.
Then swap elements indexed by *l* and *r*.
Repeat until whole array is processed.



Quick Sort – Partitioning – algorithm



A final swap with the pivot completes the partitioning.



Basic Quicksort

```
public static void recursive_qs(int[] arr, int left, int right)
{
    if (right <= left) return;


    // partition,
    int i=left-1, j=right;
    while (true) {
        while (arr[++i] <= arr[right])
            if (i == right) break;
        while (arr[--j] >= arr[right])
            if (j == left || j == i) break;
        if (j <= i) break;
        Sort.swap(arr, i, j);
    }
    if (i < right) Sort.swap(arr, i, right);


    // recursively sort the left & the right parts
    recursive_qs(arr, left, i-1);
    recursive_qs(arr, i+1, right);
}
```

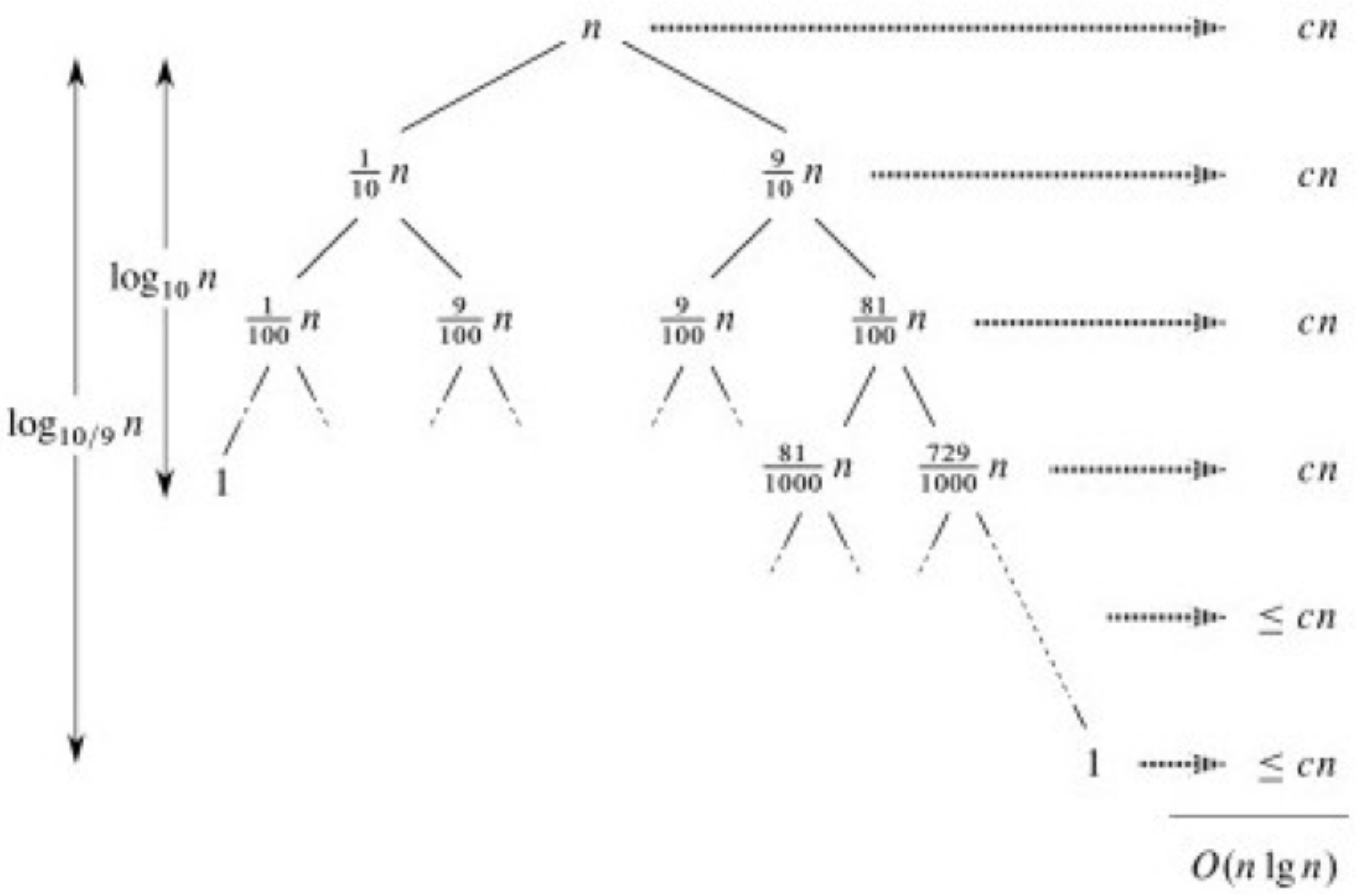
Properties

- Worst-case run time $\Omega(n^2)$
  - Which sequence of pivots lead to this?
- Not stable
- Not adaptive

- Why is it called "quick" sort then?
  - Work well on "average", $O(n \log n)$

Why O(*n* log *n*) can be expected?

$$T(n) = T(n/10) + T(9n/10) + cn$$



Making Quicksort Quick

- We can make it more likely to work well by randomizing the pivot!!!

- It is very slow on almost-equal inputs
  - Randomization can fix that too!

Randomized Quick sort

```
public static void recursive_rqs(int[] arr, int left, int right)
{
    if (right <= left) return;

    // pick a random pivot
    int m = (int)(Math.random() * (right-left+1));
    swap(arr, right, left+m);

    int i=left-1, j=right;
    while (true) {
        while (arr[++i] <= arr[right])
            if (i == right) break;
        while (arr[--j] >= arr[right])
            if (j == left || j == i) break;
        if (j <= i) break;
        Sort.swap(arr, i, j);
    }
    if (i < right) Sort.swap(arr, i, right);


    // recursively sort the left & the right parts
    recursive_rqs(arr, left, i-1);
    recursive_rqs(arr, i+1, right);
}
```

GENERIC SORTING ROUTINES

Selection Sort

```
public static void selection_sort(int[] array) {
    int i, j, k;
    for (i=0; i< array.length-1; i++) {
        j=i;
        for (k=i+1; k< array.length; k++)
            if (array[k] < array[j]) j=k;
        if (j!=i) swap(array, i, j);
    }
}
```

Generic Selection Sort

```
public static <T extends Comparable<T>> void
selection_sort_generic(T[] array) {

    int i, j, k;
    for (i=0; i< array.length-1; i++) {
        j=i;
        for (k=i+1; k< array.length; k++)

            if (array[k].compareTo(array[j]) < 0) j=k;


        if (j != i) swapGeneric(array, i, j);
    }
}
```

Summary

- Simple Sorting Algorithms
  - Bubble Sort, Selection Sort, and Insertion Sort.
- Optimal Comparison-based Sorting Algorithms
  - Merge Sort, Quick Sort
- Later we will also cover Heap Sort

Acknowledgement

- Douglas Wilhelm Harder.
  - Thanks for making an excellent set of slides for ECE 250 *Algorithms and Data Structures* course

- Prof. Hung Q. Ngo:
  - Thanks for those beautiful slides created for CSC 250 (Data Structures) course at UB.

- Many of these slides are taken from these two sources.