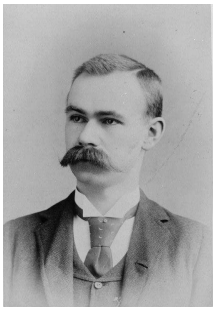
5

Sorting and Selection



Telephone directories are sorted alphabetically by last name. Why? Because a sorted index can be searched quickly. Even in the telephone directory of a huge city, one can usually find a name in a few seconds. In an unsorted index, nobody would even try to find a name. To a first approximation, this chapter teaches you how to turn an unordered collection of elements into an ordered collection, i.e., how to sort *the collection. However, sorting has many other uses as well. An early example of a massive data-processing task was the statistical evaluation of census data; 1500 people needed seven years to manually process data from the US census in 1880. The engineer Herman Hollerith,*[1] *who participated in this evaluation as a statistician, spent much of the ten years to the next census developing counting and sorting machines for mechanizing this gigantic endeavor. Although the 1890 census had to evaluate more people and more questions, the basic evaluation was finished in 1891. Hollerith's company continued to play an important role in the development of the information-processing industry; since 1924, it has been known as International Business Machines (IBM). Sorting is important for census statistics because one often wants to form subcollections, for example, all persons between age 20 and 30 and living on a farm. Two applications of sorting solve the problem. First, we sort all persons by age and form the subcollection of persons between 20 and 30 years of age. Then we sort the subcollection by home and extract the subcollection of persons living on a farm.*

Although we probably all have an intuitive concept of what *sorting* is about, let us give a formal definition. The input is a sequence $s = \langle e_1, \ldots, e_n \rangle$ of n elements. Each element e_i has an associated *key* $k_i = key(e_i)$. The keys come from an ordered universe, i.e., there is a *linear order* $\leq$ defined on the keys.[2] For ease of notation, we extend the comparison relation to elements so that $e \leq e'$ if and only

[1] The photograph was taken by C. M. Bell (see US Library of Congress's Prints and Photographs Division, ID cph.3c15982).

[2] A linear order is a reflexive, transitive, and weakly antisymmetric relation. In contrast to a *total order*, it allows equivalent elements (see Appendix A for details).

if $key(e) \leq key(e')$. The task is to produce a sequence $s' = \langle e'_1, \ldots, e'_n \rangle$ such that s' is a permutation of s and such that $e'_1 \leq e'_2 \leq \cdots \leq e'_n$. Observe that the ordering of equivalent elements is arbitrary.

Although different comparison relations for the same data type may make sense, the most frequent relations are the obvious order for numbers and the *lexicographic order* (see Appendix A) for tuples, strings, and sequences. The lexicographic order for strings comes in different flavors. We may treat corresponding lower-case and upper-case characters as being equivalent, and different rules for treating accented characters are used in different contexts.

Exercise 5.1. Given linear orders $\leq_A$ for A and $\leq_B$ for B, define a linear order on $A \times B$.

Exercise 5.2. Define a total order for complex numbers with the property that $x \leq y$ implies $|x| \leq |y|$.

Sorting is a ubiquitous algorithmic tool; it is frequently used as a preprocessing step in more complex algorithms. We shall give some examples.

- *Preprocessing for fast search*. In Sect. 2.5 on binary search, we have already seen that a sorted directory is easier to search, both for humans and computers. Moreover, a sorted directory supports additional operations, such as finding all elements in a certain range. We shall discuss searching in more detail in Chap. 7. Hashing is a method for searching unordered sets.
- *Grouping*. Often, we want to bring equal elements together to count them, eliminate duplicates, or otherwise process them. Again, hashing is an alternative. But sorting has advantages, since we shall see rather fast, space-efficient, deterministic sorting algorithm that scale to huge data sets.
- *Processing in a sorted order*. Certain algorithms become very simple if the inputs are processed in sorted order. Exercise 5.3 gives an example. Other examples are Kruskal's algorithm in Sect. 11.3 and several of the algorithms for the knapsack problem in Chap. 12. You may also want to remember sorting when you solve Exercise 8.6 on interval graphs.

In Sect. 5.1, we shall introduce several simple sorting algorithms. They have quadratic complexity, but are still useful for small input sizes. Moreover, we shall learn some low-level optimizations. Section 5.2 introduces *mergesort*, a simple divide-and-conquer sorting algorithm that runs in time $O(n \log n)$. Section 5.3 establishes that this bound is optimal for all *comparison-based* algorithms, i.e., algorithms that treat elements as black boxes that can only be compared and moved around. The *quicksort* algorithm described in Sect. 5.4 is again based on the divide-and-conquer principle and is perhaps the most frequently used sorting algorithm. Quicksort is also a good example of a randomized algorithm. The idea behind quicksort leads to a simple algorithm for a problem related to sorting. Section 5.5 explains how the k-th smallest of n elements can be *selected* in time $O(n)$. Sorting can be made even faster than the lower bound obtained in Sect. 5.3 by looking at the bit patterns of the keys, as explained in Sect. 5.6. Finally, Section 5.7 generalizes quicksort and mergesort to very good algorithms for sorting inputs that do not fit into internal memory.

Exercise 5.3 (a simple scheduling problem). A hotel manager has to process n advance bookings of rooms for the next season. His hotel has k identical rooms. Bookings contain an arrival date and a departure date. He wants to find out whether there are enough rooms in the hotel to satisfy the demand. Design an algorithm that solves this problem in time $O(n \log n)$. Hint: consider the set of all arrivals and departures. Sort the set and process it in sorted order.

Exercise 5.4 (sorting with a small set of keys). Design an algorithm that sorts n elements in $O(k \log k + n)$ expected time if there are only k different keys appearing in the input. Hint: combine hashing and sorting.

Exercise 5.5 (checking). It is easy to check whether a sorting routine produces a sorted output. It is less easy to check whether the output is also a permutation of the input. But here is a fast and simple Monte Carlo algorithm for integers: (a) Show that $\langle e_1, \ldots, e_n \rangle$ is a permutation of $\langle e'_1, \ldots, e'_n \rangle$ iff the polynomial

$$q(z) := \prod_{i=1}^{n} (z - e_i) - \prod_{i=1}^{n} (z - e'_i)$$

is identically zero. Here, z is a variable. (b) For any $\varepsilon > 0$, let p be a prime with $p > \max\{n/\varepsilon, e_1, \ldots, e_n, e'_1, \ldots, e'_n\}$. Now the idea is to evaluate the above polynomial mod p for a random value $z \in [0..p-1]$. Show that if $\langle e_1, \ldots, e_n \rangle$ is *not* a permutation of $\langle e'_1, \ldots, e'_n \rangle$, then the result of the evaluation is zero with probability at most ε. Hint: a nonzero polynomial of degree n has at most n zeros.

5.1 Simple Sorters

We shall introduce two simple sorting techniques here: *selection sort* and *insertion sort*.

Selection sort repeatedly selects the smallest element from the input sequence, deletes it, and adds it to the end of the output sequence. The output sequence is initially empty. The process continues until the input sequence is exhausted. For example,

$$\langle\rangle, \langle 4,7,1,1 \rangle \rightsquigarrow \langle 1 \rangle, \langle 4,7,1 \rangle \rightsquigarrow \langle 1,1 \rangle, \langle 4,7 \rangle \rightsquigarrow \langle 1,1,4 \rangle, \langle 7 \rangle \rightsquigarrow \langle 1,1,4,7 \rangle, \langle\rangle \; .$$

The algorithm can be implemented such that it uses a single array of n elements and works *in-place*, i.e., it needs no additional storage beyond the input array and a constant amount of space for loop counters, etc. The running time is quadratic.

Exercise 5.6 (simple selection sort). Implement selection sort so that it sorts an array with n elements in time $O(n^2)$ by repeatedly scanning the input sequence. The algorithm should be in-place, i.e., the input sequence and the output sequence should share the same array. Hint: the implementation operates in n phases numbered 1 to n. At the beginning of the i-th phase, the first $i-1$ locations of the array contain the $i-1$ smallest elements in sorted order and the remaining $n-i+1$ locations contain the remaining elements in arbitrary order.

In Sect. 6.5, we shall learn about a more sophisticated implementation where the input sequence is maintained as a *priority queue*. Priority queues support efficient repeated selection of the minimum element. The resulting algorithm runs in time $O(n \log n)$ and is frequently used. It is efficient, it is deterministic, it works in-place, and the input sequence can be dynamically extended by elements that are larger than all previously selected elements. The last feature is important in discrete-event simulations, where events are to be processed in increasing order of time and processing an event may generate further events in the future.

Selection sort maintains the invariant that the output sequence is sorted by carefully choosing the element to be deleted from the input sequence. *Insertion sort* maintains the same invariant by choosing an arbitrary element of the input sequence but taking care to insert this element at the right place in the output sequence. For example,

$$\langle\rangle, \langle 4,7,1,1\rangle \rightsquigarrow \langle 4\rangle, \langle 7,1,1\rangle \rightsquigarrow \langle 4,7\rangle, \langle 1,1\rangle \rightsquigarrow \langle 1,4,7\rangle, \langle 1\rangle \rightsquigarrow \langle 1,1,4,7\rangle, \langle\rangle \ .$$

Figure 5.1 gives an in-place array implementation of insertion sort. The implementation is straightforward except for a small trick that allows the inner loop to use only a single comparison. When the element e to be inserted is smaller than all previously inserted elements, it can be inserted at the beginning without further tests. Otherwise, it suffices to scan the sorted part of a from right to left while e is smaller than the current element. This process has to stop, because $a[1] \leq e$.

In the worst case, insertion sort is quite slow. For example, if the input is sorted in decreasing order, each input element is moved all the way to $a[1]$, i.e., in iteration i of the outer loop, i elements have to be moved. Overall, we obtain

$$\sum_{i=2}^{n}(i-1) = -n + \sum_{i=1}^{n} i = \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2} = \Omega\left(n^2\right)$$

movements of elements (see also (A.11)).

Nevertheless, insertion sort is useful. It is fast for small inputs (say, $n \leq 10$) and hence can be used as the base case in divide-and-conquer algorithms for sorting.

```
Procedure insertionSort(a : Array [1..n] of Element)
    for i := 2 to n do
        invariant a[1] ≤ ··· ≤ a[i−1]
        // move a[i] to the right place
        e := a[i]
        if e < a[1] then                                          // new minimum
            for j := i downto 2 do a[j] := a[j−1]
            a[1] := e
        else                                                      // use a[1] as a sentinel
            for j := i downto −∞ while a[j−1] > e do a[j] := a[j−1]
            a[j] := e
```

Fig. 5.1. Insertion sort.

Furthermore, in some applications the input is already "almost" sorted, and in this situation insertion sort will be fast.

Exercise 5.7 (almost sorted inputs). Prove that insertion sort runs in time $O(n+D)$ where $D = \sum_i |r(e_i) - i|$ and $r(e_i)$ is the *rank* (position) of e_i in the sorted output.

Exercise 5.8 (average-case analysis). Assume that the input to an insertion sort is a permutation of the numbers 1 to n. Show that the average execution time over all possible permutations is $\Omega(n^2)$. Hint: argue formally that about one-third of the input elements in the right third of the array have to be moved to the left third of the array. Can you improve the argument to show that, on average, $n^2/4 - O(n)$ iterations of the inner loop are needed?

Exercise 5.9 (insertion sort with few comparisons). Modify the inner loops of the array-based insertion sort algorithm in Fig. 5.1 so that it needs only $O(n \log n)$ comparisons between elements. Hint: use binary search as discussed in Chap. 7. What is the running time of this modification of insertion sort?

Exercise 5.10 (efficient insertion sort?). Use the data structure for sorted sequences described in Chap. 7 to derive a variant of insertion sort that runs in time $O(n \log n)$.

***Exercise 5.11 (formal verification).** Use your favorite verification formalism, for example Hoare calculus, to prove that insertion sort produces a permutation of the input (i.e., it produces a sorted permutation of the input).

5.2 Mergesort – an $O(n \log n)$ Sorting Algorithm

Mergesort is a straightforward application of the divide-and-conquer principle. The unsorted sequence is split into two parts of about equal size. The parts are sorted recursively, and the sorted parts are merged into a single sorted sequence. This approach is efficient because merging two sorted sequences a and b is quite simple. The globally smallest element is either the first element of a or the first element of b. So we move the smaller element to the output, find the second smallest element using the same approach, and iterate until all elements have been moved to the output. Figure 5.2 gives pseudocode, and Figure 5.3 illustrates a sample execution. If the sequences are represented as linked lists (see, Sect. 3.1), no allocation and deallocation of list items is needed. Each iteration of the inner loop of *merge* performs one element comparison and moves one element to the output. Each iteration takes constant time. Hence, merging runs in linear time.

Theorem 5.1. *The function merge, applied to sequences of total length n, executes in time $O(n)$ and performs at most $n-1$ element comparisons.*

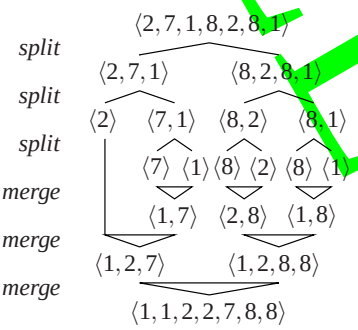
For the running time of mergesort, we obtain the following result.

Theorem 5.2. *Mergesort runs in time $O(n \log n)$ and performs no more than $\lceil n \log n \rceil$ element comparisons.*

```
Function mergeSort(⟨e_1,...,e_n⟩) : Sequence of Element
    if n = 1 then return ⟨e_1⟩
    else return merge(  mergeSort(⟨e_1,...,e_⌊n/2⌋⟩),
                        mergeSort(⟨e_⌊n/2⌋+1,...,e_n⟩))

// merging two sequences represented as lists
Function merge(a,b : Sequence of Element) : Sequence of Element
    c := ⟨⟩
    loop
        invariant a, b, and c are sorted and ∀e ∈ c,e' ∈ a∪b : e ≤ e'
        if a.isEmpty  then  c.concat(b); return c
        if b.isEmpty  then  c.concat(a); return c
        if a.first ≤ b.first  then  c.moveToBack(a.first)
        else                        c.moveToBack(b.first)
```

Fig. 5.2. Mergesort.



a	b	c	operation
$\langle 1,2,7\rangle$	$\langle 1,2,8,8\rangle$	$\langle\rangle$	move a
$\langle 2,7\rangle$	$\langle 1,2,8,8\rangle$	$\langle 1\rangle$	move b
$\langle 2,7\rangle$	$\langle 2,8,8\rangle$	$\langle 1,1\rangle$	move a
$\langle 7\rangle$	$\langle 2,8,8\rangle$	$\langle 1,1,2\rangle$	move b
$\langle 7\rangle$	$\langle 8,8\rangle$	$\langle 1,1,2,2\rangle$	move a
$\langle\rangle$	$\langle 8,8\rangle$	$\langle 1,1,2,2,7\rangle$	concat b
$\langle\rangle$	$\langle\rangle$	$\langle 1,1,2,2,7,8,8\rangle$	

Fig. 5.3. Execution of $mergeSort(\langle 2,7,1,8,2,8,1\rangle)$. The *left* part illustrates the recursion in *mergeSort* and the *right* part illustrates the *merge* in the outermost call

Proof. Let $C(n)$ denote the worst-case number of element comparisons performed. We have $C(1) = 0$ and $C(n) \le C(\lfloor n/2\rfloor) + C(\lceil n/2\rceil) + n - 1$, using Theorem 5.1. The master theorem for recurrence relations (2.5) suggests that $C(n) = O(n \log n)$. We shall give two proofs. The first proof shows that $C(n) \le 2n\lceil \log n\rceil$, and the second proof shows that $C(n) \le n\lceil \log n\rceil$.

For n a power of two, we define $D(1) = 0$ and $D(n) = 2D(n/2) + n$. Then $D(n) = n \log n$ for n a power of two, by the master theorem for recurrence relations. We claim that $C(n) \le D(2^k)$, where k is such that $2^{k-1} < n \le 2^k$. Then $C(n) \le D(2^k) = 2^k k \le 2n\lceil \log n\rceil$. It remains to argue the inequality $C(n) \le D(2^k)$. We use induction on k. For $k = 0$, we have $n = 1$ and $C(1) = 0 = D(1)$, and the claim certainly holds. For $k > 1$, we observe that $\lfloor n/2\rfloor \le \lceil n/2\rceil \le 2^{k-1}$, and hence

$$C(n) \le C(\lfloor n/2\rfloor) + C(\lceil n/2\rceil) + n - 1 \le 2D(2^{k-1}) + 2^k - 1 \le D(2^k)\ .$$

This completes the first proof. We turn now to the second, refined proof. We prove that

$$C(n) \leq n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 \leq n \log n$$

by induction over n. For $n = 1$, the claim is certainly true. So, assume $n > 1$. We distinguish two cases. Assume first that we have $2^{k-1} < \lfloor n/2 \rfloor \leq \lceil n/2 \rceil \leq 2^k$ for some integer k. Then $\lceil \log \lfloor n/2 \rfloor \rceil = \lceil \log \lceil n/2 \rceil \rceil = k$ and $\lceil \log n \rceil = k+1$, and hence

$$\begin{aligned} C(n) &\leq C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1 \\ &\leq \big(\lfloor n/2 \rfloor k - 2^k + 1\big) + \big(\lceil n/2 \rceil k - 2^k + 1\big) + n - 1 \\ &= nk + n - 2^{k+1} + 1 = n(k+1) - 2^{k+1} + 1 = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 \,. \end{aligned}$$

Otherwise, we have $\lfloor n/2 \rfloor = 2^{k-1}$ and $\lceil n/2 \rceil = 2^{k-1} + 1$ for some integer k, and therefore $\lceil \log \lfloor n/2 \rfloor \rceil = k-1$, $\lceil \log \lceil n/2 \rceil \rceil = k$, and $\lceil \log n \rceil = k+1$. Thus

$$\begin{aligned} C(n) &\leq C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1 \\ &\leq \big(2^{k-1}(k-1) - 2^{k-1} + 1\big) + \big((2^{k-1}+1)k - 2^k + 1\big) + 2^k + 1 - 1 \\ &= (2^k + 1)k - 2^{k-1} - 2^{k-1} + 1 + 1 \\ &= (2^k + 1)(k+1) - 2^{k+1} + 1 = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 \,. \end{aligned}$$

The bound for the execution time can be verified using a similar recurrence relation. □

Mergesort is the method of choice for sorting linked lists and is therefore frequently used in functional and logical programming languages that have lists as their primary data structure. In Sect. 5.3, we shall see that mergesort is basically optimal as far as the number of comparisons is concerned; so it is also a good choice if comparisons are expensive. When implemented using arrays, mergesort has the additional advantage that it streams through memory in a sequential way. This makes it efficient in memory hierarchies. Section 5.7 has more on that issue. Mergesort is still not the usual method of choice for an efficient array-based implementation, however, since *merge* does not work in-place. (But see Exercise 5.17 for a possible way out.)

Exercise 5.12. Explain how to insert k new elements into a sorted list of size n in time $O(k \log k + n)$.

Exercise 5.13. We discussed *merge* for lists but used abstract sequences for the description of *mergeSort*. Give the details of *mergeSort* for linked lists.

Exercise 5.14. Implement mergesort in a functional programming language.

Exercise 5.15. Give an efficient array-based implementation of mergesort in your favorite imperative programming language. Besides the input array, allocate one auxiliary array of size n at the beginning and then use these two arrays to store all intermediate results. Can you improve the running time by switching to insertion sort for small inputs? If so, what is the optimal switching point in your implementation?

Exercise 5.16. The way we describe *merge*, there are three comparisons for each loop iteration – one element comparison and two termination tests. Develop a variant using sentinels that needs only one termination test. Can you do this task without appending dummy elements to the sequences?

Exercise 5.17. Exercise 3.20 introduced a list-of-blocks representation for sequences. Implement merging and mergesort for this data structure. During merging, reuse emptied input blocks for the output sequence. Compare the space and time efficiency of mergesort for this data structure, for plain linked lists, and for arrays. Pay attention to constant factors.

5.3 A Lower Bound

Algorithms give upper bounds on the complexity of a problem. By the preceding discussion, we know that we can sort n items in time $O(n \log n)$. Can we do better, and maybe even achieve linear time? A "yes" answer requires a better algorithm and its analysis. But how could we potentially argue a "no" answer? We would have to argue that no algorithm, however ingenious, can run in time $o(n \log n)$. Such an argument is called a *lower bound*. So what is the answer? The answer is both no and yes. The answer is no, if we restrict ourselves to comparison-based algorithms, and the answer is yes if we go beyond comparison-based algorithms. We shall discuss non-comparison-based sorting in Sect. 5.6.

So what is a comparison-based sorting algorithm? The input is a set $\{e_1, \ldots, e_n\}$ of n elements, and the only way the algorithm can learn about its input is by comparing elements. In particular, it is not allowed to exploit the representation of keys, for example as bit strings. Deterministic comparison-based algorithms can be viewed as trees. They make an initial comparison; for instance, the algorithms asks "$e_i \leq e_j$?", with outcomes yes and no. On the basis of the outcome, the algorithm proceeds to the next comparison. The key point is that the comparison made next depends only on the outcome of all preceding comparisons and nothing else. Figure 5.4 shows a sorting tree for three elements.

When the algorithm terminates, it must have collected sufficient information so that it can commit to a permutation of the input. When can it commit? We perform the following thought experiment. We assume that the input keys are distinct, and consider any of the $n!$ permutations of the input, say π. The permutation π corresponds to the situation that $e_{\pi(1)} < e_{\pi(2)} < \ldots < e_{\pi(n)}$. We answer all questions posed by the algorithm so that they conform to the ordering defined by π. This will lead us to a leaf ℓ_π of the comparison tree.

Lemma 5.3. *Let π and σ be two distinct permutations of n elements. The leaves ℓ_π and ℓ_σ must then be distinct.*

Proof. Assume otherwise. In a leaf, the algorithm commits to some ordering of the input and so it cannot commit to both π and σ. Say it commits to π. Then, on an input ordered according to σ, the algorithm is incorrect, which is a contradiction. □

The lemma above tells us that any comparison tree for sorting must have at least $n!$ leaves. Since a tree of depth T has at most 2^T leaves, we must have

$$2^T \geq n! \quad \text{or} \quad T \geq \log n! \; .$$

Via Stirling's approximation to the factorial (A.9), we obtain

$$T \geq \log n! \geq \log \left(\frac{n}{e}\right)^n = n \log n - n \log e \; .$$

Theorem 5.4. *Any comparison-based sorting algorithm needs $n \log n - \mathrm{O}(n)$ comparisons in the worst case.*

We state without proof that this bound also applies to randomized sorting algorithms and to the average-case complexity of sorting, i.e., worst-case instances are not much more difficult than random instances. Furthermore, the bound applies even if we only want to solve the seemingly simpler problem of checking whether some element appears twice in a sequence.

Theorem 5.5. *Any comparison-based sorting algorithm needs $n \log n - \mathrm{O}(n)$ comparisons on average, i.e.,*

$$\frac{\sum_\pi d_\pi}{n!} = n \log n - \mathrm{O}(n) \; ,$$

where the sum extends over all $n!$ permutations of the n elements and d_π is the depth of the leaf ℓ_π.

Exercise 5.18. Show that any comparison-based algorithm for determining the smallest of n elements requires $n-1$ comparisons. Show also that any comparison-based algorithm for determining the smallest and second smallest elements of n elements requires at least $n-1+\log n$ comparisons. Give an algorithm with this performance.

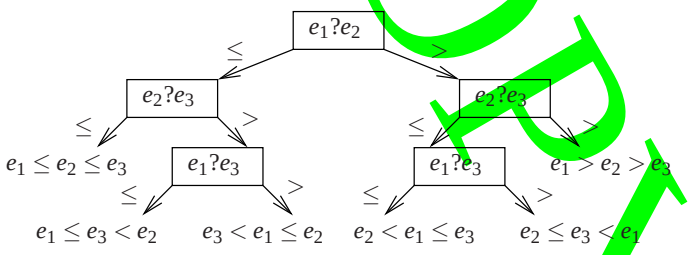


Fig. 5.4. A tree that sorts three elements. We first compare e_1 and e_2. If $e_1 \leq e_2$, we compare e_2 with e_3. If $e_2 \leq e_3$, we have $e_1 \leq e_2 \leq e_3$ and are finished. Otherwise, we compare e_1 with e_3. For either outcome, we are finished. If $e_1 > e_2$, we compare e_2 with e_3. If $e_2 > e_3$, we have $e_1 > e_2 > e_3$ and are finished. Otherwise, we compare e_1 with e_3. For either outcome, we are finished. The worst-case number of comparisons is three. The average number is $(2+3+3+2+3+3)/6 = 8/3$

Exercise 5.19. The *element uniqueness problem* is the task of deciding whether in a set of n elements, all elements are pairwise distinct. Argue that comparison-based algorithms require $\Omega(n \log n)$ comparisons. Why does this not contradict the fact that we can solve the problem in linear expected time using hashing?

Exercise 5.20 (lower bound for average case). With the notation above, let d_π be the depth of the leaf ℓ_π. Argue that $A = (1/n!)\sum_\pi d_\pi$ is the average-case complexity of a comparison-based sorting algorithm. Try to show that $A \geq \log n!$. Hint: prove first that $\sum_\pi 2^{-d_\pi} \leq 1$. Then consider the minimization problem "minimize $\sum_\pi d_\pi$ subject to $\sum_\pi 2^{-d_\pi} \leq 1$". Argue that the minimum is attained when all d_i's are equal.

Exercise 5.21 (sorting small inputs optimally). Give an algorithm for sorting k elements using at most $\lceil \log k! \rceil$ element comparisons. (a) For $k \in \{2,3,4\}$, use mergesort. (b) For $k = 5$, you are allowed to use seven comparisons. This is difficult. Mergesort does not do the job, as it uses up to eight comparisons. (c) For $k \in \{6,7,8\}$, use the case $k = 5$ as a subroutine.

5.4 Quicksort

Quicksort is a divide-and-conquer algorithm that is complementary to the mergesort algorithm of Sect. 5.2. Quicksort does all the difficult work *before* the recursive calls. The idea is to distribute the input elements into two or more sequences that represent nonoverlapping ranges of key values. Then, it suffices to sort the shorter sequences recursively and concatenate the results. To make the duality to mergesort complete, we would like to split the input into two sequences of equal size. Unfortunately, this is a nontrivial task. However, we can come close by picking a random splitter element. The splitter element is usually called the *pivot*. Let p denote the pivot element chosen. Elements are classified into three sequences a, b, and c of elements that are smaller than, equal to, or larger than p, respectively. Figure 5.5 gives a high-level realization of this idea, and Figure 5.6 depicts a sample execution. Quicksort has an expected execution time of $O(n \log n)$, as we shall show in Sect. 5.4.1. In Sect. 5.4.2, we discuss refinements that have made quicksort the most widely used sorting algorithm in practice.

```
Function quickSort(s : Sequence of Element) : Sequence of Element
    if |s| ≤ 1 then return s                                  // base case
    pick p ∈ s uniformly at random                            // pivot key
    a := ⟨e ∈ s : e < p⟩
    b := ⟨e ∈ s : e = p⟩
    c := ⟨e ∈ s : e > p⟩
    return concatenation of quickSort(a), b, and quickSort(c)
```

Fig. 5.5. High-level formulation of quicksort for lists.

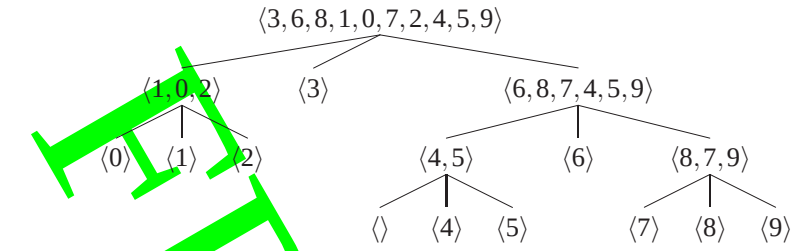


Fig. 5.6. Execution of *quickSort* (Fig. 5.5) on $\langle 3,6,8,1,0,7,2,4,5,9\rangle$ using the first element of a subsequence as the pivot. The first call of quicksort uses 3 as the pivot and generates the subproblems $\langle 1,0,2\rangle$, $\langle 3\rangle$, and $\langle 6,8,7,4,5,9\rangle$. The recursive call for the third subproblem uses 6 as a pivot and generates the subproblems $\langle 4,5\rangle$, $\langle 6\rangle$, and $\langle 8,7,9\rangle$

5.4.1 Analysis

To analyze the running time of quicksort for an input sequence $s = \langle e_1, \ldots, e_n\rangle$, we focus on the number of element comparisons performed. We allow *three-way* comparisons here, with possible outcomes "smaller", "equal", and "larger". Other operations contribute only constant factors and small additive terms to the execution time.

Let $C(n)$ denote the worst-case number of comparisons needed for any input sequence of size n and any choice of pivots. The worst-case performance is easily determined. The subsequences a, b, and c in Fig. 5.5 are formed by comparing the pivot with all other elements. This makes $n-1$ comparisons. Assume there are k elements smaller than the pivot and k' elements larger than the pivot. We obtain $C(0) = C(1) = 0$ and

$$C(n) \le n-1+\max\left\{C(k)+C(k') : 0 \le k \le n-1, 0 \le k' < n-k\right\} .$$

It is easy to verify by induction that

$$C(n) \le \frac{n(n-1)}{2} = \Theta\left(n^2\right) .$$

The worst case occurs if all elements are different and we always pick the largest or smallest element as the pivot. Thus $C(n) = n(n-1)/2$.

The expected performance is much better. We first argue for an $O(n \log n)$ bound and then show a bound of $2n \ln n$. We concentrate on the case where all elements are different. Other cases are easier because a pivot that occurs several times results in a larger middle sequence b that need not be processed any further. Consider a fixed element e_i, and let X_i denote the total number of times e_i is compared with a pivot element. Then $\sum_i X_i$ is the total number of comparisons. Whenever e_i is compared with a pivot element, it ends up in a smaller subproblem. Therefore, $X_i \le n-1$, and we have another proof for the quadratic upper bound. Let us call a comparison "good" for e_i if e_i moves to a subproblem of at most three-quarters the size. Any e_i

can be involved in at most $\log_{4/3} n$ good comparisons. Also, the probability that a pivot which is good for e_i is chosen, is at least $1/2$; this holds because a bad pivot must belong to either the smallest or the largest quarter of the elements. So $\mathrm{E}[X_i] \leq 2\log_{4/3} n$, and hence $\mathrm{E}[\sum_i X_i] = \mathrm{O}(n \log n)$. We shall now give a different argument and a better bound.

Theorem 5.6. *The expected number of comparisons performed by quicksort is*

$$\bar{C}(n) \leq 2n \ln n \leq 1.45 n \log n \,.$$

Proof. Let $s' = \langle e'_1, \ldots, e'_n \rangle$ denote the elements of the input sequence in sorted order. Elements e'_i and e'_j are compared at most once, and only if one of them is picked as a pivot. Hence, we can count comparisons by looking at the indicator random variables X_{ij}, $i < j$, where $X_{ij} = 1$ if e'_i and e'_j are compared and $X_{ij} = 0$ otherwise. We obtain

$$\bar{C}(n) = \mathrm{E}\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{prob}(X_{ij} = 1) \,.$$

The middle transformation follows from the linearity of expectations (A.2). The last equation uses the definition of the expectation of an indicator random variable $\mathrm{E}[X_{ij}] = \mathrm{prob}(X_{ij} = 1)$. Before we can further simplify the expression for $\bar{C}(n)$, we need to determine the probability of X_{ij} being 1.

Lemma 5.7. *For any $i < j$,* $\mathrm{prob}(X_{ij} = 1) = \dfrac{2}{j - i + 1}$.

Proof. Consider the $j - i + 1$-element set $M = \{e'_i, \ldots, e'_j\}$. As long as no pivot from M is selected, e'_i and e'_j are not compared, but all elements from M are passed to the same recursive calls. Eventually, a pivot p from M is selected. Each element in M has the same chance $1/|M|$ of being selected. If $p = e'_i$ or $p = e'_j$ we have $X_{ij} = 1$. The probability for this event is $2/|M| = 2/(j - i + 1)$. Otherwise, e'_i and e'_j are passed to different recursive calls, so that they will never be compared. □

Now we can finish proving Theorem 5.6 using relatively simple calculations:

$$\begin{aligned}\bar{C}(n) &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{prob}(X_{ij} = 1) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \\ &\leq \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k} = 2n \sum_{k=2}^{n} \frac{1}{k} = 2n(H_n - 1) \leq 2n(1 + \ln n - 1) = 2n \ln n \,.\end{aligned}$$

For the last three steps, recall the properties of the n-th harmonic number $H_n := \sum_{k=1}^{n} 1/k \leq 1 + \ln n$ (A.12). □

Note that the calculations in Sect. 2.8 for left-to-right maxima were very similar, although we had quite a different problem at hand.

5.4.2 *Refinements

We shall now discuss refinements of the basic quicksort algorithm. The resulting algorithm, called *qsort*, works in-place, and is fast and space-efficient. Figure 5.7 shows the pseudocode, and Figure 5.8 shows a sample execution. The refinements are nontrivial and we need to discuss them carefully.

```
Procedure qSort(a : Array of Element; ℓ, r : ℕ)          // Sort the subarray a[ℓ..r]
  while r − ℓ + 1 > n0 do                                 // Use divide-and-conquer.
    j := pickPivotPos(a, ℓ, r)                            // Pick a pivot element and
    swap(a[ℓ], a[j])                                      // bring it to the first position.
    p := a[ℓ]                                             // p is the pivot now.
    i := ℓ; j := r                                        // a: [ℓ  i→  ←j  r]
    repeat
      while a[i] < p do i++                               // Skip over elements
      while a[j] > p do j--                               // already in the correct subarray.
      if i ≤ j then                                       // If partitioning is not yet complete,
        swap(a[i], a[j]); i++; j--                        // (*) swap misplaced elements and go on.
    until i > j                                           // Partitioning is complete.
    if i < (ℓ + r)/2 then qSort(a, ℓ, j); ℓ := i          // Recurse on
    else                  qSort(a, i, r); r := j          // smaller subproblem.
  endwhile
  insertionSort(a[ℓ..r])                                  // faster for small r − ℓ
```

Fig. 5.7. Refined quicksort for arrays.

The function *qsort* operates on an array a. The arguments ℓ and r specify the subarray to be sorted. The outermost call is $qsort(a, 1, n)$. If the size of the subproblem is smaller than some constant n_0, we resort to a simple algorithm[3] such as the insertion sort shown in Fig. 5.1. The best choice for n_0 depends on many details of the machine and compiler and needs to be determined experimentally; a value somewhere between 10 and 40 should work fine under a variety of conditions.

The pivot element is chosen by a function *pickPivotPos* that we shall not specify further. The correctness does not depend on the choice of the pivot, but the efficiency does. Possible choices are the first element; a random element; the median ("middle") element of the first, middle, and last elements; and the median of a random sample consisting of k elements, where k is either a small constant, say three, or a number depending on the problem size, say $\lceil \sqrt{r-\ell+1} \rceil$. The first choice requires the least amount of work, but gives little control over the size of the subproblems; the last choice requires a nontrivial but still sublinear amount of work, but yields balanced

[3] Some authors propose leaving small pieces unsorted and cleaning up at the end using a single insertion sort that will be fast, according to Exercise 5.7. Although this nice trick reduces the number of instructions executed, the solution shown is faster on modern machines because the subarray to be sorted will already be in cache.

Fig. 5.8. Execution of *qSort* (Fig. 5.7) on $\langle 3,6,8,1,0,7,2,4,5,9\rangle$ using the first element as the pivot and $n_0 = 1$. The *left-hand side* illustrates the first partitioning step, showing elements in **bold** that have just been swapped. The *right-hand side* shows the result of the recursive partitioning operations

subproblems with high probability. After selecting the pivot p, we swap it into the first position of the subarray (= position ℓ of the full array).

The repeat–until loop partitions the subarray into two proper (smaller) subarrays. It maintains two indices i and j. Initially, i is at the left end of the subarray and j is at the right end; i scans to the right, and j scans to the left. After termination of the loop, we have $i = j+1$ or $i = j+2$, all elements in the subarray $a[\ell..j]$ are no larger than p, all elements in the subarray $a[i..r]$ are no smaller than p, each subarray is a proper subarray, and, if $i = j+2$, $a[i-1]$ is equal to p. So, recursive calls $qSort(a,\ell,j)$ and $qsort(a,i,r)$ will complete the sort. We make these recursive calls in a nonstandard fashion; this is discussed below.

Let us see in more detail how the partitioning loops work. In the first iteration of the repeat loop, i does not advance at all but remains at ℓ, and j moves left to the rightmost element no larger than p. So j ends at ℓ or at a larger value; generally, the latter is the case. In either case, we have $i \le j$. We swap $a[i]$ and $a[j]$, increment i, and decrement j. In order to describe the total effect more generally, we distinguish cases.

If p is the unique smallest element of the subarray, j moves all the way to ℓ, the swap has no effect, and $j = \ell - 1$ and $i = \ell + 1$ after the increment and decrement. We have an empty subproblem $\ell..\ell-1$ and a subproblem $\ell+1..r$. Partitioning is complete, and both subproblems are proper subproblems.

If j moves down to $i+1$, we swap, increment i to $\ell+1$, and decrement j to ℓ. Partitioning is complete, and we have the subproblems $\ell..\ell$ and $\ell+1..r$. Both subarrays are proper subarrays.

If j stops at an index larger than $i+1$, we have $\ell < i \le j < r$ after executing the line in Fig. 5.7 marked **(*)**. Also, all elements left of i are at most p (and there is at least one such element), and all elements right of j are at least p (and there is at least one such element). Since the scan loop for i skips only over elements smaller than p and the scan loop for j skips only over elements larger than p, further iterations of the repeat loop maintain this invariant. Also, all further scan loops are guaranteed to terminate by the claims in parentheses and so there is no need for an index-out-of-bounds check in the scan loops. In other words, the scan loops are as concise as possible; they consist of a test and an increment or decrement.

Let us next study how the repeat loop terminates. If we have $i \leq j+2$ after the scan loops, we have $i \leq j$ in the termination test. Hence, we continue the loop. If we have $i = j-1$ after the scan loops, we swap, increment i, and decrement j. So $i = j+1$, and the repeat loop terminates with the proper subproblems $\ell..j$ and $i..r$. The case $i = j$ after the scan loops can occur only if $a[i] = p$. In this case, the swap has no effect. After incrementing i and decrementing j, we have $i = j+2$, resulting in the proper subproblems $\ell..j$ and $j+2..r$, separated by one occurrence of p. Finally, when $i > j$ after the scan loops, then either i goes beyond j in the first scan loop, or j goes below i in the second scan loop. By our invariant, i must stop at $j+1$ in the first case, and then j does not move in its scan loop or j must stop at $i-1$ in the second case. In either case, we have $i = j+1$ after the scan loops. The line marked **(*)** is not executed, so that we have subproblems $\ell..j$ and $i..r$, and both subproblems are proper.

We have now shown that the partitioning step is correct, terminates, and generates proper subproblems.

Exercise 5.22. Is it safe to make the scan loops skip over elements equal to p? Is this safe if it is known that the elements of the array are pairwise distinct?

The refined quicksort handles recursion in a seemingly strange way. Recall that we need to make the recursive calls $qSort(a,\ell,j)$ and $qSort(a,i,r)$. We may make these calls in either order. We exploit this flexibility by making the call for the smaller subproblem first. The call for the larger subproblem would then be the last thing done in *qSort*. This situation is known as *tail recursion* in the programming-language literature. Tail recursion can be eliminated by setting the parameters (ℓ and r) to the right values and jumping to the first line of the procedure. This is precisely what the while loop does. Why is this manipulation useful? Because it guarantees that the recursion stack stays logarithmically bounded; the precise bound is $\lceil \log(n/n_0) \rceil$. This follows from the fact that we make a single recursive call for a subproblem which is at most half the size.

Exercise 5.23. What is the maximal depth of the recursion stack without the "smaller subproblem first" strategy? Give a worst-case example.

***Exercise 5.24 (sorting strings using multikey quicksort [22]).** Let s be a sequence of n strings. We assume that each string ends in a special character that is different from all "normal" characters. Show that the function $mkqSort(s,1)$ below sorts a sequence s consisting of *different* strings. What goes wrong if s contains equal strings? Solve this problem. Show that the expected execution time of *mkqSort* is $O(N + n\log n)$ if $N = \sum_{e \in s} |e|$.

Function *mkqSort*(s : *Sequence* **of** *String*, $i : \mathbb{N}$) : *Sequence* **of** *String*
  assert $\forall e, e' \in s : e[1..i-1] = e'[1..i-1]$
  if $|s| \leq 1$ **then return** s // base case
  pick $p \in s$ uniformly at random // pivot character
  return *concatenation of* $mkqSort(\langle e \in s : e[i] < p[i] \rangle, i)$,
    $mkqSort(\langle e \in s : e[i] = p[i] \rangle, i+1)$, *and*
    $mkqSort(\langle e \in s : e[i] > p[i] \rangle, i)$

Exercise 5.25. Implement several different versions of *qSort* in your favorite programming language. Use and do not use the refinements discussed in this section, and study the effect on running time and space consumption.

5.5 Selection

Selection refers to a class of problems that are easily reduced to sorting but do not require the full power of sorting. Let $s = \langle e_1, \ldots, e_n \rangle$ be a sequence and call its sorted version $s' = \langle e'_1, \ldots, e'_n \rangle$. Selection of the smallest element requires determining e'_1, selection of the largest requires determining e'_n, and selection of the k-th smallest requires determining e'_k. Selection of the median refers to selecting $e_{\lfloor n/2 \rfloor}$. Selection of the median and also of quartiles is a basic problem in statistics. It is easy to determine the smallest element or the smallest and the largest element by a single scan of a sequence in linear time. We now show that the k-th smallest element can also be determined in linear time. The simple recursive procedure shown in Fig. 5.9 solves the problem.

This procedure is akin to quicksort and is therefore called *quickselect*. The key insight is that it suffices to follow one of the recursive calls. As before, a pivot is chosen, and the input sequence s is partitioned into subsequences a, b, and c containing the elements smaller than the pivot, equal to the pivot, and larger than the pivot, respectively. If $|a| \geq k$, we recurse on a, and if $k > |a| + |b|$, we recurse on c with a suitably adjusted k. If $|a| < k \leq |a| + |b|$, the task is solved: the pivot has rank k and we return it. Observe that the latter case also covers the situation $|s| = k = 1$, and hence no special base case is needed. Figure 5.10 illustrates the execution of quickselect.

```
// Find an element with rank k
Function select(s : Sequence of Element; k : ℕ) : Element
    assert |s| ≥ k
    pick p ∈ s uniformly at random                          // pivot key
    a :=⟨e ∈ s : e < p⟩
    if |a| ≥ k then return select(a,k)                      //  a | k
    b :=⟨e ∈ s : e = p⟩
    if |a| + |b| ≥ k then return p                          //  a | b = ⟨p,...,p⟩ | k
    c :=⟨e ∈ s : e > p⟩
    return select(c,k − |a| − |b|)                          //  a | b | c | k
```

Fig. 5.9. Quickselect.

s	k	p	a	b	c
$\langle 3,1,4,5,9,\mathbf{2},6,5,3,5,8 \rangle$	6	2	$\langle 1 \rangle$	$\langle 2 \rangle$	$\langle 3,4,5,9,6,5,3,5,8 \rangle$
$\langle 3,4,5,9,\mathbf{6},5,3,5,8 \rangle$	4	6	$\langle 3,4,5,5,3,4 \rangle$	$\langle 6 \rangle$	$\langle 9,8 \rangle$
$\langle 3,4,\mathbf{5},5,3,5 \rangle$	4	5	$\langle 3,4,3 \rangle$	$\langle 5,5,5 \rangle$	$\langle \rangle$

Fig. 5.10. The execution of $select(\langle 3,1,4,5,9,2,6,5,3,5,8,6 \rangle, 6)$. The middle element (**bold**) of the current s is used as the pivot p

As for quicksort, the worst-case execution time of quickselect is quadratic. But the expected execution time is linear and hence is a logarithmic factor faster than quicksort.

Theorem 5.8. *The quickselect algorithm runs in expected time* $O(n)$ *on an input of size* n.

Proof. We shall give an analysis that is simple and shows a linear expected execution time. It does not give the smallest constant possible. Let $T(n)$ denote the expected execution time of quickselect. We call a pivot *good* if neither $|a|$ nor $|c|$ is larger than $2n/3$. Let γ denote the probability that a pivot is good; then $\gamma \geq 1/3$. We now make the conservative assumption that the problem size in the recursive call is reduced only for good pivots and that, even then, it is reduced only by a factor of $2/3$. Since the work outside the recursive call is linear in n, there is an appropriate constant c such that

$$T(n) \leq cn + \gamma T\left(\frac{2n}{3}\right) + (1-\gamma)T(n).$$

Solving for $T(n)$ yields

$$T(n) \leq \frac{cn}{\gamma} + T\left(\frac{2n}{3}\right) \leq 3cn + T\left(\frac{2n}{3}\right) \leq 3c\left(n + \frac{2n}{3} + \frac{4n}{9} + \ldots\right)$$
$$\leq 3cn \sum_{i \geq 0} \left(\frac{2}{3}\right)^i \leq 3cn \frac{1}{1-2/3} = 9cn. \qquad \square$$

Exercise 5.26. Modify quickselect so that it returns the k smallest elements.

Exercise 5.27. Give a selection algorithm that permutes an array in such a way that the k smallest elements are in entries $a[1], \ldots, a[k]$. No further ordering is required except that $a[k]$ should have rank k. Adapt the implementation tricks used in the array-based quicksort to obtain a nonrecursive algorithm with fast inner loops.

Exercise 5.28 (streaming selection).

(a) Develop an algorithm that finds the k-th smallest element of a sequence that is presented to you one element at a time in an order you cannot control. You have only space $O(k)$ available. This models a situation where voluminous data arrives over a network or at a sensor.

(b) Refine your algorithm so that it achieves a running time $O(n \log k)$. You may want to read some of Chap. 6 first.

*(c) Refine the algorithm and its analysis further so that your algorithm runs in average-case time $O(n)$ if $k = O(n/\log n)$. Here, "average" means that all orders of the elements in the input sequence are equally likely.

5.6 Breaking the Lower Bound

The title of this section is, of course, nonsense. A lower bound is an absolute statement. It states that, in a certain model of computation, a certain task cannot be carried out faster than the bound. So a lower bound cannot be broken. But be careful. It cannot be broken within the model of computation used. The lower bound does not exclude the possibility that a faster solution exists in a richer model of computation. In fact, we may even interpret the lower bound as a guideline for getting faster. It tells us that we must enlarge our repertoire of basic operations in order to get faster.

What does this mean in the case of sorting? So far, we have restricted ourselves to comparison-based sorting. The only way to learn about the order of items was by comparing two of them. For structured keys, there are more effective ways to gain information, and this will allow us to break the $\Omega(n \log n)$ lower bound valid for comparison-based sorting. For example, numbers and strings have structure; they are sequences of digits and characters, respectively.

Let us start with a very simple algorithm *Ksort* that is fast if the keys are small integers, say in the range $0..K-1$. The algorithm runs in time $O(n+K)$. We use an array $b[0..K-1]$ of *buckets* that are initially empty. We then scan the input and insert an element with key k into bucket $b[k]$. This can be done in constant time per element, for example by using linked lists for the buckets. Finally, we concatenate all the nonempty buckets to obtain a sorted output. Figure 5.11 gives the pseudocode. For example, if the elements are pairs whose first element is a key in the range $0..3$ and

$$s = \langle (3,a),(1,b),(2,c),(3,d),(0,e),(0,f),(3,g),(2,h),(1,i) \rangle ,$$

we obtain $b = [\langle (0,e),(0,f) \rangle, \langle (1,b),(1,i) \rangle, \langle (2,c),(2,h) \rangle, \langle (3,a),(3,d),(3,g) \rangle]$ and output $\langle (0,e),(0,f),(1,b),(1,i),(2,c),(2,h),(3,a),(3,d),(3,g) \rangle$. This example illustrates an important property of *Ksort*. It is *stable*, i.e., elements with the same key inherit their relative order from the input sequence. Here, it is crucial that elements are *appended* to their respective bucket.

KSort can be used as a building block for sorting larger keys. The idea behind *radix sort* is to view integer keys as numbers represented by digits in the range $0..K-1$. Then *KSort* is applied once for each digit. Figure 5.12 gives a radix-sorting algorithm for keys in the range $0..K^d-1$ that runs in time $O(d(n+K))$. The elements are first sorted by their least significant digit (*LSD radix sort*), then by the second least significant digit, and so on until the most significant digit is used for sorting. It is not obvious why this works. The correctness rests on the stability of

```
Procedure KSort(s : Sequence of Element)
    b = ⟨⟨⟩,...,⟨⟩⟩ : Array [0..K−1] of Sequence of Element
    foreach e ∈ s do b[key(e)].pushBack(e)
    s := concatenation of b[0],...,b[K−1]
```

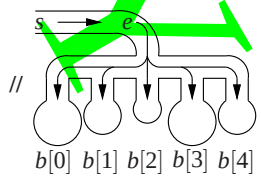


Fig. 5.11. Sorting with keys in the range $0..K-1$.

```
Procedure LSDRadixSort(s : Sequence of Element)
    for i := 0 to d − 1 do                                   digits
        redefine key(x) as (x div K^i) mod K          // x  d−1 | ··· | i | ··· | 1 | 0
        KSort(s)                                              key(x)
        invariant s is sorted with respect to digits i..0
```

Fig. 5.12. Sorting with keys in $0..K^d - 1$ using **l**east **s**ignificant **d**igit (LSD) radix sort.

```
Procedure uniformSort(s : Sequence of Element)
    n := |s|
    b = ⟨⟨⟩, ..., ⟨⟩⟩ : Array [0..n − 1] of Sequence of Element
    foreach e ∈ s do b[⌊key(e) · n⌋].pushBack(e)
    for i := 0 to n − 1 do sort b[i] in time O(|b[i]| log |b[i]|)
    s := concatenation of b[0], ..., b[n − 1]
```

Fig. 5.13. Sorting random keys in the range $[0,1)$.

Ksort. Since *KSort* is stable, the elements with the same i-th digit remain sorted with respect to digits $i-1..0$ during the sorting process with respect to digit i. For example, if $K = 10$, $d = 3$, and

$$s = \langle 017, 042, 666, 007, 111, 911, 999 \rangle, \text{ we successively obtain}$$
$$s = \langle 11\mathbf{1}, 91\mathbf{1}, 04\mathbf{2}, 66\mathbf{6}, 01\mathbf{7}, 00\mathbf{7}, 99\mathbf{9} \rangle ,$$
$$s = \langle 0\mathbf{0}7, 1\mathbf{1}1, 9\mathbf{1}1, 0\mathbf{1}7, 0\mathbf{4}2, 6\mathbf{6}6, 9\mathbf{9}9 \rangle , \text{ and}$$
$$s = \langle \mathbf{0}07, \mathbf{0}17, \mathbf{0}42, \mathbf{1}11, \mathbf{6}66, \mathbf{9}11, \mathbf{9}99 \rangle .$$

Radix sort starting with the most significant digit (*MSD radix sort*) is also possible. We apply *KSort* to the most significant digit and then sort each bucket recursively. The only problem is that the buckets might be much smaller than K, so that it would be expensive to apply *KSort* to small buckets. We then have to switch to another algorithm. This works particularly well if we can assume that the keys are uniformly distributed. More specifically, let us now assume that the keys are real numbers with $0 \le key(e) < 1$. The algorithm *uniformSort* in Fig. 5.13 scales these keys to integers between 0 and $n-1 = |s|-1$, and groups them into n buckets, where bucket $b[i]$ is responsible for keys in the range $[i/n, (i+1)/n)$. For example, if $s = \langle 0.8, 0.4, 0.7, 0.6, 0.3 \rangle$, we obtain five buckets responsible for intervals of size 0.2, and

$$b = [\langle\rangle, \quad \langle 0.3 \rangle, \quad \langle 0.4 \rangle, \quad \langle 0.7, 0.6 \rangle, \quad \langle 0.8 \rangle] ;$$

only $b[3] = \langle 0.7, 0.6 \rangle$ is a nontrivial subproblem. *uniformSort* is very efficient for *random* keys.

Theorem 5.9. *If the keys are independent uniformly distributed random values in* $[0,1)$*, uniformSort sorts n keys in expected time* $O(n)$ *and worst-case time* $O(n \log n)$.

Proof. We leave the worst-case bound as an exercise and concentrate on the average case. The total execution time T is $O(n)$ for setting up the buckets and concatenating the sorted buckets, plus the time for sorting the buckets. Let T_i denote the time for sorting the i-th bucket. We obtain

$$E[T] = O(n) + E\left[\sum_{i<n} T_i\right] = O(n) + \sum_{i<n} E[T_i] = O(n) + nE[T_0] \ .$$

The second equality follows from the linearity of expectations (A.2), and the third equality uses the fact that all bucket sizes have the same distribution for uniformly distributed inputs. Hence, it remains to show that $E[T_0] = O(1)$. We shall prove the stronger claim that $E[T_0] = O(1)$ even if a quadratic-time algorithm such as insertion sort is used for sorting the buckets. The analysis is similar to the arguments used to analyze the behavior of hashing in Chap. 4.

Let $B_0 = |b[0]|$. We have $E[T_0] = O\big(E[B_0^2]\big)$. The random variable B_0 obeys a binomial distribution (A.7) with n trials and success probability $1/n$, and hence

$$\text{prob}(B_0 = i) = \binom{n}{i}\left(\frac{1}{n}\right)^i\left(1-\frac{1}{n}\right)^{n-i} \le \frac{n^i}{i!}\frac{1}{n^i} = \frac{1}{i!} \le \left(\frac{e}{i}\right)^i \ ,$$

where the last inequality follows from Stirling's approximation to the factorial (A.9). We obtain

$$\begin{aligned} E[B_0^2] &= \sum_{i\le n} i^2 \text{prob}(B_0 = i) \le \sum_{i\le n} i^2 \left(\frac{e}{i}\right)^i \\ &\le \sum_{i\le 5} i^2 \left(\frac{e}{i}\right)^i + e^2 \sum_{i\ge 6} \left(\frac{e}{i}\right)^{i-2} \\ &\le O(1) + e^2 \sum_{i\ge 6} \left(\frac{1}{2}\right)^{i-2} = O(1) \ , \end{aligned}$$

and hence $E[T] = O(n)$ (note that the split at $i = 6$ allows us to conclude that $e/i \le 1/2$). □

***Exercise 5.29.** Implement an efficient sorting algorithm for elements with keys in the range $0..K-1$ that uses the data structure of Exercise 3.20 for the input and output. The space consumption should be $n + O(n/B + KB)$ for n elements, and blocks of size B.

5.7 *External Sorting

Sometimes the input is so huge that it does not fit into internal memory. In this section, we shall learn how to sort such data sets in the external-memory model introduced in Sect. 2.2. This model distinguishes between a fast internal memory of size M and a large external memory. Data is moved in the memory hierarchy in

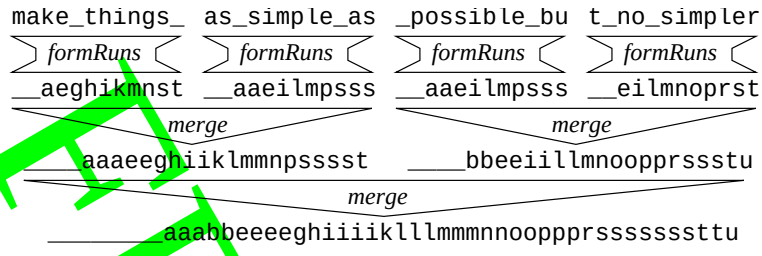


Fig. 5.14. An example of two-way mergesort with initial runs of length 12

blocks of size B. Scanning data is fast in external memory and mergesort is based on scanning. We therefore take mergesort as the starting point for external-memory sorting.

Assume that the input is given as an array in external memory. We shall describe a nonrecursive implementation for the case where the number of elements n is divisible by B. We load subarrays of size M into internal memory, sort them using our favorite algorithm, for example *qSort*, and write the sorted subarrays back to external memory. We refer to the sorted subarrays as *runs*. The *run formation phase* takes n/B block reads and n/B block writes, i.e., a total of $2n/B$ I/Os. We then merge pairs of runs into larger runs in $\lceil \log(n/M) \rceil$ *merge phases*, ending up with a single sorted run. Figure 5.14 gives an example for $n = 48$ and runs of length 12.

How do we merge two runs? We keep one block from each of the two input runs and from the output run in internal memory. We call these blocks *buffers*. Initially, the input buffers are filled with the first B elements of the input runs, and the output buffer is empty. We compare the leading elements of the input buffers and move the smaller element to the output buffer. If an input buffer becomes empty, we fetch the next block of the corresponding input run; if the output buffer becomes full, we write it to external memory.

Each merge phase reads all current runs and writes new runs of twice the length. Therefore, each phase needs n/B block reads and n/B block writes. Summing over all phases, we obtain $(2n/B)(1 + \lceil \log n/M \rceil)$ I/Os. This technique works provided that $M \geq 3B$.

5.7.1 Multiway Mergesort

In general, internal memory can hold many blocks and not just three. We shall describe how to make full use of the available internal memory during merging. The idea is to merge more than just two runs; this will reduce the number of phases. In *k-way merging*, we merge k sorted sequences into a single output sequence. In each step we find the input sequence with the smallest first element. This element is removed and appended to the output sequence. External-memory implementation is easy as long as we have enough internal memory for k input buffer blocks, one output buffer block, and a small amount of additional storage.

For each sequence, we need to remember which element we are currently considering. To find the smallest element out of all k sequences, we keep their current elements in a *priority queue*. A priority queue maintains a set of elements supporting the operations of insertion and deletion of the minimum. Chapter 6 explains how priority queues can be implemented so that insertion and deletion take time $O(\log k)$ for k elements. The priority queue tells us at each step, which sequence contains the smallest element. We delete this element from the priority queue, move it to the output buffer, and insert the next element from the corresponding input buffer into the priority queue. If an input buffer runs dry, we fetch the next block of the corresponding sequence, and if the output buffer becomes full, we write it to the external memory.

How large can we choose k? We need to keep $k+1$ blocks in internal memory and we need a priority queue for k keys. So we need $(k+1)B + O(k) \leq M$ or $k = O(M/B)$. The number of merging phases is reduced to $\lceil \log_k(n/M) \rceil$, and hence the total number of I/Os becomes

$$2\frac{n}{B}\left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil\right) . \tag{5.1}$$

The difference from binary merging is the much larger base of the logarithm. Interestingly, the above upper bound for the I/O complexity of sorting is also a lower bound [5], i.e., under fairly general assumptions, no external sorting algorithm with fewer I/O operations is possible.

In practice, the number of merge phases will be very small. Observe that a single merge phase suffices as long as $n \leq M^2/B$. We first form M/B runs of length M each and then merge these runs into a single sorted sequence. If internal memory stands for DRAM and "external memory" stands for hard disks, this bound on n is no real restriction, for all practical system configurations.

Exercise 5.30. Show that a multiway mergesort needs only $O(n \log n)$ element comparisons.

Exercise 5.31 (balanced systems). Study the current market prices of computers, internal memory, and mass storage (currently hard disks). Also, estimate the block size needed to achieve good bandwidth for I/O. Can you find any configuration where multiway mergesort would require more than one merging phase for sorting an input that fills all the disks in the system? If so, what fraction of the cost of that system would you have to spend on additional internal memory to go back to a single merging phase?

5.7.2 Sample Sort

The most popular internal-memory sorting algorithm is not mergesort but quicksort. So it is natural to look for an external-memory sorting algorithm based on quicksort. We shall sketch *sample sort*. In expectation, it has the same performance guarantees as multiway mergesort (5.1). Sample sort is easier to adapt to parallel disks and

parallel processors than merging-based algorithms. Furthermore, similar algorithms can be used for fast external sorting of integer keys along the lines of Sect. 5.6.

Instead of the single pivot element of quicksort, we now use $k-1$ *splitter elements* $s_1,\ldots,s_{k-1}$ to split an input sequence into k output sequences, or *buckets*. Bucket i gets the elements e for which $s_{i-1} \le e < s_i$. To simplify matters, we define the artificial splitters $s_0 = -\infty$ and $s_k = \infty$ and assume that all elements have different keys. The splitters should be chosen in such a way that the buckets have a size of roughly n/k. The buckets are then sorted recursively. In particular, buckets that fit into the internal memory can subsequently be sorted internally. Note the similarity to MSD-radix sort described in Sect. 5.6.

The main challenge is to find good splitters quickly. Sample sort uses a fast, simple randomized strategy. For some integer a, we randomly choose $(a+1)k-1$ *sample* elements from the input. The sample S is then sorted internally, and we define the splitters as $s_i = S[(a+1)i]$ for $1 \le i \le k-1$, i.e., consecutive splitters are separated by a samples, the first splitter is preceded by a samples, and the last splitter is followed by a samples. Taking $a = 0$ results in a small sample set, but the splitting will not be very good. Moving all elements to the sample will result in perfect splitters, but the sample will be too big. The following analysis shows that setting $a = \mathrm{O}(\log k)$ achieves roughly equal bucket sizes at low cost for sampling and sorting the sample.

The most I/O-intensive part of sample sort is the k-way distribution of the input sequence to the buckets. We keep one buffer block for the input sequence and one buffer block for each bucket. These buffers are handled analogously to the buffer blocks in k-way merging. If the splitters are kept in a sorted array, we can find the right bucket for an input element e in time $\mathrm{O}(\log k)$ using binary search.

Theorem 5.10. *Sample sort uses*

$$\mathrm{O}\left(\frac{n}{B}\left(1+\left\lceil \log_{M/B}\frac{n}{M}\right\rceil\right)\right)$$

expected I/O steps for sorting n elements. The internal work is $\mathrm{O}(n\log n)$.

We leave the detailed proof to the reader and describe only the key ingredient of the analysis here. We use $k = \Theta(\min(n/M, M/B))$ buckets and a sample of size $\mathrm{O}(k\log k)$. The following lemma shows that with this sample size, it is unlikely that any bucket has a size much larger than the average. We hide the constant factors behind $\mathrm{O}(\cdot)$ notation because our analysis is not very tight in this respect.

Lemma 5.11. *Let $k \ge 2$ and $a+1 = 12\ln k$. A sample of size $(a+1)k-1$ suffices to ensure that no bucket receives more than $4n/k$ elements with probability at least $1/2$.*

Proof. As in our analysis of quicksort (Theorem 5.6), it is useful to study the sorted version $s' = \langle e'_1,\ldots,e'_n\rangle$ of the input. Assume that there is a bucket with at least $4n/k$ elements assigned to it. We estimate the probability of this event.

We split s' into $k/2$ segments of length $2n/k$. The j-th segment t_j contains elements $e'_{2jn/k+1}$ to $e'_{2(j+1)n/k}$. If $4n/k$ elements end up in some bucket, there must be some segment t_j such that all its elements end up in the same bucket. This can only

happen if fewer than $a+1$ samples are taken from t_j, because otherwise at least one splitter would be chosen from t_j and its elements would not end up in a single bucket. Let us concentrate on a fixed j.

We use a random variable X to denote the number of samples taken from t_j. Recall that we take $(a+1)k-1$ samples. For each sample i, $1 \le i \le (a+1)k-1$, we define an indicator variable X_i with $X_i = 1$ if the i-th sample is taken from t_j and $X_i = 0$ otherwise. Then $X = \sum_{1 \le i \le (a+1)k-1} X_i$. Also, the X_i's are independent, and $\text{prob}(X_i = 1) = 2/k$. Independence allows us to use the Chernoff bound (A.5) to estimate the probability that $X < a+1$. We have

$$\mathrm{E}[X] = ((a+1)k-1) \cdot \frac{2}{k} = 2(a+1) - \frac{2}{k} \ge \frac{3(a+1)}{2} .$$

Hence $X < a+1$ implies $X < (1-1/3)\mathrm{E}[X]$, and so we can use (A.5) with $\varepsilon = 1/3$. Thus

$$\text{prob}(X < a+1) \le e^{-(1/9)\mathrm{E}[X]/2} \le e^{-(a+1)/12} = e^{-\ln k} = \frac{1}{k} .$$

The probability that an insufficient number of samples is chosen from a fixed t_j is thus at most $1/k$, and hence the probability that an insufficient number is chosen from some t_j is at most $(k/2) \cdot (1/k) = 1/2$. Thus, with probability at least $1/2$, each bucket receives fewer than $4n/k$ elements. □

Exercise 5.32. Work out the details of an external-memory implementation of sample sort. In particular, explain how to implement multiway distribution using $2n/B + k + 1$ I/O steps if the internal memory is large enough to store $k+1$ blocks of data and $O(k)$ additional elements.

Exercise 5.33 (many equal keys). Explain how to generalize multiway distribution so that it still works if some keys occur very often. Hint: there are at least two different solutions. One uses the sample to find out which elements are frequent. Another solution makes all elements unique by interpreting an element e at an input position i as the pair (e,i).

***Exercise 5.34 (more accurate distribution).** A larger sample size improves the quality of the distribution. Prove that a sample of size $O\big((k/\varepsilon^2)\log(k/\varepsilon m)\big)$ guarantees, with probability (at least $1-1/m$), that no bucket has more than $(1+\varepsilon)n/k$ elements. Can you get rid of the ε in the logarithmic factor?

5.8 Implementation Notes

Comparison-based sorting algorithms are usually available in standard libraries, and so you may not have to implement one yourself. Many libraries use tuned implementations of quicksort.

Canned non-comparison-based sorting routines are less readily available. Figure 5.15 shows a careful array-based implementation of *Ksort*. It works well for

```
Procedure KSortArray(a,b : Array [1..n] of Element)
    c = ⟨0,...,0⟩ : Array [0..K−1] of ℕ                    // counters for each bucket
    for i := 1 to n do c[key(a[i])]++                      // Count bucket sizes

    C := 0
    for k := 0 to K−1 do (C,c[k]) := (C+c[k],C)            // Store ∑_{i<k} c[k] in c[k].

    for i := 1 to n do                                     // Distribute a[i]
        b[c[key(a[i])]] := a[i]
        c[key(a[i])]++
```

Fig. 5.15. Array-based sorting with keys in the range $0..K-1$. The input is an unsorted array a. The output is b, containing the elements of a in sorted order. We first count the number of inputs for each key. Then we form the partial sums of the counts. Finally, we write each input element to the correct position in the output array.

small to medium-sized problems. For large K and n, it suffers from the problem that the distribution of elements to the buckets may cause a cache fault for every element.

To fix this problem, one can use multiphase algorithms similar to MSD radix sort. The number K of output sequences should be chosen in such a way that one block from each bucket is kept in the cache (see also [134]). The distribution degree K can be larger when the subarray to be sorted fits into the cache. We can then switch to a variant of *uniformSort* (see Fig. 5.13).

Another important practical aspect concerns the type of elements to be sorted. Sometimes we have rather large elements that are sorted with respect to small keys. For example, you may want to sort an employee database by last name. In this situation, it makes sense to first extract the keys and store them in an array together with pointers to the original elements. Then, only the key–pointer pairs are sorted. If the original elements need to be brought into sorted order, they can be permuted accordingly in linear time using the sorted key–pointer pairs.

Multiway merging of a small number of sequences (perhaps up to eight) deserves special mention. In this case, the priority queue can be kept in the processor registers [160, 206].

5.8.1 C/C++

Sorting is one of the few algorithms that is part of the C standard library. However, the C sorting routine *qsort* is slower and harder to use than the C++ function *sort*. The main reason is that the comparison function is passed as a function pointer and is called for every element comparison. In contrast, *sort* uses the template mechanism of C++ to figure out at compile time how comparisons are performed so that the code generated for comparisons is often a single machine instruction. The parameters passed to *sort* are an iterator pointing to the start of the sequence to be sorted, and an iterator pointing after the end of the sequence. In our experiments using an Intel Pentium III and GCC 2.95, *sort* on arrays ran faster than our manual implementation of quicksort. One possible reason is that compiler designers may tune their

code optimizers until they find that good code for the library version of quicksort is generated. There is an efficient parallel-disk external-memory sorter in STXXL [48], an external-memory implementation of the STL. Efficient parallel sorters (parallel quicksort and parallel multiway mergesort) for multicore machines are available with the **M**ulti-**C**ore **S**tandard **T**emplate **L**ibrary [180, 125].

Exercise 5.35. Give a C or C++ implementation of the procedure *qSort* in Fig. 5.7. Use only two parameters: a pointer to the (sub)array to be sorted, and its size.

5.8.2 Java

The Java 6 platform provides a method *sort* which implements a stable binary mergesort for *Arrays* and *Collections*. One can use a customizable *Comparator*, but there is also a default implementation for all classes supporting the interface *Comparable*.

5.9 Historical Notes and Further Findings

In later chapters, we shall discuss several generalizations of sorting. Chapter 6 discusses priority queues, a data structure that supports insertions of elements and removal of the smallest element. In particular, inserting n elements followed by repeated deletion of the minimum amounts to sorting. Fast priority queues result in quite good sorting algorithms. A further generalization is the *search trees* introduced in Chap. 7, a data structure for maintaining a sorted list that allows searching, inserting, and removing elements in logarithmic time.

We have seen several simple, elegant, and efficient randomized algorithms in this chapter. An interesting question is whether these algorithms can be replaced by deterministic ones. Blum et al. [25] described a deterministic median selection algorithm that is similar to the randomized algorithm discussed in Sect. 5.5. This deterministic algorithm makes pivot selection more reliable using recursion: it splits the input set into subsets of five elements, determines the median of each subset by sorting the five-element subset, then determines the median of the $n/5$ medians by calling the algorithm recursively, and finally uses the median of the medians as the splitter. The resulting algorithm has linear worst-case execution time, but the large constant factor makes the algorithm impractical. (We invite the reader to set up a recurrence for the running time and to show that it has a linear solution.)

There are quite practical ways to reduce the expected number of comparisons required by quicksort. Using the median of three random elements yields an algorithm with about $1.188n\log n$ comparisons. The median of three medians of three-element subsets brings this down to $\approx 1.094n\log n$ [20]. The number of comparisons can be reduced further by making the number of elements considered for pivot selection dependent on the size of the subproblem. Martinez and Roura [123] showed that for a subproblem of size m, the median of $\Theta(\sqrt{m})$ elements is a good choice for the pivot. With this approach, the total number of comparisons becomes $(1+o(1))n\log n$, i.e., it matches the lower bound of $n\log n - O(n)$ up to lower-order terms. Interestingly,

the above optimizations can be counterproductive. Although fewer instructions are executed, it becomes impossible to predict when the inner while loops of quicksort will be aborted. Since modern, deeply pipelined processors only work efficiently when they can predict the directions of branches taken, the net effect on performance can even be negative [102]. Therefore, in [167] , a comparison-based sorting algorithm that avoids conditional branch instructions was developed. An interesting deterministic variant of quicksort is proportion-extend sort [38].

A classical sorting algorithm of some historical interest is *Shell sort* [174, 100], a generalization of insertion sort, that gains efficiency by also comparing nonadjacent elements. It is still open whether some variant of Shell sort achieves $O(n \log n)$ average running time [100, 124].

There are some interesting techniques for improving external multiway mergesort. The *snow plow* heuristic [112, Sect. 5.4.1] forms runs of expected size $2M$ using a fast memory of size M: whenever an element is selected from the internal priority queue and written to the output buffer and the next element in the input buffer can extend the current run, we add it to the priority queue. Also, the use of *tournament trees* instead of general priority queues leads to a further improvement of multiway merging [112].

Parallelism can be used to improve the sorting of very large data sets, either in the form of a uniprocessor using parallel disks or in the form of a multiprocessor. Multiway mergesort and distribution sort can be adapted to D parallel disks by *striping*, i.e., any D consecutive blocks in a run or bucket are evenly distributed over the disks. Using randomization, this idea can be developed into almost optimal algorithms that also overlap I/O and computation [49]. The sample sort algorithm of Sect. 5.7.2 can be adapted to parallel machines [24] and results in an efficient parallel sorter.

We have seen linear-time algorithms for highly structured inputs. A quite general model, for which the $n \log n$ lower bound does not hold, is the *word model*. In this model, keys are integers that fit into a single memory cell, say 32- or 64-bit keys, and the standard operations on words (bitwise-AND, bitwise-OR, addition, ...) are available in constant time. In this model, sorting is possible in deterministic time $O(n \log \log n)$ [11]. With randomization, even $O(n\sqrt{\log \log n})$ is possible [85]. *Flash sort* [149] is a distribution-based algorithm that works almost in-place.

Exercise 5.36 (Unix spellchecking). Assume you have a dictionary consisting of a sorted sequence of correctly spelled words. To check a text, you convert it to a sequence of words, sort it, scan the text and dictionary simultaneously, and output the words in the text that do not appear in the dictionary. Implement this spellchecker using Unix tools in a small number of lines of code. Can you do this in one line?