10
Sorting

10.1 A Sequence of Sorting Algorithms

Arranging the elements of a data set according to some specific order is one of the most basic tasks a program might perform, and there are even claims that most of the CPU time of many computers is spent on sorting. Although the relevant algorithms cannot be identified with some specific data structure, the topics connected to sorting are important enough to deserve being treated here in their own chapter. There is no intention to compile an exhaustive inventory of sorting techniques, but rather to study some noteworthy ones and related matters.

There is no need for the existence of a computer to understand the importance and advantage of keeping some items we wish to process in order, and the action of sorting is probably as old as mankind. Here is a several thousand years old written evidence:

He searched, beginning with the oldest and ending with the youngest, . . .
Genesis *44:12, New American Standard Bible*

To facilitate the discussion, we shall use numerical data in our examples, to be arranged into ascending sequences, but the order could of course also be reversed, the data can be alphabetic, and moreover, the same set of items may be arranged differently according to varying criteria. For example, the orders of months

January, February, March, April, May, June;
April, February, January, June, March, May;
February, April, June, January, March, May;
May, June, March, April, January, February

are, respectively, chronological, alphabetical, sorted by the lengths of the months or by the lengths of their names.

Some sorting methods have already been mentioned earlier: Mergesort in Chapter 2 and Heapsort in Section 7.5. Both have a worst case time complexity of $O(n \log n)$ for an input of n elements, which, as we shall see, is the best one can expect. The simpler methods, however, those used in real life applications rather than in programs, are generally less efficient and require up to $\Omega(n^2)$ comparisons. Examples are the way a card player incrementally updates her or his hand by adding new cards into their proper place (Insertion sort), or arranging a team of game players by their increasing heights, allowing only neighbors to swap places (Bubble sort).

The following technique generates a sequence of sorting methods with gradually improving performances. For all of them, the input sequence is supposed to be given in an array A_1 of size n. We start with a method of repeatedly removing the largest element from the remaining set, using $i-1$ comparisons to find the maximum in the ith iteration, and storing the removed items in the order of their processing in another array B. This is called Selection sort and requires $(n-1)+(n-2)+\cdots=\sum_{i=1}^{n-1} i = O(n^2)$ comparisons.

A better performance can be achieved by partitioning the set A_1 into $\sqrt{n}$ subsets of $\sqrt{n}$ elements each (rounding, if necessary). The maximal element m can then be retrieved in two stages. We first find the maximum in each subset, using $\sqrt{n}-1$ comparisons for each of them, and define the set A_2 of these maxima. The maximum element of A_1 is then the maximum of A_2, which again can be found in time $\sqrt{n}-1$. So the total time to find m is

$$\sqrt{n} \times (\sqrt{n}-1) + \sqrt{n} - 1 = n - 1,$$

just as before. However, for the second element of A_1, we need only to locate the subset from which the maximum originated, repeat the search in this subset, as well as again in A_2. This takes only $2(\sqrt{n}-1)$ comparisons.

As example, refer again to the expansion of π and take the 25 first pairs of digits after the decimal point, yielding the set A_1 of Figure 10.1. The number of comparisons to get $m = 93$ is 24, but for the second largest element, we retrieve again the maximum of the last subset {69, 39, 75, 10}, after having removed 93, and then from the updated set A_2, {92, 89, 79, 84, 75}.

All the subsequent elements can be found similarly in $2(\sqrt{n}-1)$ comparisons, so the total time for sorting the set is

$$(n-1) + (n-1) \times 2(\sqrt{n}-1) \simeq 2n\sqrt{n}.$$

To further improve the complexity, we can try a similar solution in three instead of only two stages. The set A_1 is partitioned into subsets of size $\sqrt[3]{n}$, so the number of subsets will be $\sqrt[3]{n}^2$. The maximum elements of these subsets are stored in A_2, which in turn is partitioned itself into $\sqrt[3]{n}$ subsets of size $\sqrt[3]{n}$, and their maxima will form the set A_3, as can be seen in Figure 10.2. The largest element is now found by

$$\sqrt[3]{n}^2(\sqrt[3]{n}-1)+\sqrt[3]{n}(\sqrt[3]{n}-1)+\sqrt[3]{n}-1=n-1$$

comparisons, as in the 2-stage algorithm. To find the second largest element, one only needs to re-evaluate the maximum of one subset of size $\sqrt[3]{n}$ for each level, thus using $3(\sqrt[3]{n}-1)$ comparisons. The same is true also for the subsequent elements. For the example in Figure 10.2, after removing the global maximum 93, the three processed subsets to find the second element are {39, 75} in A_1, {71, 75, 10} in A_2 and $A_3 = \{92, 84, 75\}$. The complexity of this 3-stage approach is therefore

$$(n-1)+(n-1)\times 3(\sqrt[3]{n}-1)\simeq 3n\sqrt[3]{n}.$$

In fact, since increasing the number of stages seems to reduce the number of requested comparisons, there is no reason to stop here. In the general case, k layers representing the sets $A_1, A_2, \ldots, A_k$ are constructed, and the set A_i, for $1 \le i < k$ is partitioned into $n^{(k-i)/k}$ subsets of size $n^{1/k}$. The maximal element of each of the subsets of A_i is taken to form the set A_{i+1} of the next layer. The number of comparisons to find the global maximum is then $n-1$, and for the subsequent elements it is $kn^{1/k}$, yielding a total complexity of about $kn^{1+1/k}$ for the k-layer sort. This is a decreasing function of k, which suggests to let k grow as large as possible. There is, however, an upper limit for k, because the size of the subsets in the partitions is $n^{1/k}$, which is also a decreasing function of k. To have any comparisons at all, this size has to be at least 2, and from $n^{1/k} = 2$ one can derive that the largest possible value of k is $\log_2 n$.

For our example of 25 elements, the number of layers will be $\lceil \log_2 25 \rceil = 5$, and each subset will be of size 2, as depicted in Figure 10.3. In fact, if the maximum element 93 is added as a single element in an additional layer A_6, this structure can be seen as a binary tree in which each node is the maximum of its two children – this is the definition of a *heap*. Finding the maximum element is equivalent to building the heap, and can be done in time $O(n)$, but then repeatedly removing the maximal element requires only one comparison for each of the $\log_2 n$ layers, for a total of $O(n \log n)$ comparisons. Building a heap and then maintaining the heap property while removing the elements in order is the basic idea of Heapsort we have seen in Section 7.5.

There are some technical differences between the heaps we studied in Chapter 7 and the heap resulting from the preceding derivation process, as given in

$$\begin{array}{l ccccc|ccccc|ccccc|ccccc|ccccc}
A_2 & \multicolumn{5}{c}{92} & \multicolumn{5}{c}{89} & \multicolumn{5}{c}{79} & \multicolumn{5}{c}{84} & \multicolumn{5}{c}{93} \\
A_1 & 14 & 15 & 92 & 65 & 35 & 89 & 79 & 32 & 38 & 46 & 26 & 43 & 38 & 32 & 79 & 50 & 28 & 84 & 19 & 71 & 69 & 39 & 93 & 75 & 10
\end{array}$$

Figure 10.1. Partition into $\sqrt{n}$ subsets of $\sqrt{n}$ elements.

$$\begin{array}{l ccc|ccc|ccc|ccc|ccc|ccc|ccc|ccc|c}
A_3 & \multicolumn{9}{c}{92} & \multicolumn{9}{c}{84} & \multicolumn{7}{c}{93} \\
A_2 & \multicolumn{3}{c}{92} & \multicolumn{3}{c}{89} & \multicolumn{3}{c|}{79} & \multicolumn{3}{c}{46} & \multicolumn{3}{c}{79} & \multicolumn{3}{c|}{84} & \multicolumn{3}{c}{71} & \multicolumn{3}{c}{93} & \multicolumn{1}{c}{10} \\
A_1 & 14 & 15 & 92 & 65 & 35 & 89 & 79 & 32 & 38 & 46 & 26 & 43 & 38 & 32 & 79 & 50 & 28 & 84 & 19 & 71 & 69 & 39 & 93 & 75 & 10
\end{array}$$

Figure 10.2. Partition into $\sqrt[3]{n}^2$ subsets of $\sqrt[3]{n}$ elements.

$$\begin{array}{l cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|c}
A_5 & \multicolumn{16}{c}{92} & \multicolumn{9}{c}{93} \\
A_4 & \multicolumn{8}{c}{92} & \multicolumn{8}{c|}{79} & \multicolumn{8}{c}{93} & \multicolumn{1}{c}{10} \\
A_3 & \multicolumn{4}{c}{92} & \multicolumn{4}{c|}{89} & \multicolumn{4}{c}{46} & \multicolumn{4}{c|}{79} & \multicolumn{4}{c}{84} & \multicolumn{4}{c|}{93} & \multicolumn{1}{c}{10} \\
A_2 & \multicolumn{2}{c}{15} & \multicolumn{2}{c|}{92} & \multicolumn{2}{c}{89} & \multicolumn{2}{c|}{79} & \multicolumn{2}{c}{46} & \multicolumn{2}{c|}{43} & \multicolumn{2}{c}{38} & \multicolumn{2}{c|}{79} & \multicolumn{2}{c}{84} & \multicolumn{2}{c|}{71} & \multicolumn{2}{c}{69} & \multicolumn{2}{c|}{93} & \multicolumn{1}{c}{10} \\
A_1 & 14 & 15 & 92 & 65 & 35 & 89 & 79 & 32 & 38 & 46 & 26 & 43 & 38 & 32 & 79 & 50 & 28 & 84 & 19 & 71 & 69 & 39 & 93 & 75 & 10
\end{array}$$

Figure 10.3. Partition into subsets of two elements – Heapsort.

Figure 10.3. All the elements of the original set A_1 appear here in the lowest layer, and the other sets contain just copies of some of the elements, whereas the heaps of Chapter 7 had no duplicates. The difference has thus some similarity with the difference between B-trees and B$^+$-trees of Chapter 6. Moreover, the update process here proceeds bottom-up in each iteration. By contrast, the definition in Chapter 7 insisted on representing a heap as a full binary tree, so to remove the root, it was swapped with a leaf and the comparisons were performed top-down. Nevertheless, the algorithm and its complexity are essentially the same.

Summarizing, we have seen a family of connected sorting procedures, starting with the trivial Selection sort which requires quadratic time, and leading naturally to Heapsort with its optimal worst case complexity of $O(n \log n)$.

10.2 Lower Bound on the Worst Case

An obvious question arises, and not only for sorting algorithms: how can we know that an algorithm we found for a given problem is reasonable in terms of its time complexity? Sorting by Bubblesort in time $O(n^2)$, for instance, might have looked as a good solution before a better one, like Mergesort, has been discovered, with time $O(n \log n)$. So maybe somebody will come up with an even faster sorting procedure?

This is not just a theoretical question. If I found some solution to a problem, but my boss is not satisfied with its performance, I might invest a major effort to design an improved algorithm, which may not always be justified. If one can prove mathematically that the suggested solution is already *optimal*, the waste of time could be avoided. However, it is often not possible to show such optimality, and then we might settle for finding a so-called lower bound for a given problem.

A *lower bound* for a problem P with input size n is a function $B(n)$ such that it can be shown that no algorithm solving P in time less than $B(n)$ can possibly exist. Note that this does not imply the existence of an algorithm that does solve P in time $B(n)$. For example, it might be possible to show that a lower bound for P is n^2, but that the best known algorithms for its solution require $\Omega(n^3)$. Much of the research in Computer Science strives to reduce, and possibly, close, this gap between a lower bound and the complexity of the best known algorithm. Sorting is one of the problems for which this goal has been achieved.

In what follows, we show how to derive a lower bound for the time complexity of sorting, but the technique is general enough to be applied also to other computational problems. We also restrict the discussion to sorting techniques based on comparisons between the elements, and all the sort algorithms mentioned so far belong to this class. There are, however, sorting methods like Radix sort or Counting sort, that place an element according to its own value rather than comparing it with another element, and the lower bound that we shall derive does not apply to them. For example, for comparison based methods, the input $\{4, 9, 7\}$ is equivalent to $\{40, 90, 70\}$ and even to $\{40, 90, 54\}$, as they all exhibit the same order: the smallest element first, then the largest, finally the middle one. The sequence of comparisons will thus be the same. For Radix sort. on the other hand, 4 is treated differently from 40.

Suppose then that a sequence of n different elements $a_1, a_2, \dots, a_n$ is given that should be arranged into increasing order. It will be convenient to reformulate this sorting task as finding the permutation $\{\pi(1), \pi(2), \dots, \pi(n)\}$ of the indices $\{1, 2, \dots, n\}$ for which

$$a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}.$$

Obviously, if the permutation is known, the actual sorting is just a matter of re-arranging the items in a linear scan.

To abbreviate the notation, we shall identify a given permutation of the elements $a_1, a_2, \dots, a_n$ by the ordered string of the indices of the terms. For example, if $n = 4$ and the order of a given permutation is $a_2 < a_4 < a_1 < a_3$, it will be represented by the string 2413.

A common pitfall for showing lower bounds is to claim that the task can "obviously" only be done is some specific manner, which requires a complexity of at least $\Omega(f)$ for some function f. This is generally wrong, as there might be infinitely many options to tackle the problem. The difficulty of showing a lower bound is thus that the proof has to encompass all the possible algorithms, even those that have not yet been invented! The proof must therefore refer generically to all these algorithms and cannot restrict the discussion just to some specific ones.

We shall therefore describe a sorting algorithm by the only information we have about it: that it consists of a sequence of comparisons. Indeed, most of the methods mentioned earlier can be specified by some comparison pattern. Writing $i\!:\!j$ as a shorthand for the comparison between $A[i]$ and $A[j]$, some of these sequences are as follows:

for Bubblesort – 1:2, 2:3, 3:4, . . . , $(n-1)$:n, 1:2, . . . ,

for Mergesort – 1:2, 3:4, 1:3, 5:6, 7:8, 5:7, 1:5, 9:10. . . . ,

for Quicksort (to be seen later) – 1:2, 1:3, 1:4, ...

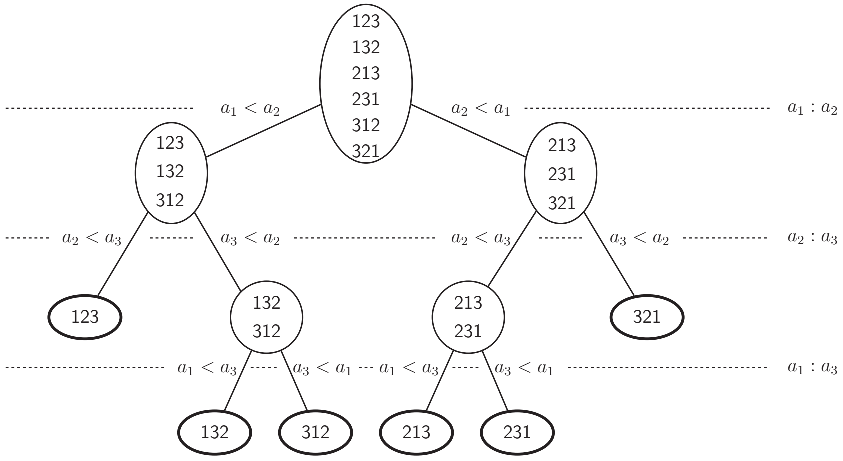

Figure 10.4. Example of a decision tree for sorting three elements.

However, not all sorting methods follow such well defined and regular comparison patterns, and most of the procedures do not even have names. In particular, the best methods for sorting a small set of items might be custom tailored to the size n of the set and not a particular case of any of the known general algorithms.

The tool by means of which the lower bound will be derived is called a *decision tree*. This is a binary tree corresponding to a specific sorting algorithm, that is, to a specific sequence of comparisons. Figure 10.4 is an example for a small decision tree, corresponding to the algorithm of sorting three elements a_1, a_2, a_3 using the sequence of comparisons 1:2, 2:3, 1:3. Each level of the tree corresponds to one of the comparisons, in order, and the nodes on level i of the tree contain the state of our knowledge after the first i comparisons. In particular, the root, on level 0, reflects the fact that there has not been any comparison yet, so we have not gained any information so far.

The knowledge is described as a set of permutations. The root contains all the $n!$ permutations of the numbers $\{1, \ldots, n\}$, $3! = 6$ in our example. If it is not a singleton, the set of permutations P_v in a node v on level $i - 1$ is split pursuant to the ith comparison into two nodes that will be the children of v on level i: if the comparison is $k : \ell$, one of these nodes will contain the permutations in P_v for which $a_k < a_\ell$, and the other node will contain the complementing set, those permutations of P_v for which $a_k > a_\ell$; the edges leading to these children nodes will be labeled accordingly. For example, let v be the left node on level 1 of the tree in Figure 10.4 that contains the permutations $P_v = \{123, 132, 312\}$; the second comparison is 2:3, hence P_v is split into $\{123\}$ for which $a_2 < a_3$, and {132, 312} for which $a_2 > a_3$, and these sets are stored in the left and right children of v on level 2, respectively.

The state of knowledge at a given node w on level i is a consequence of the *decisions* written as labels on the path from the root to w, which is why this is called a decision tree. For example, let w be the node on level 2 with $P_w = \{213, 231\}$. The decisions leading to w are $a_2 < a_1$ and $a_2 < a_3$, which are not enough to completely sort the three elements, because there are two permutations, 213 and 231 that are compatible with these outcomes of the first two comparisons. If the decisions had been $a_2 < a_1$ and $a_3 < a_2$, that would lead us to the node storing just one permutation 321, so in this case, the two first comparisons are sufficient to determine the order.

The aim of a sorting algorithm is to perform sufficiently many comparisons such that each leaf of the corresponding decision tree will contain just a single permutation. In other words, whatever the outcomes of these comparisons, we must be able to determine the exact order of the n elements. Note that the number of necessary comparisons is therefore the depth of the deepest leaf, which is the depth of the tree. As can be seen, for the example in Figure 10.4, three comparisons are sufficient: the tree has six leaves, which are emphasized, each containing a different single one of the six possible permutations. We also see that the first two comparisons alone are not enough: although there are some leaves on level 2, there are also nodes containing more than one permutation, so the sorting job is not completed.

How can this be generalized to a generic sorting algorithm? We obviously do not know which comparisons are used, and in which order. The answer is that we rely on certain properties of binary trees, of which decision trees are a special case. The number of different sorting orders of the given n elements is $n!$, and each of the corresponding permutations is stored as a singleton in one of the leaves of the decision tree, thus the number of leaves of this tree is $n!$.

As to the depth of the decision tree, let us again change our point of view, as we did, e.g., in the proof of Theorem 5.1 about AVL trees in Chapter 5. There is only one leaf in a tree of depth 0, which is a single node. There are at most two leaves in a binary tree of depth 1, and, by induction, there are at most 2^h leaves in a binary tree of depth h. The number of leaves of a binary tree will thus reach $n!$ only if h is large enough so that

$$2^h \geq n!,$$

from which we derive a lower bound on the depth:

$$h \geq \log_2 n!. \tag{10.1}$$

In particular, returning to our example, after 2 comparisons, the tree has at most 4 leaves. Since there are 6 permutations, there must be at least one leaf that contains more than a single permutation. This shows that not only this specific comparison order fails to sort the input of three elements in two steps, but that *any* sequence of two comparisons will equally be unsuccessful. To give a larger example, the number of permutations of 5 elements is $5! = 120$; after *any* 6 comparisons, the decision tree will only have $2^6 = 64$ leaves, so there is no way to sort 5 elements in 6 comparisons. Are 7 comparisons enough? It could be, since $2^7 = 128 > 120 = 5!$, but the technique of using the decision tree gives no clue about if it is indeed feasible and if so, about how to do it. Actually, it *is* possible to sort 5 elements in 7 steps, see the exercises.

Rewriting eq. (10.1) into a more convenient form, we get

$$\begin{aligned}\log(n!) &= \log(2 \cdot 3 \cdot 4 \cdots (n-1) \cdot n) < \log(n \cdot n \cdots n) \\ &= \log(n^n) = n \log n, \end{aligned} \tag{10.2}$$

which sets an upper bound. To derive also a lower bound, the symmetric technique does not succeed. In (10.2), we have replaced each factor by the maximum value, but if we replace it now by the minimum, we get that $n! > 2^n$, which is true but not useful. A better bound is achieved by proceeding in two steps:

$$\begin{aligned}\log(n!) &= \log(2 \cdot 3 \cdot 4 \cdots (n-1) \cdot n) \\ &> \log\left(\left(\frac{n}{2}+1\right) \cdot \left(\frac{n}{2}+2\right) \cdots (n-1) \cdot n\right) \\ &> \log\left(\left(\frac{n}{2}\right) \cdot \left(\frac{n}{2}\right) \cdots \left(\frac{n}{2}\right)\right) = \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2} \log \frac{n}{2},\end{aligned}$$

where the first inequality is obtained by dropping the first $\frac{n}{2}$ factors, and the second by replacing each factor by a lower bound on each of the remaining factors.

The conclusion is that the lower bound on the number of comparisons needed to sort n elements is in $\theta(n \log n)$. Since we have already seen algorithms that achieve this goal in $O(n \log n)$, these algorithms are *optimal*.

10.3 Lower Bound on the Average

The worst case behavior of an algorithm is surely an important criterion, since it provides a bound for all the cases, but for many problems, the worst case scenario is unrealistic. If so, it does not seem reasonable to compare competing methods considering only extreme situations that might hardly appear in practice, if at all. A better measure in such a context would then be the *average* performance of the algorithm, which depends on all its possible instances.

Background Concept: Average Complexity of an Algorithm

The *average* complexity of an algorithm will be defined as an extension of the notion of average or expectation of a random variable X. Let $\mathcal{V}$ be the (finite or infinite) set of possible values X can assume. The average or expectation of X is defined by

$$E(X) = \textstyle\sum_{v \in \mathcal{V}} v \text{ Prob}(X = v).$$

By analogy, given an algorithm $\mathcal{A}$, let $\mathcal{V}$ denote the set of all possible inputs of $\mathcal{A}$. The average complexity of $\mathcal{A}$ is defined by

$$E(\mathcal{A}) = \sum_{v \in \mathcal{V}} \begin{pmatrix} \text{number of steps executed} \\ \text{by } \mathcal{A} \text{ on input } v \end{pmatrix} \text{Prob} \begin{pmatrix} \text{the input} \\ \text{of } \mathcal{A} \text{ is } v \end{pmatrix}. \qquad (10.3)$$

For the evaluation of such an average, it is thus mandatory that the probability of occurrence of each of the possible inputs be given. The evaluation also requires the calculation of the number of steps in the execution of the algorithm on each of these inputs. Refer, for example, to the sorting problem, and consider again the decision tree of Figure 10.4. Each possible input corresponds to one of the permutations, which in turn are identified with the leaves of the tree. If $\mathcal{L}$ denotes the set of leaves of the decision tree, eq. (10.3) can be rewritten as

$$E(\mathcal{A}) = \sum_{\ell \in \mathcal{L}} \text{depth}(\ell) \ \text{ Prob} \begin{pmatrix} \text{the input of } \mathcal{A} \text{ is ordered} \\ \text{by the permutation in } \ell \end{pmatrix}. \qquad (10.4)$$

For example, if the probabilities of the permutations in the leaves of Figure 10.4 are, from left to right, 0.1, 0.4, 0.2, 0.1, 0.05, and 0.15, the corresponding average is

$$0.1 \times 2 + 0.4 \times 3 + 0.2 \times 3 + 0.1 \times 3 + 0.05 \times 3 + 0.15 \times 2 = 2.85.$$

Lacking any other information, one often assumes that all the possible inputs are equally likely. This is not a part of the definition of an average, and it has to be stated explicitly if such uniform distribution is assumed. For the sorting problem, it would mean that each permutation appears with probability $\frac{1}{n!}$. For the example of Figure 10.4, assuming equal probabilities would yield an average of

$$\frac{1}{6}(2+3+3+3+3+2) = \frac{16}{6} = 2.67.$$

To derive a lower bound on the average complexity of any sorting algorithm, we again refer to the decision tree as a model for comparison based sorting methods. We also assume that all the $n!$ possible input permutations appear with equal probability $\frac{1}{n!}$. Let $\mathcal{A}$ be such a sorting algorithm, let $\mathcal{T}$ be the corresponding decision tree, and denote by $\mathcal{L}$ the set of the leaves of $\mathcal{T}$. We then get from eq. (10.4) that according to our assumptions

$$E(\mathcal{A}) = \frac{1}{n!}\sum_{\ell\in\mathcal{L}} \text{depth}(\ell). \tag{10.5}$$

We are confronted with the same problem as in the worst case analysis of the previous section: the right-hand side of eq. (10.5) includes the sum of the depths of all the leaves of the decision tree, but the shape of this tree varies with the different sequences of comparisons. The solution, as before, is to consider properties that are common to all binary trees, and thus in particular to decision trees.

Let T be a complete binary tree with m leaves, that is, all internal nodes of T have two children. Define

$$D(T) = \sum_{\ell\in\{\text{leaves of } T\}} \text{depth}(\ell)$$

as the sum of the depths of all the leaves of T, which is a function depending on the shape of the given tree. To get a function which is independent of the shape, and only relies on the number of leaves, we further define

$$d(m) = \min\{D(T) \mid T \text{ has } m \text{ leaves}\}.$$

For example, the tree T in Figure 10.5(d) has $D(T) = 1+2+3+3 = 9$, and that in Figure 10.5(e) $2+2+2+2 = 8$. Since these are the only possible shapes for complete binary trees with 4 leaves (there are symmetric shapes, with the same depths), we get that $d(4) = \min\{8, 9\} = 8$.

To study the general behavior of the function d, we start with the first values. The trees in Figure 10.5 appear with the number m of their leaves above them, and with the depths written in the leaves. For $m = 1, 2, 3$, there is only one option for a complete binary tree (again, up to symmetry) and the trees appear in Figures 10.5(a), (b), and (c), respectively. The corresponding values of D, and

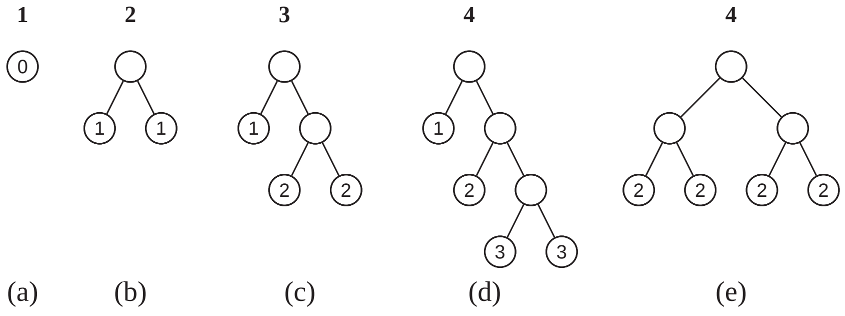

Figure 10.5. Complete binary trees with one to four leaves.

thus also of d, are 0, 2, and 5. We saw already that $d(4) = 8$, but for larger m, there are clearly many options of tree shapes (see the discussion in Section 4.2 on the number of trees).

Let us derive a recursive formula for D. Consider the tree T_m in Figure 10.6, where the subscript refers now to the number of leaves in the tree. Suppose the left subtree of T_m has i leaves, for some $1 \le i < m$, and let us therefore denote the tree by T_i; the right subtree has then $m - i$ leaves and will be denoted T_{m-i} accordingly. To express $D(T_m)$ as a function of $D(T_i)$ and $D(T_{m-i})$, note that for each leaf, its depth in the subtree is one less than its depth in the tree itself. For example, in Figure 10.6, the depth in T_i of the leaf appearing as a small white circle is the number of edges in the broken line showing the path from the root of T_i, indicated by a black dot, to this leaf; if the path starts at the root of T_m, just one edge is added, so the depth of the leaf is increased by 1. Since this is true for each of the m leaves, we get

$$D(T_m) = (D(T_i) + i) + (D(T_{m-i}) + m - i) = D(T_i) + D(T_{m-i}) + m. \quad (10.6)$$

The formula can be verified for each of the trees with $m > 1$ of Figure 10.5.

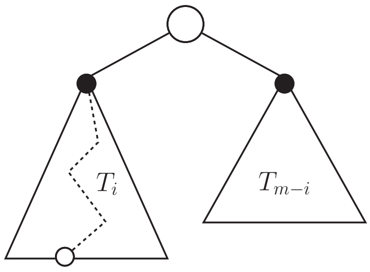

Figure 10.6. Recursive evaluation of the sum of the depths of the leaves.

We would like to infer from (10.6) a formula for d. Since the passage from D to d is by applying a minimum function, a first thought could lead to

$$d(m) = d(i) + d(m-i) + m, \tag{10.7}$$

but this formula does not make any sense: the left-hand side depends only on m, whereas the right-hand side has an additional parameter i. The reason is that the i in eq. (10.6) is not really a parameter – it is the number of leaves in the left subtree and therefore induced by T_m. Formally, we should have written $i(T_m)$ instead of just i to make this dependency explicit. It turns out that the correct form of the attempt in (10.7) is rather

Claim 10.1. $$d(m) = \min_{1\le i<m} \big(d(i) + d(m-i) + m\big). \tag{10.8}$$

Equation (10.8) may be mathematically sound as the free parameter i is now bound by the additional minimum, acting like a quantifier, but the correctness of the expression for $d(m)$ is surely not self-evident and needs to be shown.

Proof We show that there are inequalities in both directions. Let i be an index between 1 and $m-1$. A tree T with m leaves for which $D(T) = d(m)$ will be called a *minimal* tree. Let $\hat{T}_i$ and $\hat{T}_{m-i}$ be minimal trees with i and $m-i$ leaves, respectively, and define $\hat{T}_m$ as the binary tree which has $\hat{T}_i$ as its left and $\hat{T}_{m-i}$ as its right subtrees. The tree $\hat{T}_m$ is not necessarily minimal, but $d(m)$ cannot be larger than $D(\hat{T}_m)$, which yields

$$d(m) \le D(\hat{T}_m) = D(\hat{T}_i) + D(\hat{T}_{m-i}) + m = d(i) + d(m-i) + m$$

for the specific i we have chosen. But since this is true for any i in the given range, it follows that

$$d(m) \le \min_{1\le i<m} \big(d(i) + d(m-i) + m\big).$$

The proof for the complementing inequality is not symmetric. Let $\tilde{T}_m$ be a minimal tree with m leaves. Denote the number of leaves of the left subtree of $\tilde{T}_m$ by j and use the notation $\tilde{T}_j$ and $\tilde{T}_{m-j}$ for the left and right subtrees of $\tilde{T}_m$. The subtrees are not necessarily minimal. We therefore have

$$d(m) = D(\tilde{T}_m) = D(\tilde{T}_j) + D(\tilde{T}_{m-j}) + m \ge d(j) + d(m-j) + m,$$

again just for the specific j mentioned. This is not true for all indices j, but since there exists a j for which it is true, it follows that

$$d(m) \ge \min_{1\le i<m} (d(i) + d(m-i) + m),$$

and hence there must be equality. ∎

The formula can now be used to discover new values of $d(m)$, for example,

$$d(5) = \min \left\{ \begin{array}{l} d(1) + d(4) + 5 = 0 + 8 + 5 = 13 \\ d(2) + d(3) + 5 = 2 + 5 + 5 = 12 \end{array} \right\} = 12$$

$$d(6) = \min \left\{ \begin{array}{l} d(1) + d(5) + 6 = 0 + 12 + 6 = 18 \\ d(2) + d(4) + 6 = 2 + 8 + 6 = 16 \\ d(3) + d(3) + 6 = 5 + 5 + 6 = 16 \end{array} \right\} = 16,$$

but it would be more convenient to get a closed formula for $d(m)$, and not a recursive one. The following bound can be shown:

Claim 10.2. $$d(m) \geq m \log_2 m.$$

Proof by induction on m. For $m = 1$, indeed

$$d(1) \geq 1 \log_2 1 = 0.$$

Suppose then that the claim is true for $1 \leq i < m$, and let us show it also for $i = m$:

$$\begin{aligned} d(m) &= \min_{1 \leq i < m} \big(d(i) + d(m-i) + m \big) \\ &\geq \min_{1 \leq i < m} \big(i \log_2 i + (m-i) \log_2 (m-i) + m \big). \end{aligned}$$

To find the minimum, define the function $f(x) = x \log_2 x + (m-x) \log_2 (m-x) + m$ on all the reals $1 \leq x \leq m$. The function $f(x)$ is differentiable and its derivative is

$$f'(x) = \frac{1}{\ln 2} \left(\frac{x}{x} + \ln x - \frac{m-x}{m-x} - \ln(m-x) \right).$$

Setting this to zero implies $\ln x = \ln(m-x)$ and thus $x = m - x$, that is, $x = \frac{m}{2}$. The second derivative of $f(x)$ at $x = \frac{m}{2}$ is $f''(\frac{m}{2}) = \frac{4}{m \ln 2} > 0$, so this is a minimum. Plugging the index at which the minimum is reached into the formula, one gets

$$d(m) \geq \left(\frac{m}{2} \log_2 \frac{m}{2} + \frac{m}{2} \log_2 \frac{m}{2} + m \right) = m \log_2 \frac{m}{2} + m = m \log_2 m. \quad \blacksquare$$

We can now return to the problem of finding a lower bound to the average complexity of a comparison based sorting algorithm. The quantity we wish to evaluate is $\frac{1}{n!} D(\mathcal{T})$, where $\mathcal{T}$ is the decision tree of a given sorting algorithm. The difficulty was that $D(\mathcal{T})$ depends on the shape of the tree, which is unknown, so the evaluation has to be based on the only information we have about the decision tree, namely, the number of its leaves, $n!$. The conclusion is

$$\begin{array}{c}\text{average number of}\\ \text{comparisons for sorting}\end{array} = \frac{1}{n!}D(\mathcal{T}) \geq \frac{1}{n!}d(n!) \geq \frac{1}{n!}n!\log_2 n!$$
$$= \log_2 n! = \theta(n \log n).$$

Interestingly, the lower bound on the average case is the same as for the worst case! It also implies that the average case complexity for methods that are optimal from the worst case point of view, like Mergesort or Heapsort, must also be $\theta(n \log n)$.

10.4 Quicksort

This section presents yet another example of the Divide and Conquer family of algorithms, a sorting technique known by its somewhat pretentious name Quicksort, especially in view of its very slow worst case performance. The method and some of its variants are nonetheless important enough to deserve their own section.

Given is an array $A[1] \cdots A[n]$ of n numbers, and we shall again assume that they are all different, to facilitate the discussion. Quicksort starts by picking one of the elements, K, and using it to partition the array into three subsets, which can be performed in a single linear scan of the set:

$S_1 = \{x \in A \mid x < K\}$	the elements of A that are smaller than K,
$\{K\}$	the singleton containing only K,
$S_2 = \{x \in A \mid x > K\}$	the elements of A that are larger than K.

The elements are then rearranged in-place in the original array, with S_1 on the left and S_2 on the right, as depicted schematically in Figure 10.7. The sets S_1 and S_2 are not sorted themselves, but K is already at its intended location in the final sorting order. All that remains to be done is thus to sort each of the subsets S_1 and S_2, which can be done recursively. The formal algorithm, in which $A[x : y]$ denotes the elements of A with indices between x and y, inclusive, is given in Figure 10.8.

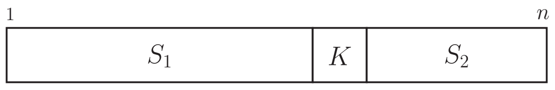

Figure 10.7. Partition of the array into three subsets.

```
Quicksort($A$)
    if $|A| > 1$ then
        $j \longleftarrow$ Partition($A$)
        $S_1 \longleftarrow A[1 : j-1]$        Quicksort($S_1$)
        $S_2 \longleftarrow A[j+1 : n]$        Quicksort($S_2$)
```

Figure 10.8. Quicksort.

Quicksort does not specify which element K should be used for the partition step, and therefore programmers often pick the easiest choice, which is to use the first element $A[1]$ of the input array. The partition can then be done by initializing two pointers i and j, the former to the beginning and the latter to the end of the remaining array. We wish to have the elements of S_1 toward the left end of the array, and those of S_2 at the right end. This can be achieved by comparing the element i points to, with K. If it is smaller, it is already in the proper area, so we can increment i and check the next element, until an element that is larger than K is found. At this point, we start a similar process from the end, pointed to by j. If $A[j]$ is larger than K, we can step backward, until an element is found that is smaller. All one has to do to rectify the order is then to swap the elements $A[i]$ and $A[j]$, and to repeat the whole process until the pointers meet. Figure 10.9 is a schematic of one iteration of this process, where the $<$ and $>$ signs indicate whether the element is smaller or larger than K; the next iteration will start at the positions indicated by the broken line arrows.

At the end of this process, the indices i and j will have switched places, as can be seen in Figure 10.10(a), which corresponds to the general case. Figure 10.10(b) is the special case in which K is the largest element, so to stop the loop on i, it is convenient to use the first element *after* the array, $A[n+1]$, as a sentinel element, a technique we have seen in Chapter 2. The other extreme case, when K is the smallest element, is shown in Figure 10.10(c). Now it is the loop of j that processes the entire array, but here, no sentinel element is needed, since $A[1]$ contains K, which will stop the iteration. Thus in all three cases, at the end of the main loop, we have $j = i - 1$, and the number of comparisons

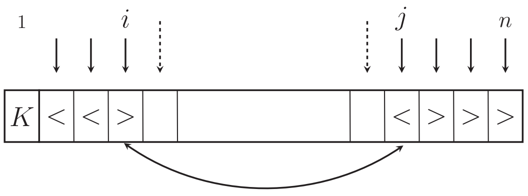

Figure 10.9. Schematic of the partition step.

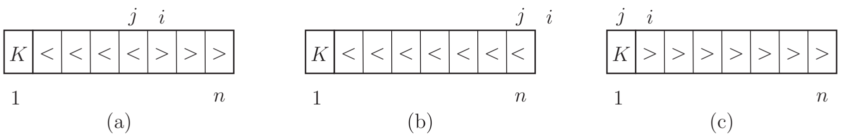

Figure 10.10. At the end of the partition step.

performed is $n+1$: each of the $n-1$ elements $A[2]\cdots A[n]$ is compared once, plus an additional comparison for each of the two elements pointed to by i and j at the end of the main loop. The index j points then to the rightmost of the elements that are smaller than K, or to K itself, if no such smaller elements exist; to finalize the partition into the form depicted in Figure 10.7, what remains to be done is to swap the elements K and $A[j]$. The partition algorithm returns the index j of the partition element K at the end of this step, and is given in Figure 10.11.

10.4.1 Worst Case

The worst case will occur when the partition element is repeatedly one of the extreme elements of the array. In that case, one of the sets S_1 or S_2 is empty and there is only one recursive call, but it is with a set of size $n-1$. If $T(n)$ denotes the worst case number of comparisons used by Quicksort on an input of size n, one has

$$T(n) = n + T(n-1) = \sum_{i=1}^{n} i = \theta(n^2).$$

It is quite embarrassing that for the earlier choice of the partition element, the worst case is attained, among others, for an input which is already sorted, whereas in this case one would expect only very little work to be done. In addition, such an input is not unrealistic, as the input of a sorting procedure is often the output of some preprocessing that might already have produced some

```
Partition($A$)
    $K \longleftarrow A[1]$
    $A[n+1] \longleftarrow \infty$          // sentinel element
    $i \longleftarrow 2$        $j \longleftarrow n$
    while $i < j$ do
        while $A[i] < K$ do $i$++
        while $A[j] > K$ do $j$--
        if $i < j$ swap($A[i], A[j]$)
        $i$++        $j$--
    swap($A[1], A[j]$)
    return $j$
```

Figure 10.11. Partition step used in Quicksort.

order. To remedy this behavior, some heuristics suggest choosing the partition element differently, as we shall see in the following section, but the worst case will still be $\theta(n^2)$.

10.4.2 Average Case

For the average case, let us redefine $T(n)$ to stand for the average number of comparisons required by Quicksort to sort n elements, assuming a uniform distribution on the input permutations. As boundary condition, we have $T(1) = 0$. The number of comparisons in the partition part is $n + 1$, independently of the choice of K. The recursive steps, however, are influenced by K, since its relative position, as depicted in Figure 10.7, determines the sizes of the subsets S_1 and S_2. Assuming a uniform distribution implies that in each iteration, K has an equal chance to end up in one of the possible positions. For the first iteration this means that the probability of K being in position i, for $1 \le i \le n$, is $\frac{1}{n}$; the corresponding sizes of S_1 and S_2 are $i - 1$ and $n - i$, respectively. This can be summarized by the formula

$$T(n) = n + 1 + \frac{1}{n}\sum_{i=1}^{n}\big(T(i-1) + T(n-i)\big) = n + 1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i). \tag{10.9}$$

To solve this recursion, it will be helpful to get rid of the fractional part, so that only integers are manipulated. Multiplying both sides of (10.9) by n, one gets

$$nT(n) = n(n+1) + 2\sum_{i=0}^{n-1} T(i). \tag{10.10}$$

We handled recursions already, but this one is different in that $T(n)$ does not just depend on $T(n-1)$ or $T\left(\frac{n}{2}\right)$, but rather on all of its previous values. This should remind us the summation technique studied in Section 9.4.1: introduce an additional equation, which is similar to, but different from, eq. (10.10). Since the formula is true for every n, let us rewrite it for $n - 1$:

$$(n-1)T(n-1) = (n-1)n + 2\sum_{i=0}^{n-2} T(i). \tag{10.11}$$

Subtracting (10.11) from (10.10) then yields

$$nT(n) - (n-1)T(n-1) = 2n + 2T(n-1), \tag{10.12}$$

in which most elements of the summation have been canceled. Rearranging the terms, we get

$$nT(n) = 2n + (n+1)T(n-1), \tag{10.13}$$

in which the $(n + 1)$ is annoying, because if we had $(n - 1)$ instead, we could have defined $G(n) = nT(n)$ and rewrite (10.13) as $G(n) = 2n + G(n-1)$, which is already easy to solve. To still enable a substitution, we divide the two sides of (10.13) by both n and $n+1$:

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{T(n-1)}{n}, \tag{10.14}$$

suggesting to define $G(n) = \frac{T(n)}{n+1}$, which transforms (10.14) into

$$\begin{aligned} G(n) &= \frac{2}{n+1} + G(n-1) \\ &= \frac{2}{n+1} + \frac{2}{n} + G(n-2) \\ &\vdots \\ &= \frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{k+2} + G(k) \qquad \text{for all } k < n. \end{aligned}$$

The boundary condition for T implies $G(1) = 0$, so we get

$$G(n) = \sum_{i=3}^{n+1} \frac{1}{i} \simeq \ln n,$$

as this is the harmonic sum we have seen in the analysis of uniform hashing in Section 9.4. Returning to the function T, the conclusion is

$$T(n) = (n+1)G(n) \in \theta(n \log n).$$

Incidentally, the average case complexity of Quicksort can be used to prove the divergence of the harmonic series: if it converges, then $G(n)$ would be bounded by a constant, implying that $T(n) \in O(n)$, but this contradicts the lower bound on the average for sorting.

In spite of its bad worst case, the average complexity of Quicksort is the best one could expect, which is why the method is still popular. Another reason for the importance of Quicksort is that it is the basis for the linear time select algorithm to be studied in the following section.

10.5 Finding the *k*th Largest Element

The average of a set of numbers is a widely used statistical tool, but it should not always be applied, because it may be biased by extreme values. For instance, if a billionaire joins a group of 99 people who earn very little, the average wealth in the group may be more than 10 million! A more adequate measure in this and many other cases would be the median, rather than the average. The *median* of a set of different numbers is the element m that splits the set into equal halves, that is, (about) half of the elements are smaller, and half of the elements are larger than m. The average is better known, probably because it is easier to evaluate.

A first attempt for finding the median of a set could be based on sorting the set beforehand; the median is then stored in the middle element of the sorted array, so there are $O(n \log n)$ algorithms for finding the median. But are $\Omega(n \log n)$ comparisons really necessary? Sorting seems to be an overkill, since it would yield not just the median, but also any other percentile, so maybe if we are willing to settle for getting only the median, the complexity could be lowered?

This leads to the idea of using just a *partial* sort, though there are many possible interpretations for this partiality. For Bubblesort and Heapsort, for instance, the last k elements are already in order after k iterations. This means that to find the median, one could exit the sorting procedure after $\frac{n}{2}$ iterations. But for Bubblesort, this would still require $\sum_{i=n/2}^{n} i = \Omega(n^2)$ comparisons, and for Heapsort, $n + \sum_{i=n/2}^{n} \log i = \Omega(n \log n)$, so not much has been gained.

Quicksort seems to be a better choice in this context, because the partition step in itself is already a kind of partial sort. Refer to Figure 10.7 describing the order of the elements after the partition step. To complete the sorting, both S_1 and S_2 need to be sorted, but for finding the median, it is sufficient to sort only one of the sets, the one containing the element indexed $\frac{n}{2}$. This means that one of the recursive calls in the algorithm in Figure 10.8 could be saved in each iteration.

A new problem, however, arises. A recursive program is based on the assumption that the subproblem to be solved by a recursive call is identical to the original problem, just applied to a smaller input. For example, the first action of Mergesort is to sort, recursively, half of the original input array. But this assumption does not hold for the present problem. In our attempt to adapt Quicksort to find the median, the recursive call will be either with S_1 or S_2, unless, of course, their sizes are equal in which case the median is K. Yet it is not the *median* of S_1 or S_2 we are looking for, but the element in position $\frac{n}{2}$. An example will illustrate this difficulty. Suppose the set A in Figure 10.7 is of size 101, and that $|S_1|$, the size of S_1, is 69, so that K is stored at index 70. The median of A is the element that should be at position 50, and therefore belongs to S_1, but this is not the median of S_1; the latter should be at index 35, not at index 50.

This is one of the cases in which a more general, and thus supposedly, more difficult, problem will actually turn out to be easier to solve. Instead of writing an algorithm that returns the median of a set, we generalize the problem by adding an additional parameter, and look for an algorithm Select(A, k) which, given a set A of size n and an index k, with $1 \leq k \leq n$, returns the kth largest

```
Select1$(A, k)$
    $j \longleftarrow$ Partition$(A)$
    if $k = j$ then return $A[j]$
    else
        $S_1 \longleftarrow A[1 : j - 1]$
        $S_2 \longleftarrow A[j + 1 : n]$
        if $k < j$ then      Select1$(S_1, k)$
        else   // here $k > j$
                             Select1$(S_2, k - j)$
```

Figure 10.12. Select1(A, k) – find the kth largest element of A, first version.

element of A. Clearly,

$$\text{Median}(A) = \text{Select}\left(A, \frac{n}{2}\right),$$

so an algorithm for Select will in particular also solve the median problem, and in fact also any percentile.

Why is it easier to solve the more general problem? Because the addition of a parameter gives us the flexibility to use different parameters in different calls, which enables the use of recursion where it could not be applied before.

To find the kth largest element, we start with a partition step just as for Quicksort. Suppose the partition element was placed at index j. Then if $k = j$, we were very lucky: the partition element, that has been chosen independently of the input parameter k, just happened to be exactly the sought element! If k is smaller than j, then the element must be in S_1, and it is the kth largest element of S_1. The third case is when $k > j$; then the element is in S_2, but it is not the kth largest there. We know that there are j elements in A that are smaller than any element in S_2: all the elements of S_1 and the partition element K. Therefore the kth largest element of A will be the $(k - j)$th largest element of S_2. This is summarized in the algorithm of Figure 10.12, which assumes $|A| > 1$ and $1 \le k \le |A| = n$. We call it Select1 because this is only a first sketch that will be improved later.

The Select1 algorithm of Figure 10.12 might look better than Quicksort of Figure 10.8, because there is only a single recursive call, but remember that in the worst case of Quicksort, when the partition element fell always at the left or right end of the array, there was also only one recursive call. In fact, the worst case remains the same for Select1 as for Quicksort and will take time $\Omega(n^2)$.

It is the fact that there are no restrictions on the choice of the partition element which is to blame for this bad performance. If we could force it to fall always in the middle of the array, the sizes of S_1 and S_2 would be at most $\frac{n}{2}$, and the worst case complexity $T(n)$ of Select1 would satisfy the recurrence

$$T(n) = n + T\left(\frac{n}{2}\right) = n + \frac{n}{2} + \frac{n}{4} + \cdots = O(n). \tag{10.15}$$

But this means that we would like to choose K as the median of the set, which is exactly the problem we started with in the first place! We circumvent this complication by relaxing the requirement on K to be not necessarily *equal* to the median, but just *close to* it, where the exact meaning of this closeness will be defined later.

Instead of choosing K simply as $A[1]$, the first element of the array, in the Partition procedure, this command will be replaced in the improved version called Select by the following curious steps:

(i) Choose a small, odd integer b as parameter – its value will be fixed after the analysis.
(ii) Consider the array A as a sequence of $\frac{n}{b}$ disjoint subsets of size b: $A[1:b], A[b+1:2b], \ldots$
(iii) Sort each of the b-tuples.
(iv) Define C as the set of the medians of the sorted b-tuples.
(v) Define K as the median of the set C of medians.

The parameter b should be small, so that the time for sorting a b-tuple may be considered as constant; it should be odd to facilitate the extraction of the median of each b-tuple. The size of the set C is $\frac{n}{b}$, thus the command in step 5 can be performed by a recursive call:

$$K \longleftarrow \text{Select}\Big(C, \frac{n}{2b}\Big).$$

To understand why this strange sequence of commands is helpful, consider the schematic of the n elements into a $b \times \frac{n}{b}$ matrix shown in Figure 10.13. Each column represents one of the b-tuples, which we assume here already sorted in increasing order bottom to top. The set C of the medians are the darker circles. We further assume that the columns are arranged so that the column containing K, the median of the medians which appears here in black, is in the center, and that the columns in the left part have their gray median elements smaller than K, and the columns in the right part being those with their gray median elements larger than K.

Pick one of the (white) elements in the lower left part of the matrix, that is, within the area surrounded by the solid black polygon. The element is smaller than the (gray) median above it, belonging to the same column, which in turn is smaller than the (black) partition element K. Since this is true for all the elements in this area, we know that all these elements are smaller than K, and thus belong to S_1. It follows that there are at least $\frac{n}{4}$ elements in S_1, so that S_2 cannot be larger than $\frac{3}{4}n$. A symmetric argument shows that all the elements in the upper right quarter, surrounded by the broken line, are larger than K and thus in S_2. Therefore $|S_2| \geq \frac{n}{4}$, implying that $|S_1| \leq \frac{3}{4}n$.

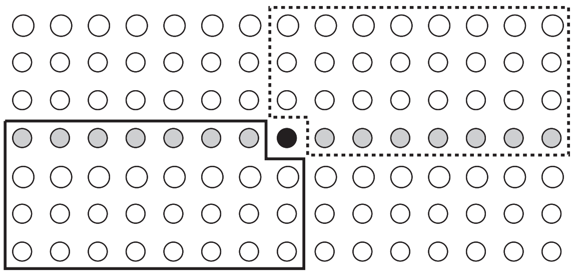
Figure 10.13. Schematic of the set A as a matrix of size $b \times \frac{n}{b}$.

We now understand that the ultimate goal of the complicated choice procedure for the partition element was to distance it from the extremities, and to force it to be close to the middle in the sense that its index should be in the range $\left[\frac{n}{4}, \frac{3n}{4}\right]$. This yields an upper bound on the size of the input set in the recursive call of at most $\frac{3}{4}$ of the size in the calling routine. Figure 10.14 summarizes the Select algorithm.

Line 1 deals with the boundary condition to exit the recursion. It obviously makes not much sense to apply the partition into b-tuples on very small input sets, and the size of 50 elements has been set empirically and does not influence the order of magnitude of the complexity. Smaller sets are just sorted by any method, and the kth largest element can then be found by direct access. Lines 3–5 are the definition of the partition element according to the steps we saw earlier, and the rest of the algorithm is just adapted from Select1.

Denote the number of comparisons made by Select on an input of size n by $T(n)$. Remark that though Select has also another parameter k, we seek a

```
Select($A, k$)
1   if $|A| < 50$ then sort($A$) and return $k$-th element
2   else
3       sort $b$-tuples $A[rb+1 : (r+1)b]$ for $0 \le r < \frac{n}{b}$
4       $C \longleftarrow$ medians of $b$-tuples
5       $K \longleftarrow$ Select$\left(C, \frac{n}{2b}\right)$
6       $S_1 \longleftarrow \{x \in A \mid x < K\}$
7       $S_2 \longleftarrow \{x \in A \mid x > K\}$

8       if $k = |S_1| + 1$ then return $K$
9       else   if $k < |S_1| + 1$ then      Select($S_1, k$)
10             else                         Select($S_2, k - |S_1| - 1$)
```

Figure 10.14. Select(A, k) – find the kth largest element of A, with better worst case.

bound on $T(n)$ that is independent of k. If $n < 50$, then $T(n) < n \log_2 n < 6n$, according to line 1. Otherwise the following apply:

(i) Each b-tuple in line 3 can be sorted in $O(b^2)$ steps and there are $\frac{n}{b}$ such tuples, so the entire step requires at most bn comparisons.
(ii) There are no comparisons in line 4.
(iii) Line 5 is a recursive call with a set C of size $\frac{n}{b}$, which can be done in time $T\left(\frac{n}{b}\right)$.
(iv) Lines 6-7 are the partition, performed by a linear scan, that is, in $O(n)$.
(v) In the worst case, there is one more recursive call, either with S_1 or with S_2; in any case, the time will be bounded by $T\left(\frac{3n}{4}\right)$.

Collecting all the terms, we get

$$T(n) \le \begin{cases} 6n & \text{if } n < 50 \\ T\left(\frac{n}{b}\right) + T\left(\frac{3n}{4}\right) + cn, & \text{if } n \ge 50 \end{cases} \tag{10.16}$$

for some constant $c > 1$. This recurrence is a bit different from those we have seen so far.

Background Concept: Recurrence Relations

Many programs are based on recursive algorithms and, consequently, the analysis of these algorithms contains recurrence relations, of which (10.16) is a typical example. The general solution of such relations is a subject for an entire course and is beyond the scope of this book. We shall just give here some useful rules of thumb.

If the function T we wish to evaluate recurs only once on the right-hand side, like, for example, in (10.15), we might be able to get a closed formula for $T(n)$ by repeatedly substituting the formula until a boundary condition is reached. But if there are multiple occurrences of T on the right-hand side, as in (10.16), substitution may not help, and the formula can get longer and increasingly complicated. It then often helps to guess the order of magnitude of $T(n)$, and to try to prove it by induction on n. The problem is, of course, to know what to guess.

For recursion to work properly, the recursive call must be with an input that is strictly smaller, in some sense, than the original one. Usually it is with an input set that is a strict subset of the input dealt with by the calling function. Suppose then that the recurrence relation for the given algorithm is of the form

$$T(n) \le T(\alpha_1 n) + T(\alpha_2 n) + \cdots + T(\alpha_s n) + cn,$$

where all α_i are smaller than 1 and c is some constant. Then

(i) if $\alpha_1 + \alpha_2 + \cdots + \alpha_s < 1$, then one may guess that $T(n) \in O(n)$;
(ii) if $\alpha_1 + \alpha_2 + \cdots + \alpha_s = 1$, then one may guess that $T(n) \in \theta(n \log n)$;
(iii) if $\alpha_1 + \alpha_2 + \cdots + \alpha_s > 1$, then one may guess that $T(n) \in \Omega(n^d)$, for some constant $d > 1$.

No proof will be given here as this is just a suggestion for an educated guess. In any case, the actual formula has to be shown by induction.

Before solving the recurrence (10.16), recall that we still have the freedom of choosing the parameter b. The formula in (10.16) corresponds to $\alpha_1 = \frac{1}{b}$ and $\alpha_2 = \frac{3}{4}$. To get a linear behavior, we would like to have $\alpha_1 + \alpha_2 < 1$, thus $b = 3$ is not adequate. The smallest value for b is therefore $b = 5$. This means that we use quintuples in the first partition, each of which can be sorted in 7 comparisons as mentioned earlier. The recurrence becomes

$$T(n) \le \begin{cases} 6n & \text{if } n < 50 \\ T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) + cn, & \text{if } n \ge 50. \end{cases} \tag{10.17}$$

Claim 10.3. $$T(n) \le 20\,c\,n.$$

Proof by induction on n. The inequality holds trivially for $n < 50$. Suppose then that it is true for all values up to $n - 1$, and let us show it for n, where $n \ge 50$:

$$\begin{aligned} T(n) &\le T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) + cn \\ &\le 4\,c\,n + 15\,c\,n + c\,n = 20\,c\,n, \end{aligned}$$

where we have used the inductive assumption on $\frac{n}{5}$ and on $\frac{3n}{4}$, which are both strictly smaller than n. This may sound trivial, but it is not: the size of a set has to be rounded to be an integer, and, for instance for $n = 2$, $\lceil \frac{3n}{4} \rceil = \lceil 1.5 \rceil = 2$, so it is *not* smaller than n. This is an additional reason for choosing the boundary condition not at $n = 1$ or 2, but high enough so that $\lceil \frac{3n}{4} \rceil < n$. ∎

We conclude that the kth largest element of an unordered set in general, and its median in particular, can be found in time that is linear in the size of the set, thus without sorting it. This fact has a multitude of applications, and has been used to improve the complexity of many algorithms.

Exercises

10.1 Show how to sort 5 elements in 7 comparisons.

10.2 You are given 12 balls that seem to be identical, but the weight of one of the balls is different. The problem is to find this ball and to decide if it is lighter or heavier by means of a two sided weighing scale.

(a) Give a lower bound ℓ on the number of necessary weighings.
(b) Show how to solve the problem in ℓ weighings.
(c) Change the number of balls to 13 and derive the corresponding lower bound.
(d) Show that the problem with 13 balls cannot be solved in 3 weighings.

10.3 Give a sorting algorithm and a probability distribution of the possible input permutations for which the average time complexity is $O(n)$.

10.4 Assuming a uniform distribution, the average time of any sorting algorithm has been shown in eq. (10.5) to be $\frac{1}{n!}D(\mathcal{T})$, where $\mathcal{T}$ is the corresponding decision tree. For the particular case of Quicksort, we saw another formula in eq. (10.9), which had only n in the denominator, not $n!$. Explain this presumed discrepancy.

10.5 You are given an unsorted array A of n numbers. Show that the task of building a search tree for the n elements of A requires at least $\Omega(n \log n)$ comparisons.

10.6 An additional possibility for a definition of a partial sort is to partition a given set of n different numbers into K subsets $S_1, \ldots, S_K$, each of size $\frac{n}{K}$, where K is an integer $K \geq 2$, such that each element in S_i is smaller than any element in S_{i+1}, for $1 \leq i < K$. The sets S_i themselves are not sorted. Show how to perform this partial sort in time $O(n \log K)$. **Hint:** Start with assuming that K is a power of 2, then show how to generalize to any K.

10.7 Instead of using the median of the medians of 5-tuples as partition element K in Select, could we not simply choose K as the median of some subset of A, for example, by

$$K \longleftarrow \text{Select}\big(A[1 : \tfrac{n}{2}], \tfrac{n}{4}\big),$$

as this would also imply an upper bound of $\frac{3}{4}n$ on the sizes of both S_1 and S_2? So what is wrong with this choice, which seems to be much easier to implement?