Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

Slides are prepared from
1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

Advance Sorting Algorithms

- We have seen sorting techniques in lecture 04
  - Bubble Sort - $O(n^2)$
  - Selection Sort - $O(n^2)$
  - Insertion Sort - $O(n^2)$ [best case $O(n)$]

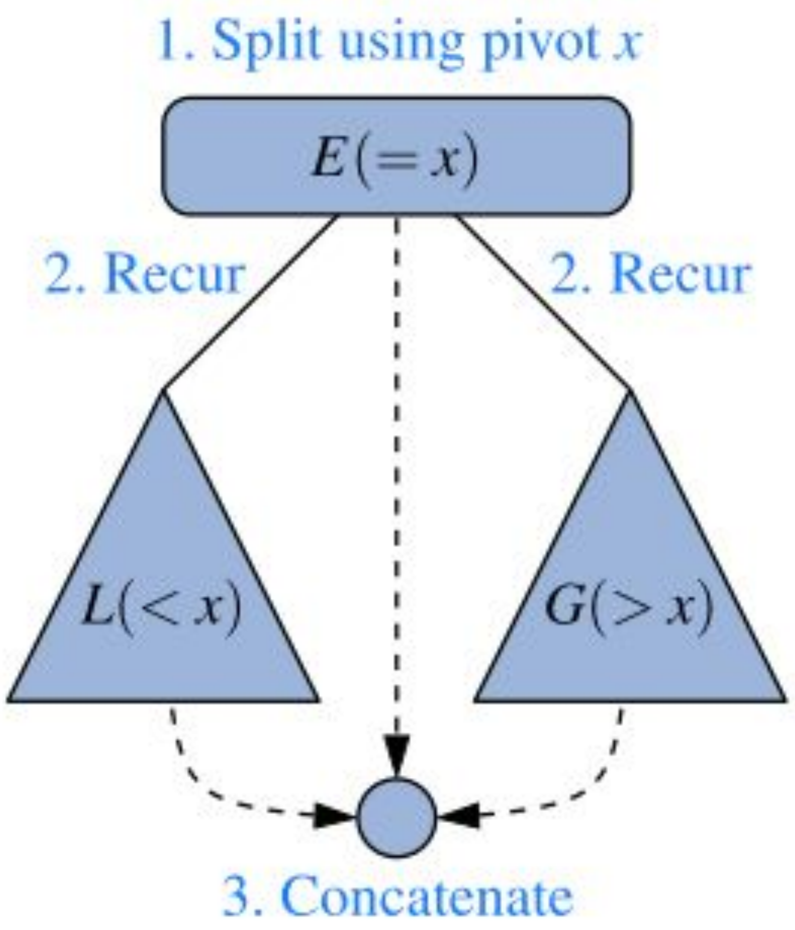
- Now we will see advanced sorting techniques
  - Heap Sort
  - Merge Sort
  - Quick Sort
  - Bucket Sort
  - Radix Sort

Quick-Sort

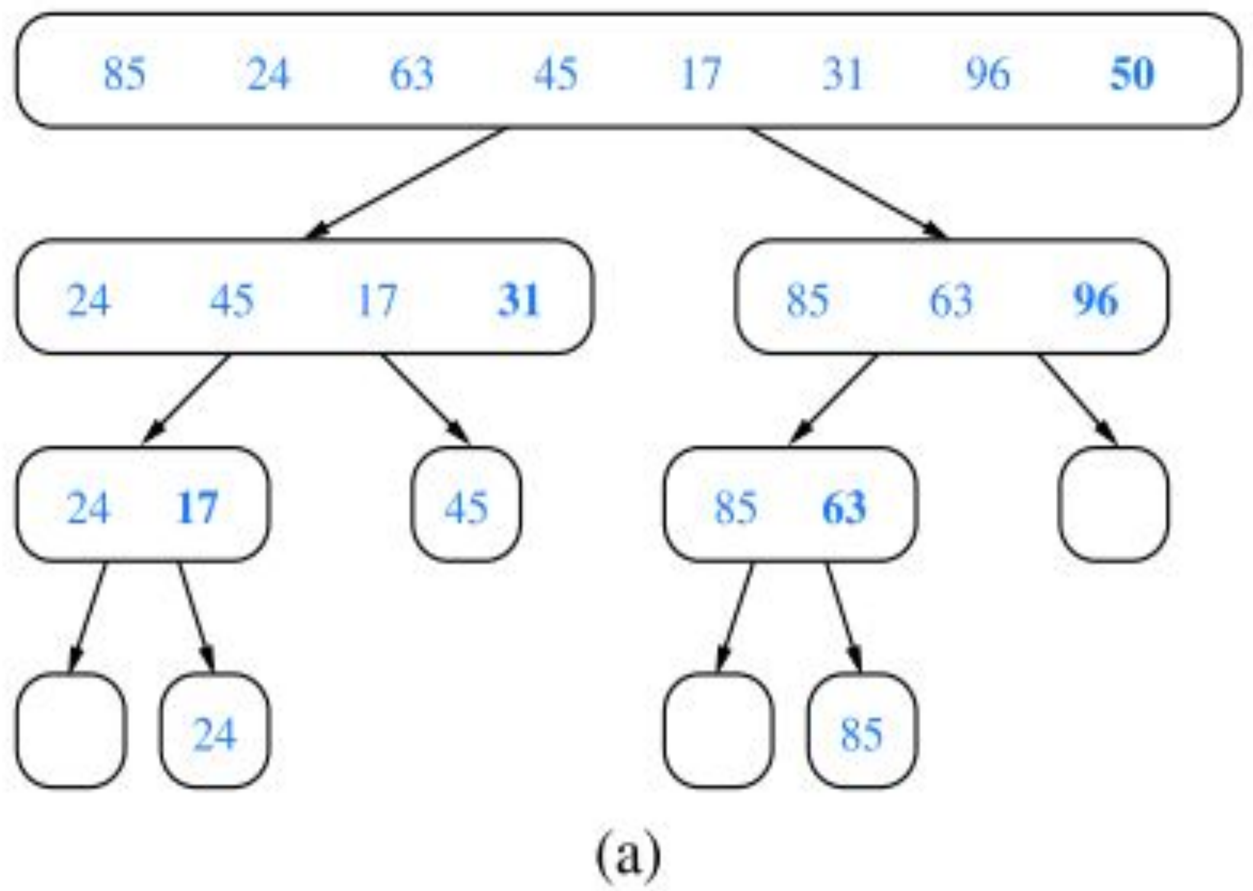
- Quicksort is undoubtedly the most popular sorting algorithm, and for good reason: In the majority of situations, it's the fastest, operating in *O(N*logN) time*. (This is only true for internal or in-memory sorting; for sorting data in disk files, other algorithms may be better.)
- Basically, the quicksort algorithm operates by *partitioning an array into two subarrays and then calling itself recursively to quicksort* each of these subarrays
- This algorithm is also based on the ***divide-and-conquer paradigm***

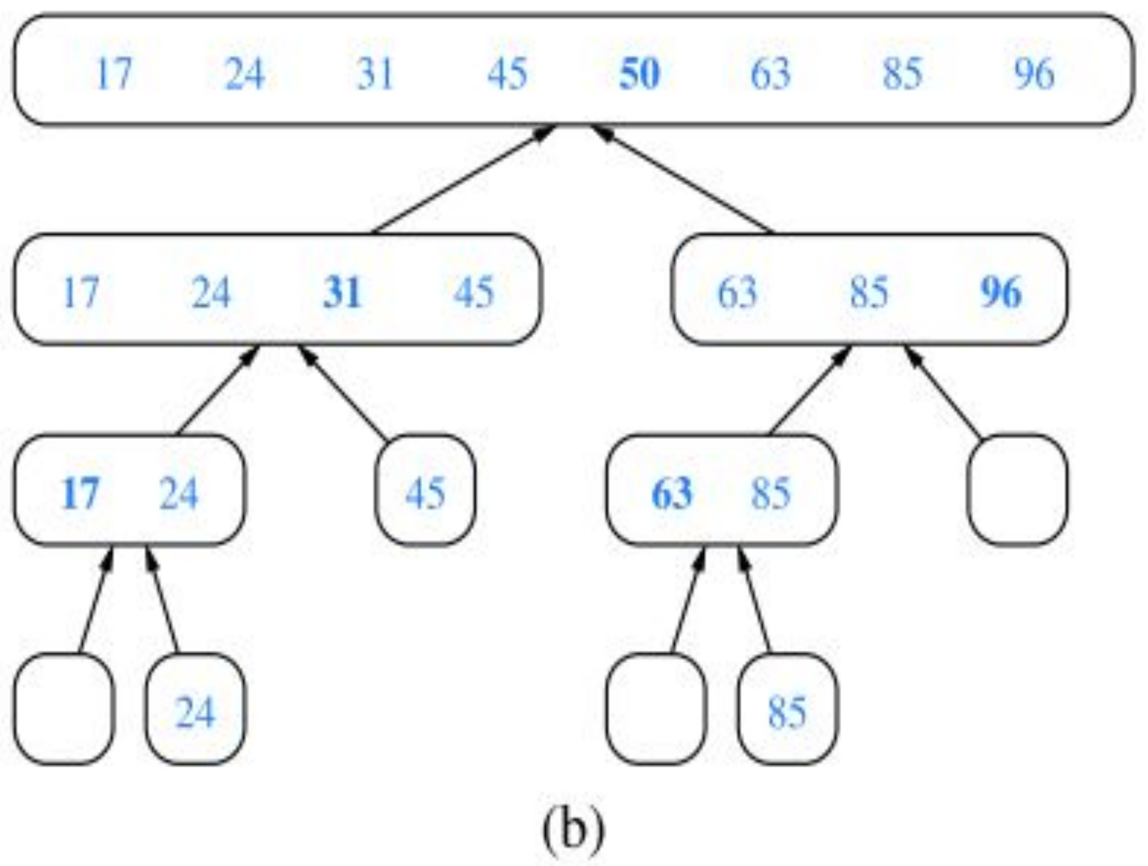
High-Level Description of Quick-Sort

- In particular, the quick-sort algorithm consists of the following three steps
  - **Divide:** If ***S*** has at least two elements (nothing needs to be done if S has zero or one element), select a specific element ***x*** from ***S***, which is called the **pivot**. As is common practice, choose the pivot ***x*** to be the last element in ***S***. Remove all the elements from ***S*** and put them into three sequences:
    - ***L***, storing the elements in ***S*** **less than** ***x***
    - ***E***, storing the elements in ***S*** **equal to** ***x***
    - ***G***, storing the elements in ***S*** **greater than** ***x***
  - **Conquer:** Recursively sort sequences ***L*** and ***G***
  - **Combine:** Put back the elements into ***S*** in order by first inserting the elements of ***L***, then those of ***E***, and finally those of ***G***



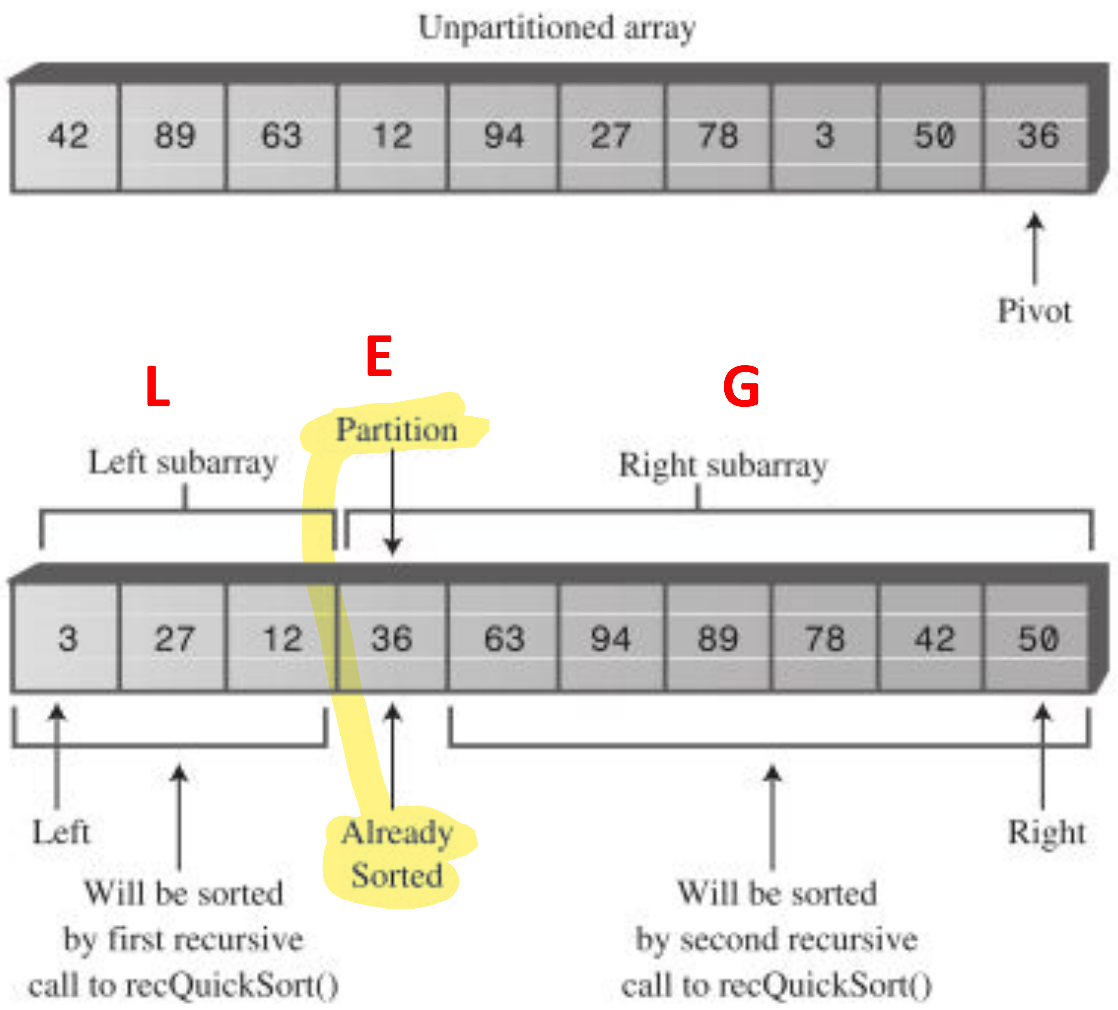
Quick-sort tree



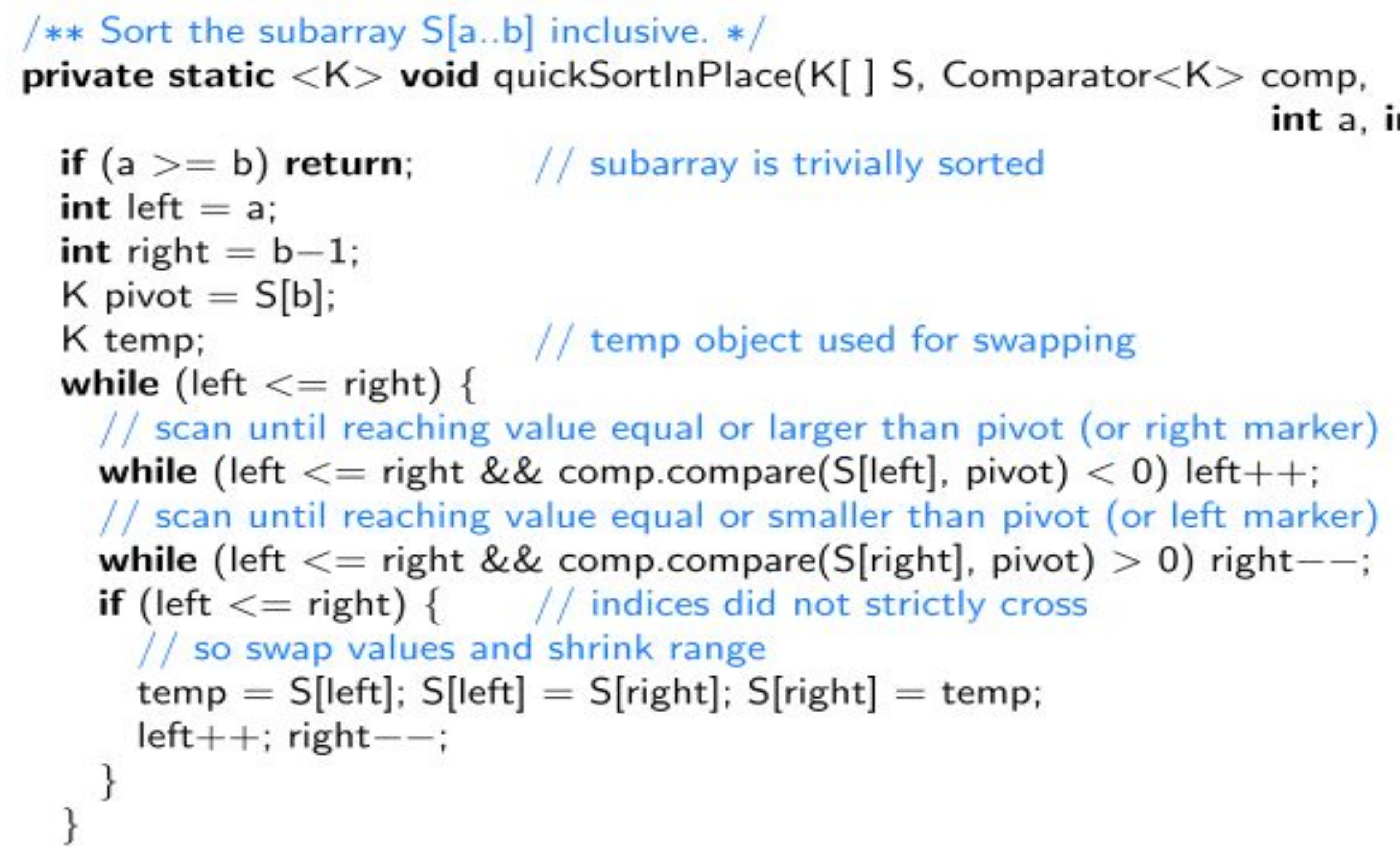


Quick Sort

- After a partition, all the items in the left subarray are smaller than all those on the right. If we then sort the left subarray and sort the right subarray, the entire array will be sorted
- Quick-sort of an array-based sequence can be adapted to be **in-place**, which uses only a small amount of memory
- **In-place sort** uses the input sequence itself to store the subsequences for all the recursive calls
- In-place quick-sort modifies the input sequence using element swapping and does not explicitly create subsequences
- Instead, a subsequence of the input sequence is implicitly represented by a range of positions specified by a leftmost **index *a*** and a rightmost **index *b***

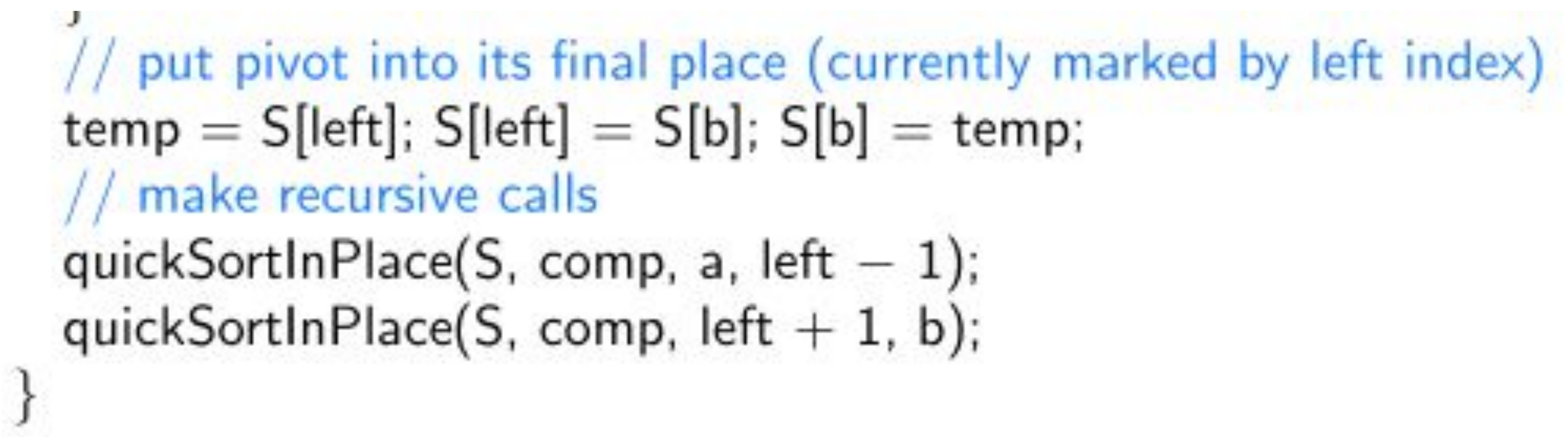


In-place Quick Sort



```
/** Sort the subarray S[a..b] inclusive. */
private static <K> void quickSortInPlace(K[ ] S, Comparator<K> comp,
                                                                  int a, int b) {
  if (a >= b) return;            // subarray is trivially sorted
  int left = a;
  int right = b−1;
  K pivot = S[b];
  K temp;                        // temp object used for swapping
  while (left <= right) {
    // scan until reaching value equal or larger than pivot (or right marker)
    while (left <= right && comp.compare(S[left], pivot) < 0) left++;
    // scan until reaching value equal or smaller than pivot (or left marker)
    while (left <= right && comp.compare(S[right], pivot) > 0) right−−;
    if (left <= right) {         // indices did not strictly cross
      // so swap values and shrink range
      temp = S[left]; S[left] = S[right]; S[right] = temp;
      left++; right−−;
    }
  }
```

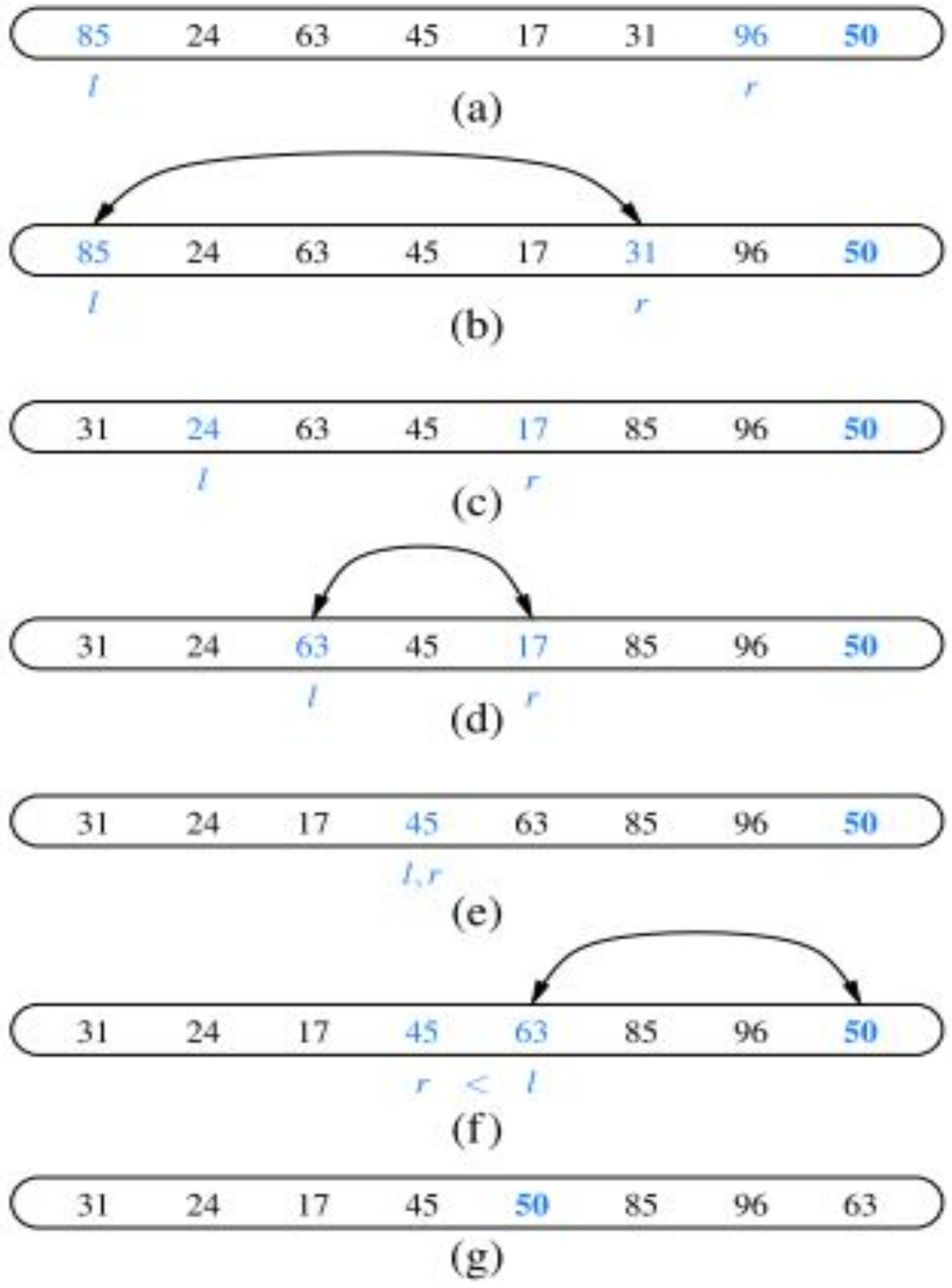
In-place Quick Sort



```
  // put pivot into its final place (currently marked by left index)
  temp = S[left]; S[left] = S[b]; S[b] = temp;
  // make recursive calls
  quickSortInPlace(S, comp, a, left − 1);
  quickSortInPlace(S, comp, left + 1, b);
}
```

In-place Quick Sort

- The divide step is performed by scanning the array simultaneously using local variables ***left***, which *advances forward*, and ***right***, which *advances backward*, swapping pairs of elements that are in reverse order
- When these two indices pass each other, the division step is complete and the algorithm completes by recurring on these two sublists
- There is *no explicit "***combine***" step*, because the concatenation of the two sublists is implicit to the in-place use of the original list



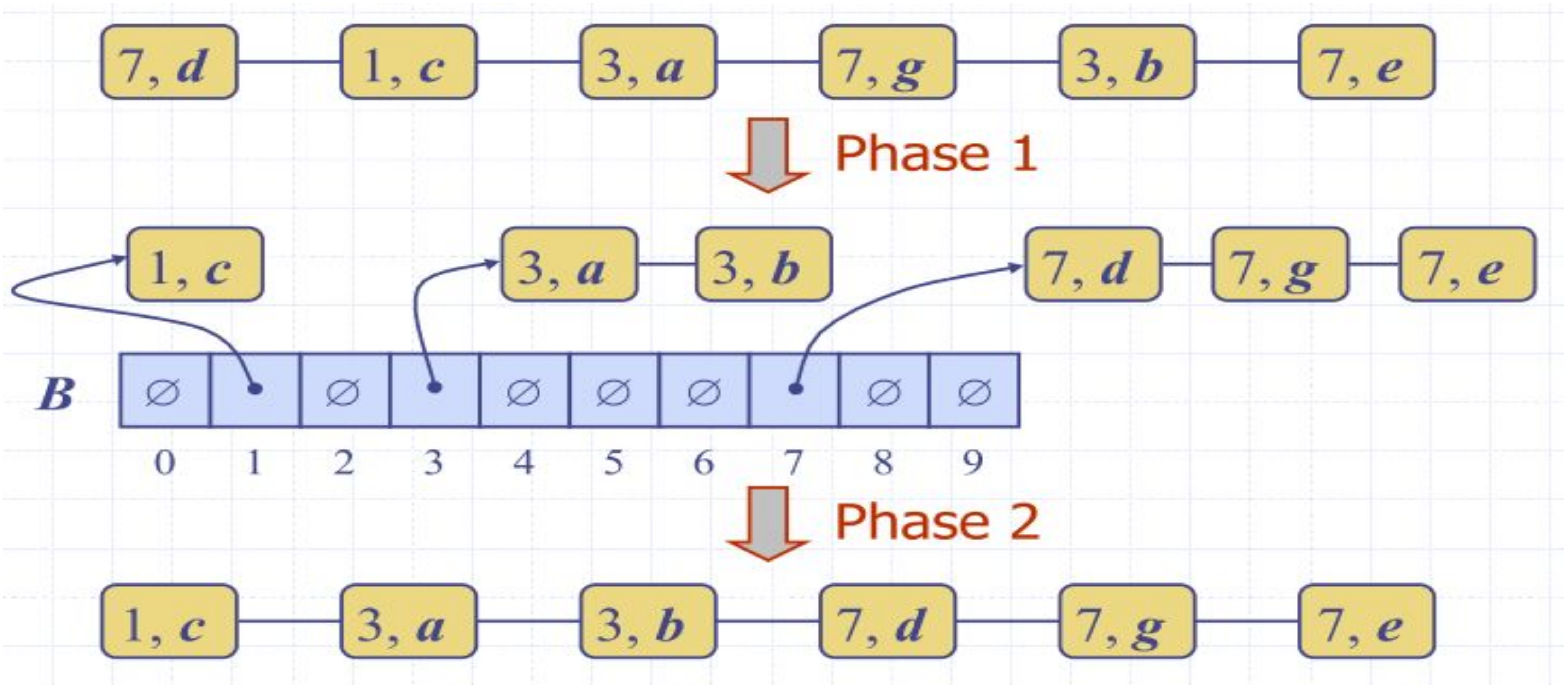
Linear-Time Sorting: Bucket-Sort and Radix-Sort

- we have seen that $\Omega(n\log n)$ time is necessary, in the worst case, to sort an n-element sequence with a comparison-based sorting algorithm
- Whether are there other kinds of sorting algorithms that can be designed to run asymptotically faster than **$O(n\log n)$** time?
- Interestingly, such algorithms exist, but they require special assumptions about the input sequence to be sorted
- Now we will consider the problem of sorting a sequence of entries, each a key-value pair, where the keys have a *restricted type*
    - **Bucket sort**
    - **Radix sort**

Linear-Time Sorting: Bucket-Sort

- Consider a sequence ***S*** of ***n*** entries whose keys are integers in the range [**0,N−1**], for some integer **N ≥ 2**, and suppose that ***S*** should be sorted according to the keys of the entries
- In this case, it is possible to sort ***S*** in **O(n+N)** time
- If **N** is **O(n)**, then we can sort ***S*** in **O(n)** time
- Bucket-sort is *not based on comparisons*, but on using *keys as indices into a bucket array B* that has cells indexed from 0 to N−1
- An entry with ***key k*** is placed in the "bucket" **B[*k*]**, which itself is a sequence (of entries with key k)
- After inserting each entry of the input sequence ***S*** into its bucket, we can put the entries back into ***S*** in sorted order by enumerating the contents of the buckets **B[*0*]**,**B[*1*]**, . . . ,**B[*N−1*]** in order

Linear-Time Sorting: Bucket-Sort



Stable Sort Property: The relative order of any two items with the same key is preserved after the execution of the algorithm

Bucket-Sort - Implementation

```
Algorithm bucketSort($S$):
    Input: Sequence $S$ of entries with integer keys in the range $[0, N-1]$
    Output: Sequence $S$ sorted in nondecreasing order of the keys
  let $B$ be an array of $n$ sequences, each of which is initially empty
  for each entry $e$ in $S$ do
    let $k$ denote the key of $e$
    remove $e$ from $S$ and insert it at the end of bucket (sequence) $B[k]$
  for $i = 0$ to $n-1$ do
    for each entry $e$ in sequence $B[i]$ do
      remove $e$ from $B[i]$ and insert it at the end of $S$
```

Linear-Time Sorting: Radix Sort

- The radix sort disassembles the key into digits and arranges the data items according to the value of the digits
- The word radix means the base of a system of numbers
- Ten is the radix of the decimal system and 2 is the radix of the binary system
- The sort involves examining each digit of the key separately, starting with the 1s (least significant) digit
- Radix-sort is applicable to tuples where the keys in each dimension i are integers in the range [0, N - 1]

Linear-Time Sorting: Radix Sort

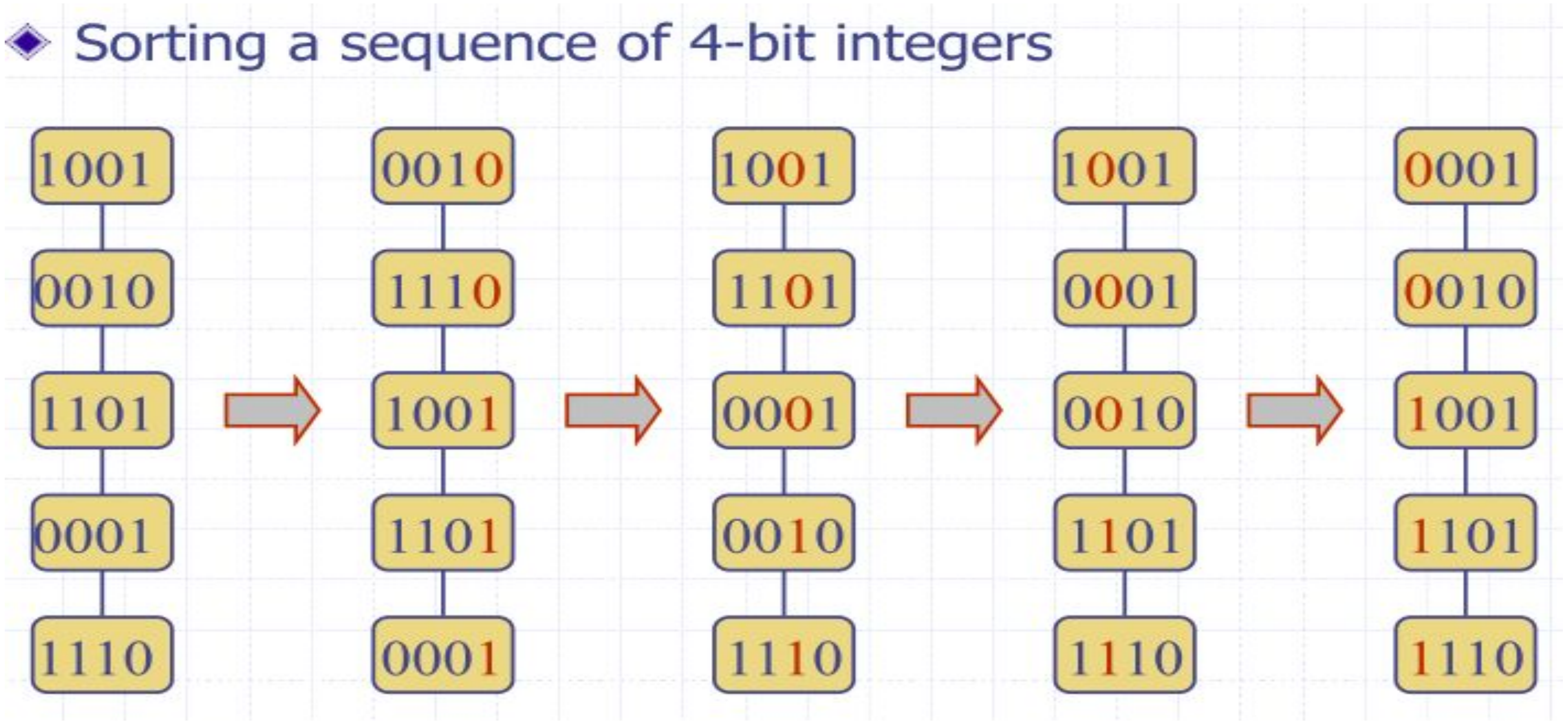
- We consider the radix sort in terms of normal base-10 arithmetic
- Algorithm steps are as follows:
  - All the data items are divided into 10 groups, according to the value of their 1s digit
  - These 10 groups are then reassembled: All the keys ending with 0 go first, followed by all the keys ending in 1, and so on up to 9. We'll call these steps a sub-sort
  - In the second sub-sort, all data is divided into 10 groups again, but this time according to the value of their 10s digit. This must be done without changing the order of the previous sort
  - Again the 10 groups are recombined, those with a 10s digit of 0 first, then those with a 10s digit of 1, and so on up to 9
  - This process is repeated for the remaining digits. If some keys have fewer digits than others, their higher-order digits are considered to be 0

Linear-Time Sorting: Radix Sort

- An example, using seven data items, each with three digits
- Leading zeros are shown for clarity

```
421 240 035 532 305 430 124                 // unsorted array
(240 430) (421) (532) (124) (035 305)       // sorted on 1s digit
(305) (421 124) (430 532 035) (240)         // sorted on 10s digit
(035) (124) (240) (305) (421 430) (532)     // sorted on 100s digit
035 124 240 305 421 430 532                 // sorted array
```

Radix Sort - Example



Lab Assignment

- Implement advance sorting techniques for increasing and decreasing order of elements.
  - Heap Sort
  - Merge Sort
  - Quick Sort
  - Bucket Sort
  - Radix Sort

- Use student information as a data
  - registrationID
  - name
  - address
  - grade