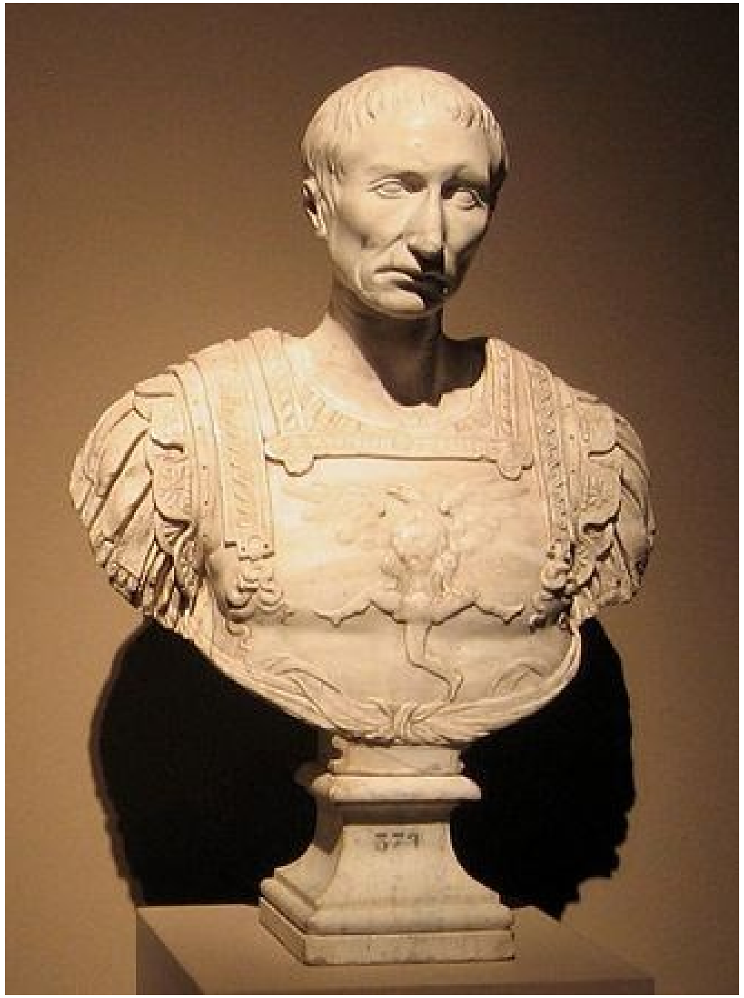
Data Structure

Sorting

Shin Hong

25 May 2023



DS&A. Chapter 11.1 Merge-sort
DS&A. Chapter 11.2 Quick-sort

Merge-sort (1/2)

- a sorting problem is to produce an ordered representation of a given sequence of objects stored in a linked list or an array
  - according to a given comparator that defines a total order on the given objects

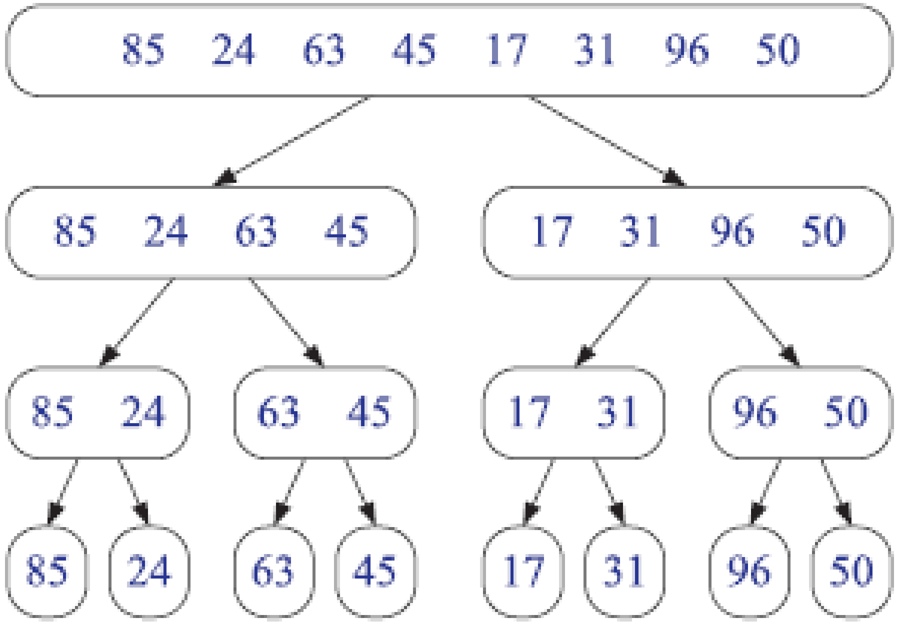
- merge-sort is based on an algorithmic design pattern called divide-and-conquer which typically consists of the followings:
  - divide the input data into two or more disjoint subsets, and recursively solve the subproblems with the subsets
  - take the solutions of the subproblems and merge them into a solution to the original program

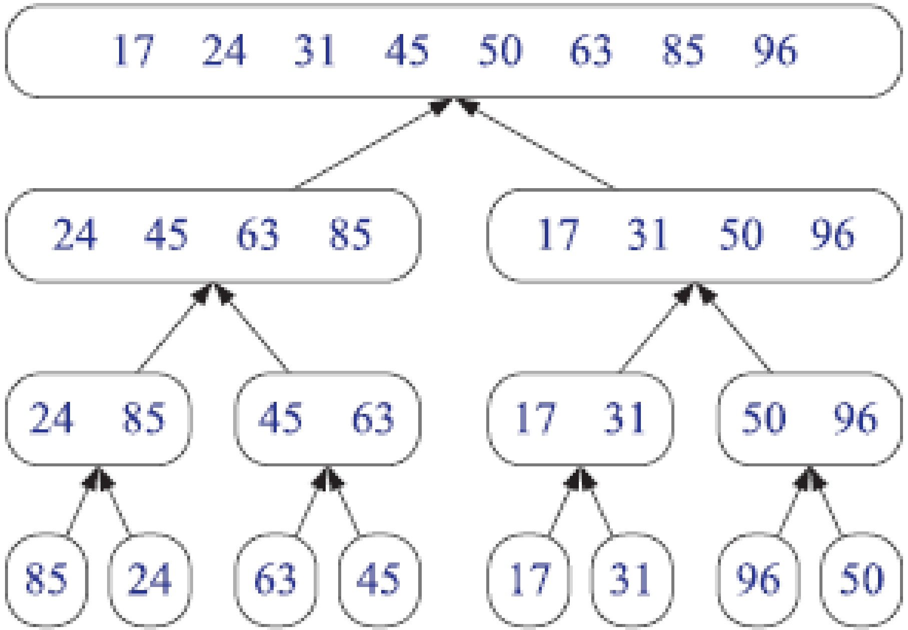
Merge-sort (2/2)

- To sort a sequence S with n elements:

  1. Return S immediately if n is zero or one.
     Otherwise, create S_1 and S_2 by dividing S evenly
  2. Sort S_1 and S_2, recursively
  3. Put back the elements into S by merging S_1 and S_2

Example


85 24 63 45 17 31 96 50
85 24 63 45
17 31 96 50
85 24
63 45
17 31
96 50
85
24
63
45
17
31
96
50


17 24 31 45 50 63 85 96
24 45 63 85
17 31 50 96
24 85
45 63
17 31
50 96
85
24
63
45
17
31
96
50

Running Time of Merge-sort

- suppose that n is a power of 2 (i.e., $n = 2^m$)

- merge-sort of 2^k elements is invoked for 2^{m-k} times
  - at merge-sort execution, it takes linear time to divide the given elements, and merge the two sorting results into one
  - the maximum depth of recursive call is m

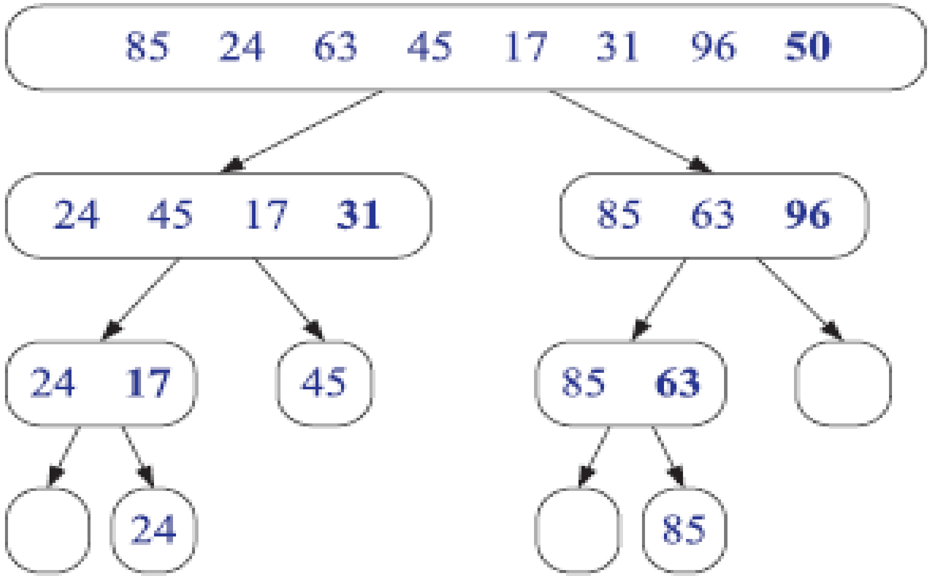
- the time complexity is $O(n \log n)$

Quick-sort

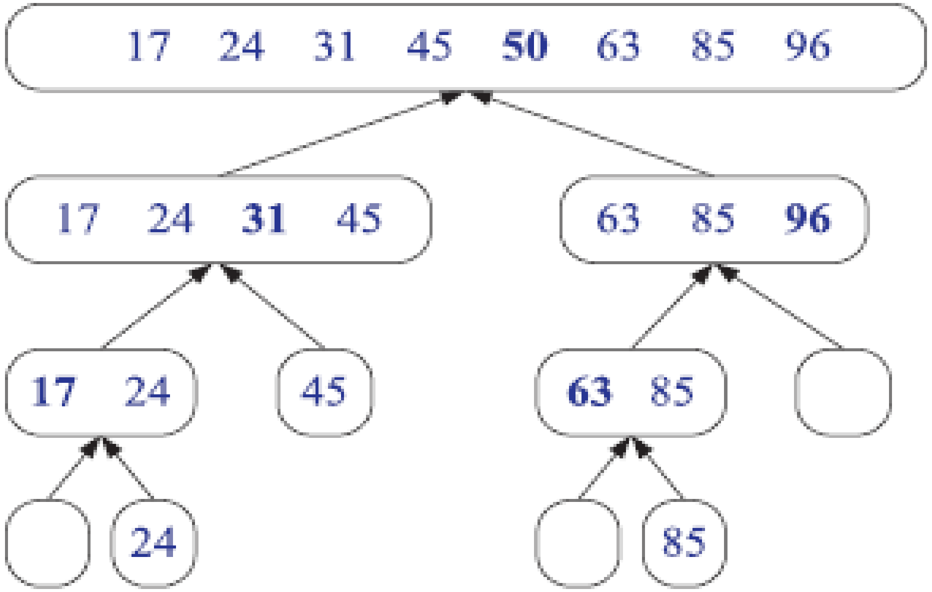
- To sort a sequence *S* with *n* elements:

  1. if *S* has at least two elements, select a specific element *x* from *S* as the pivot
  2. Remove all elements from *S* and put them into three sequences:
     - *L*, storing the elements less than *x*
     - *E*, storing the element equal to *x*
     - *G*, storing the element greater *x*
  3. Recursively sort L and G
  4. Put back the elements into S in order of *L*, *E*, and *G*

Example



85 24 63 45 17 31 96 50
24 45 17 31
85 63 96
24 17
45
85 63
24
85



17 24 31 45 50 63 85 96
17 24 31 45
63 85 96
17 24
45
63 85
24
85

Running Time of Quick-sort

- Time spent at each quick-sort is linear of the number of elements

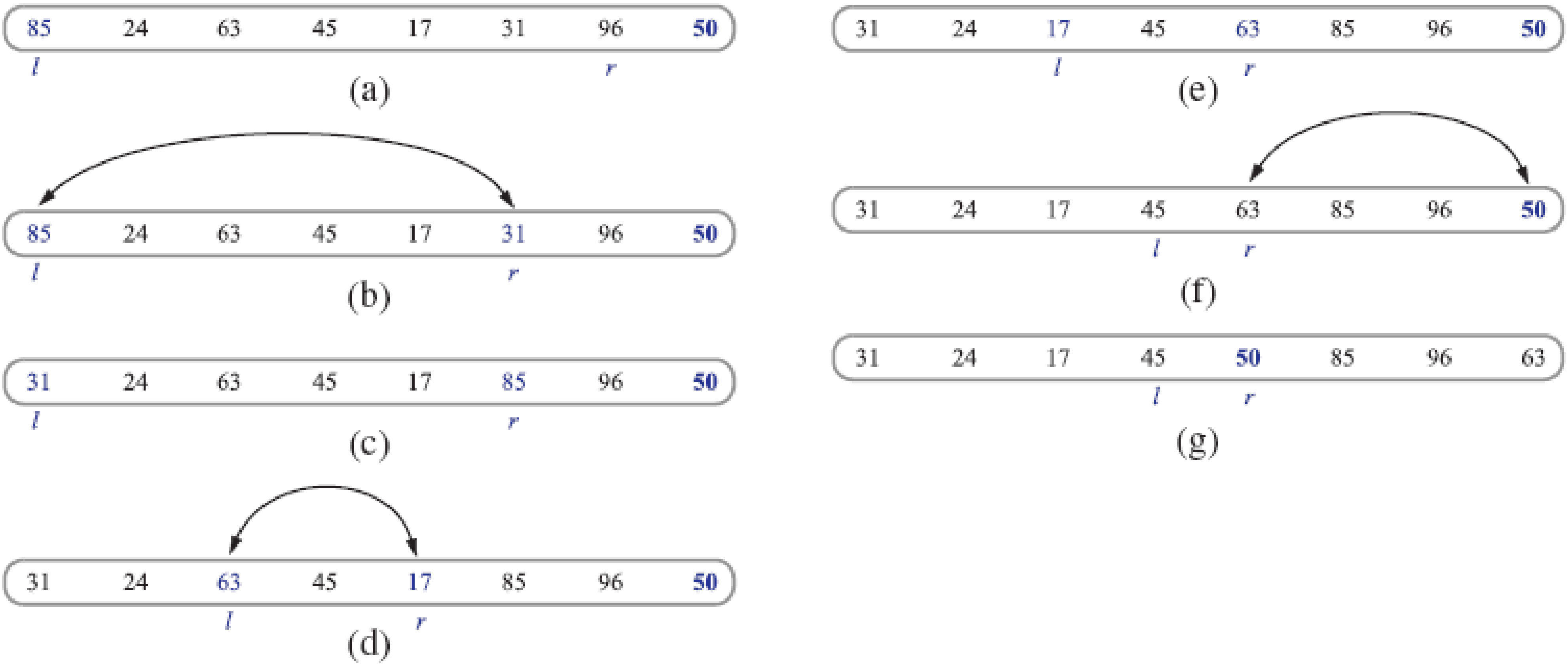
- In worst case, the number of element is reduced by one at each recursion and the recursion continues up to n-1 times, thus it takes $O(n^2)$
  - pivot does not divide given elements evenly

- The best case is where at each quick-sort invocation, the pivot divides the given elements into two subsequences of an equal size, thus it takes $O(n\log n)$

Randomized Quick-sort

- The worst case happens if a pivot does not divide the given input sequence every time
    - e.g., initially, given elements are arranged in decreasing order
- Instead of picking a pivot as the last element, select an element at a random index as the pivot
    - then, the expected running time is O(n log n)

In-place Quick-sort



Bucket-sort

- Sort n entries whose keys are integers in the range $[0, N - 1]$ in $O(n + N)$
  - assume that $N << n$
  - it is not based on comparison
- Allocate a bucket array B that has cells indexed from 0 to $N – 1$, place each entry with key k in $B[k]$, and then enumerate entries in B in order

Stability of Sorting

- Let S = ((k_0, x_0), ..., ($k_{n\text{-}1}$, $x_{n\text{-}1}$)) be a sequence of entries
- We say that a sorting algorithm is stable if, for any two entries (k_i, x_i) and (k_j, x_j) of S such that $k_\mathrm{i} = k_j$ and i < j, (k_i, x_i) precedes (k_j, x_j) after sorting

Radix-sort

- Suppose that a key is d digits each of which is an integer in the range $[0, N\text{-}1]$

- Iteratively run bucket-sort with i-th digit as key, for $i = d, d-1, \ldots, 1$
  - exploit the stability of bucket-sort
  - last digit is least-significant digit
  - e.g., 233, 46, 82, 2, 958, 33, 143, 67, 146, 92

  002 033 046 067 082 092 143 146 233 958

- Radix-sort takes $O(d(n+N))$