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

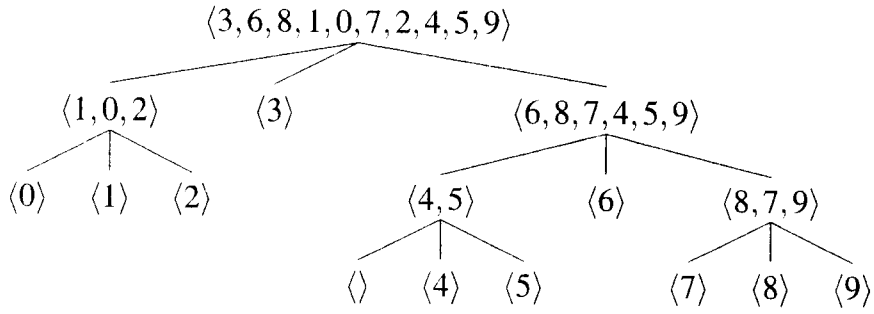

Fig. 5.6. Execution of *quickSort* (Fig. 5.5) on $\langle 3,6,8,1,0,7,2,4,5,9\rangle$ using the first element of a subsequence as the pivot. The first call of quicksort uses 3 as the pivot and generates the subproblems $\langle 1,0,2\rangle$, $\langle 3\rangle$, and $\langle 6,8,7,4,5,9\rangle$. The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

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

It is easy to verify by induction that

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

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, $X_i \leq n-1$, 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)
    while r − ℓ + 1 > n0 do                          // Sort the subarray a[ℓ..r]
                                                     // 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
        repeat                                       // a: [ℓ  i→  ←j  r]
            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 $i \leq j$. 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 $\ell < i \leq j < r$ 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)
    b := ⟨e ∈ s : e = p⟩
    if |a| + |b| ≥ k then return p
    c := ⟨e ∈ s : e > p⟩
    return select(c, k − |a| − |b|)
```

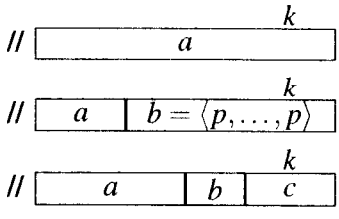

Fig. 5.9. Quickselect

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

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

□

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]
```

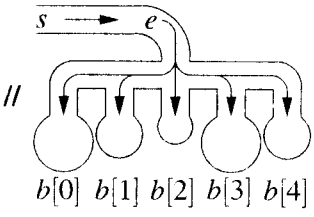

Fig. 5.11. Sorting with keys in the range $0..K-1$

```
Procedure LSDRadixSort(s : Sequence of Element)
  for i := 0 to d − 1 do
    redefine key(x) as (x div K^i) mod K          // x  ... [ i ]  digits
    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 least significant digit (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

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

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 $0 \leq key(e) < 1$. 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

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

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

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

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

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

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

***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

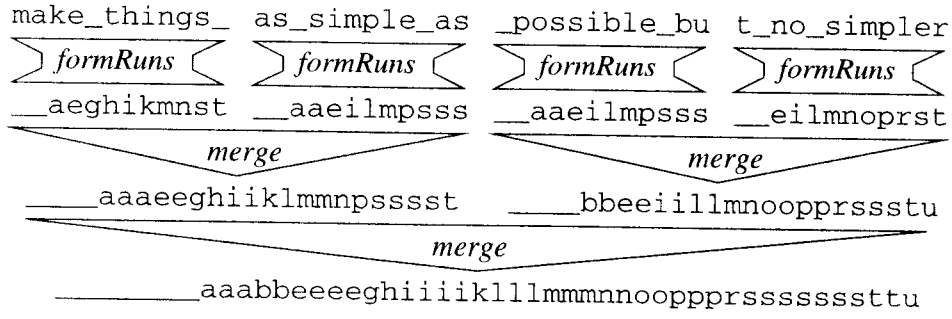

Fig. 5.14. An example of two-way mergesort with initial runs of length 12

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 $a = O(\log k)$ 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 $O(\log k)$ using binary search.

Theorem 5.10. *Sample sort uses*

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

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

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 $O(k\log k)$. 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 $O(\cdot)$ 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 Σ_{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