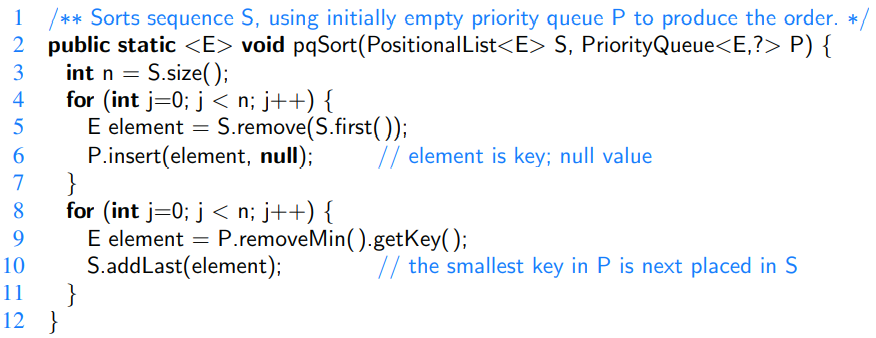
Week4

Sorting with a Priority Queue

The algorithm for sorting a sequence *S* with a priority queue *P* is quite simple and consists of the following two phases:

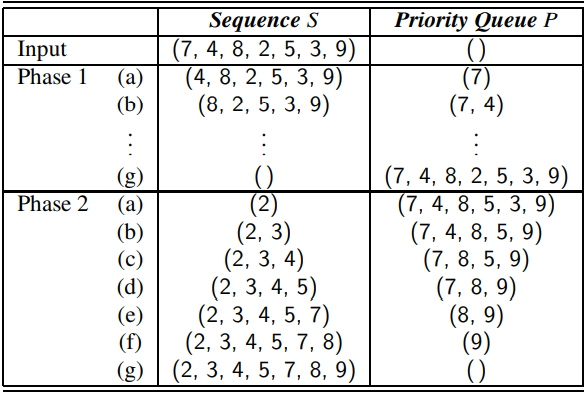
1. In the first phase, we insert the elements of *S* as keys into an initially empty priority queue *P* by means of a series of *n* insert operations, one for each element.
2. In the second phase, we extract the elements from *P* in nondecreasing order by means of a series of *n* removeMin operations, putting them back into *S* in that order.

A Java implementation of this algorithm (assuming that the sequence is stored as a positional list)



Selection Sort

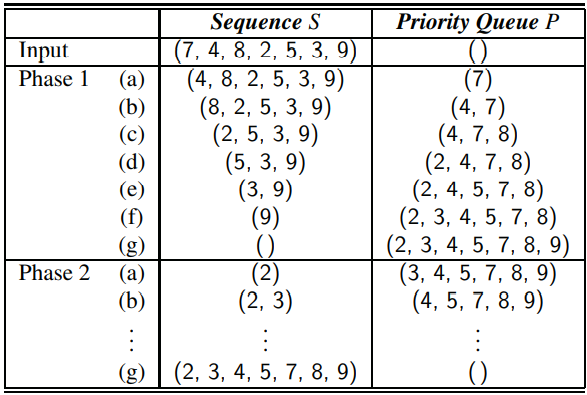
In Phase 1 of the pqSort scheme, we insert all elements into a priority queue P; in Phase 2 we repeatedly remove the minimal element from P using the removeMin method. If we implement P with an unsorted list, then Phase 1 of pqSort takes O(n) time, for we can insert each element in O(1) time. In Phase 2, the running time of each removeMin operation is proportional to the size of P. Thus, the bottleneck computation is the repeated “selection” of the minimum element in Phase 2. For this reason, this algorithm is better known as **selection-sort**.



The total time needed for the second phase is O(n^2) [Gauss identity].

Insertion Sort

If we implement the priority queue P using a sorted list, then the running time of Phase 2 improves to O(n), for each operation removeMin on P now takes O(1) time. Unfortunately, Phase 1 now becomes the bottleneck for the running time, since, in the worst case, each insert operation takes time proportional to the size of P. This sorting algorithm is therefore better known as **insertion-sort**.



This time phase one is O(n^2) [Gauss identity].

Heap Sort

Realizing a priority queue with a heap has the advantage that all the methods in the priority queue ADT run in logarithmic time or better.

During Phase 1, the ith insert operation takes O(log i) time, since the heap has i entries after the operation is performed. Therefore, this phase takes O(n log n) time. (It could be improved to O(n) with the bottom-up heap construction)

During the second phase of method pqSort, the jth removeMin operation runs in O(log(n − j + 1)), since the heap has n − j + 1 entries at the time the operation is performed. Summing over all j, this phase takes O(nlog n) time, so the entire priority-queue sorting algorithm runs in O(nlog n) time when we use a heap to implement the priority queue.

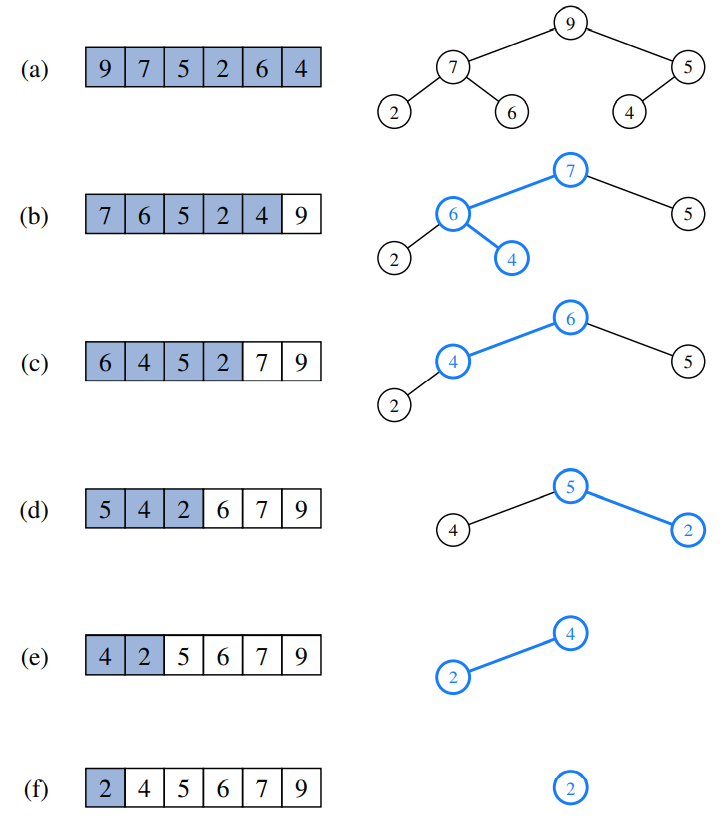


Implementing Heap-Sort In-Place

If the sequence *S* to be sorted is implemented by means of an array-based sequence, such as an ArrayList in Java, we can speed up heap-sort and reduce its space requirement by a constant factor by using a portion of the array itself to store the heap, thus avoiding the use of an auxiliary heap data structure. This is accomplished by modifying the algorithm as follows:

1. We redefine the heap operations to be a maximum-oriented heap, with each position key being at least as large as its children. This can be done by recoding the algorithm, or by providing a new comparator that reverses the outcome of each comparison. At any time during the execution of the algorithm, we use the left portion of S, up to a certain index i − 1, to store the entries of the heap, and the right portion of S, from index i to n − 1, to store the elements of the sequence. Thus, the first i elements of S (at indices 0,...,i−1) provide the array-list representation of the heap.
2. In the first phase of the algorithm, we start with an empty heap and move the boundary between the heap and the sequence from left to right, one step at a time. In step i, for i = 1,...,n, we expand the heap by adding the element at index i−1.
3. In the second phase of the algorithm, we start with an empty sequence and move the boundary between the heap and the sequence from right to left, one step at a time. At step i, for i = 1,...,n, we remove a maximal element from the heap and store it at index n−i.

[On the figure below] Phase 2 of an in-place heap-sort. The heap portion of each sequence representation is highlighted. The binary tree that each sequence (implicitly) represents is diagrammed with the most recent path of down-heap bubbling highlighted.



Merge Sort

The **divide-and-conquer** pattern consists of the following three steps:

1. Divide: If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution so obtained. Otherwise, divide the input data into two or more disjoint subsets.
2. Conquer: Recursively solve the subproblems associated with the subsets.
3. Combine: Take the solutions to the subproblems and merge them into a solution to the original problem.

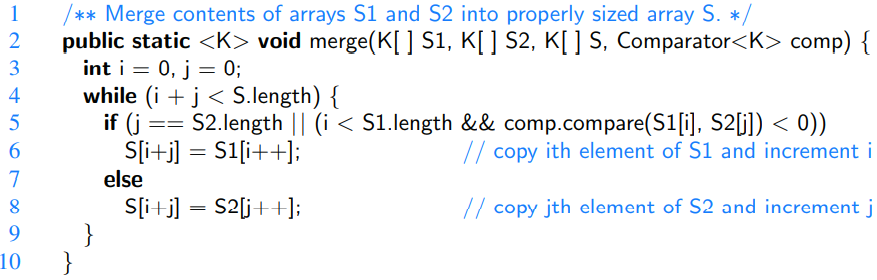
To sort a sequence S with n elements using the three divide-andconquer steps, the merge-sort algorithm proceeds as follows:

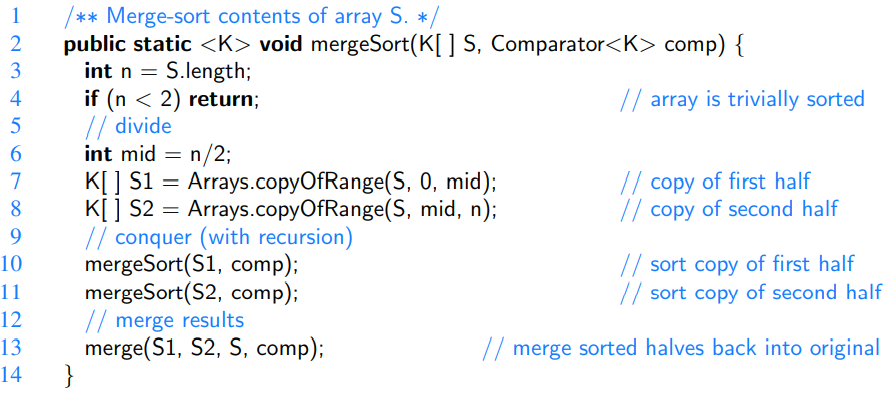
1. Divide: If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), remove all the elements from S and put them into two sequences, S1 and S2, each containing about half of the elements of S; that is, S1 contains the first ⌊n/2 ⌋ elements of S, and S2 contains the remaining ⌈n/2 ⌉ elements.
2. Conquer: Recursively sort sequences S1 and S2.
3. Combine: Put the elements back into S by merging the sorted sequences S1 and S2 into a sorted sequence.

[Study the visualisation of Merge Sort]

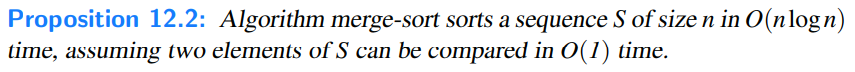


Array-Based Implementation of Merge-Sort



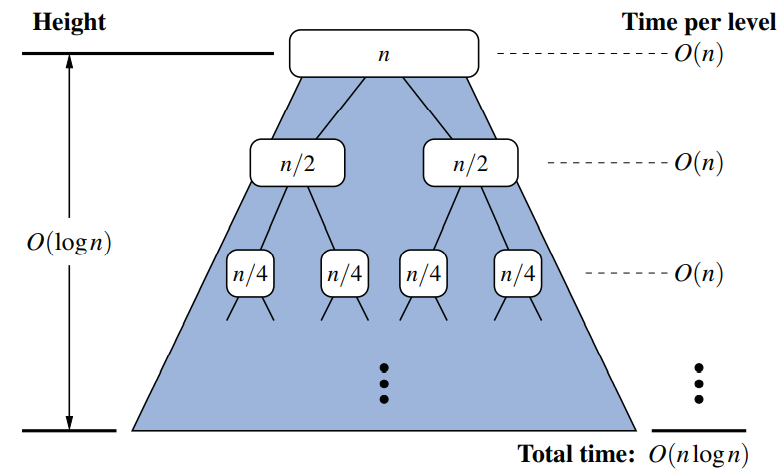


Time Complexity



A visual analysis of the running time of merge-sort

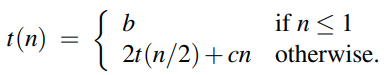
Each node represents the time spent in a particular recursive call, labeled with the size of its subproblem:



Merge-Sort and Recurrence Equations

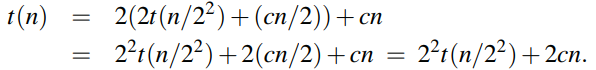
There is another way to justify that the running time of the merge-sort algorithm is O(n log n).

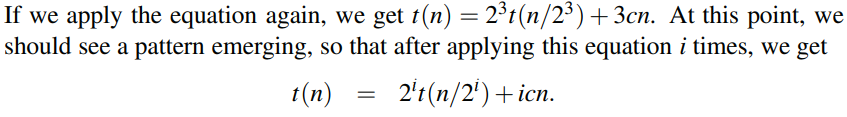
Let the function t(n) denote the worst-case running time of merge-sort on an input sequence of size n.



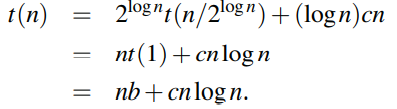
It is called a **recurrence equation**, since the function appears on both the left- and right-hand sides of the equal sign.

What we want is a big-Oh of t(n) that does not involve the function t(n) itself. That is, we want a **closed-form** characterization of t(n).





The issue that remains, then, is to determine when to stop this process. To see when to stop, recall that we switch to the closed form t(n) = b when n ≤ 1, which will occur when 2i = n. In other words, this will occur when i = log n. Making this substitution, then, yields



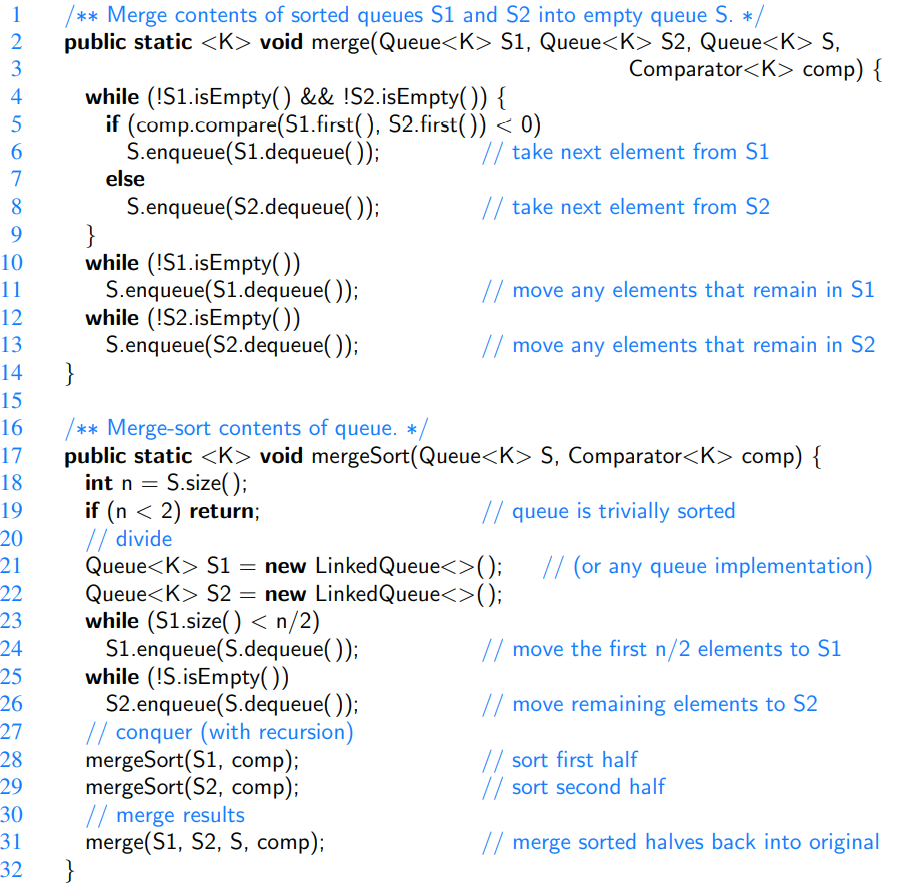
That is, we get an alternative justification of the fact that t(n) is O(n log n).

Alternative Implementations of Merge-Sort

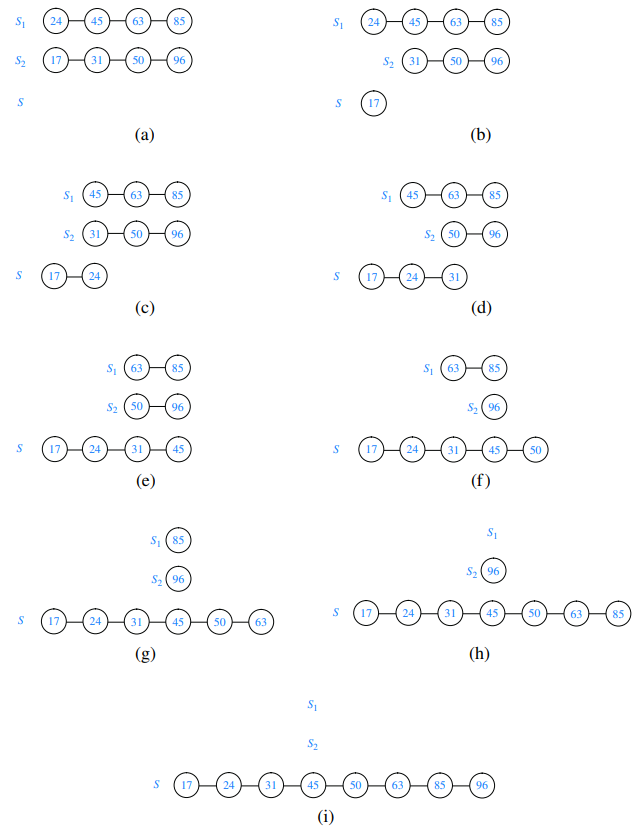
Sorting Linked Lists

The merge-sort algorithm can easily be adapted to use any form of a basic queue as its container type.We provide such an implementation, based on use of the LinkedQueue class

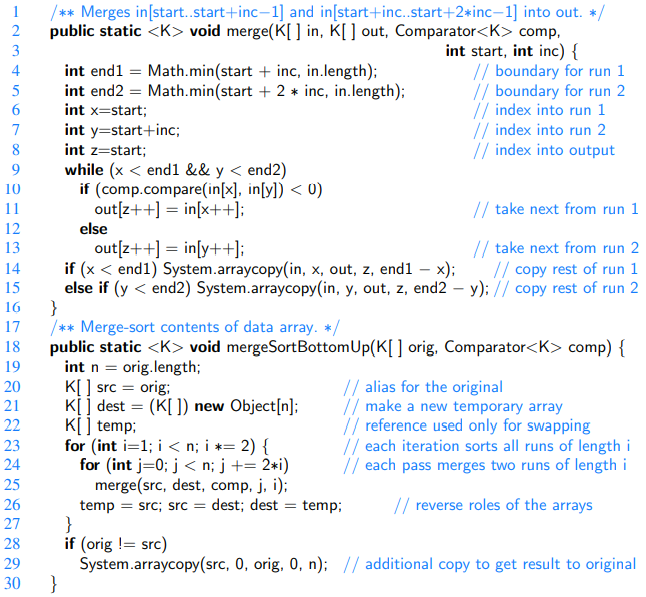
An implementation of merge-sort using a basic queue:



An Example of the Execution of the above Merge Algorithm:

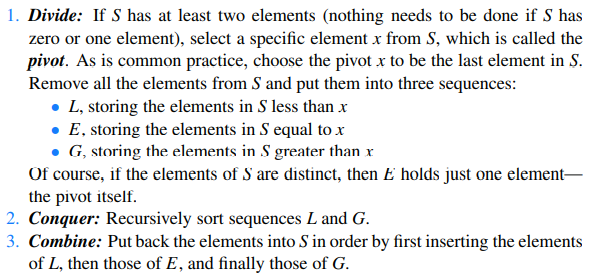


A Bottom-Up (Nonrecursive) Merge-Sort

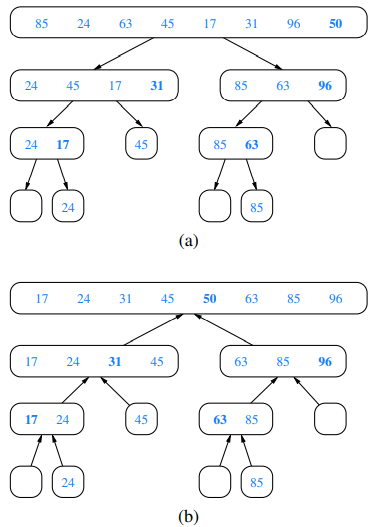


Quick-Sort

Like merge-sort, this algorithm is also based on the **divide-and-conquer** paradigm, but it uses this technique in a somewhat opposite manner, as all the hard work is done **before** the recursive calls.



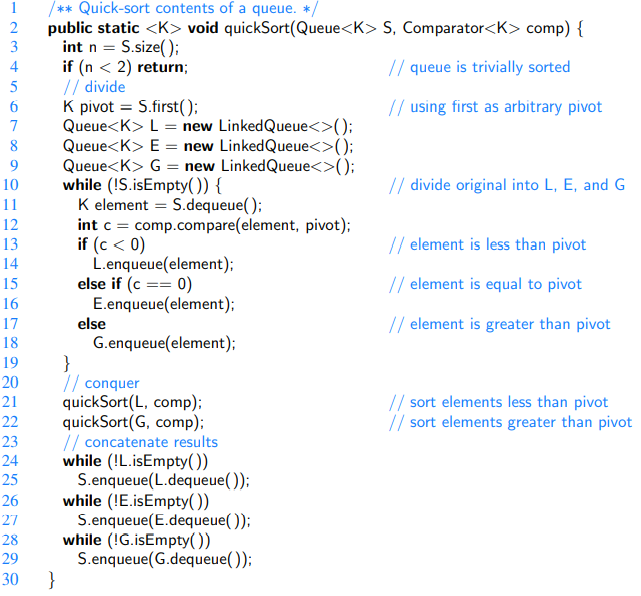
Quick-sort tree T for an execution of the quick-sort algorithm on a sequence with 8 elements: (a) input sequences processed at each node of T; (b) output sequences generated at each node of T. The pivot used at each level of the recursion is shown in bold.



[Study the visualisation of Quick-Sort]

Performing Quick-Sort on General Sequences

We give an implementation of the quick-sort algorithm that works on any sequence type that operates as a queue.



Time Complexity:

O(n log n). [Explanation is in the book]

Randomized Quick-Sort

One common method for analyzing quick-sort is to assume that the pivot will always divide the sequence in a reasonably balanced manner. However, we feel such an assumption would presuppose knowledge about the input distribution that is typically not available. For example, we would have to assume that we will rarely be given “almost” sorted sequences to sort, which are actually common in many applications.

Since the goal of the partition step of the quick-sort method is to divide the sequence S with sufficient balance, let us introduce randomization into the algorithm and pick as the pivot a **random element** of the input sequence.



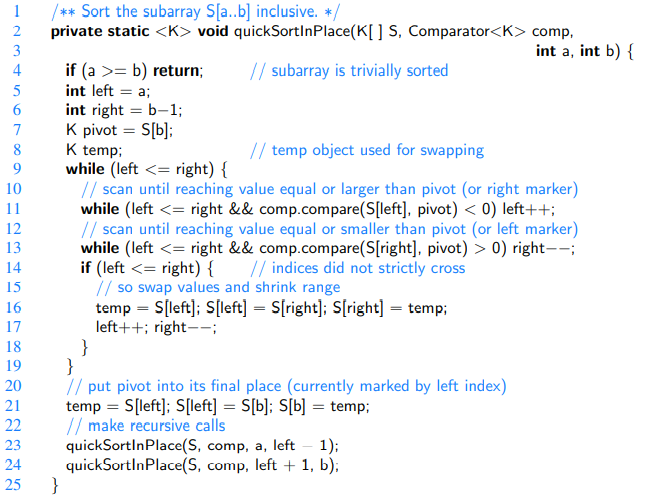
[Justification is in the book]

Additional Optimizations for Quick-Sort

An algorithm is **in-place** if it uses only a small amount of memory in addition to that needed for the original input.

Our previous implementation of quick-sort does not qualify as in-place because, we use additional containers L, E, and G when dividing a sequence S within each recursive call.

In-place quick-sort:



Making the stack size O(log n)

Although the implementation we describe in this section for dividing the sequence into two pieces is in-place, we note that the complete quick-sort algorithm needs space for a stack proportional to the depth of the recursion tree, which in this case can be as large as n−1. Admittedly, the expected stack depth is O(log n), which is small compared to n. Nevertheless, a simple trick lets us guarantee the stack size is O(logn). The main idea is to design a nonrecursive version of in-place quick-sort using an explicit stack to iteratively process subproblems (each of which can be represented with a pair of indices marking subarray boundaries). Each iteration involves popping the top subproblem, splitting it in two (if it is big enough), and pushing the two new subproblems. The trick is that when pushing the new subproblems, we should first push the larger subproblem and then the smaller one. In this way, the sizes of the subproblems will at least double as we go down the stack; hence, the stack can have depth at most O(log n).

A Note on Pivot Selection

First we selected blindly the last element as a pivot.

Then we picked the pivot randomly.

We can also use the so called **median-of-three:** the median of tree values, taken respectively from the front, middle, and tail of the array. This median-of-three heuristic will more often choose a good pivot and computing a median of three may require lower overhead than selecting a pivot with a random number generator.

For larger data sets, the median of more than three potential pivots might be computed.

Hybrid Approaches

Quick-sort has very good performance on large data sets, it has rather high overhead on relatively small data sets. It is therefore common, in optimized sorting implementations, to use a hybrid approach, with a divide-and-conquer algorithm used until the size of a subsequence falls below some threshold (perhaps 50 elements); insertion-sort can be directly invoked upon portions with length below the threshold.