shunned: the hidden constant associated with this "improved" version of quicksort is so large that it results in an algorithm worse than heapsort in every case!

7.5 Finding the median

Let $T[1..n]$ be an array of integers and let s be an integer between 1 and n. The *s–th smallest element* of T is defined as the element that would be in the s–th position if T were sorted into nondecreasing order. Given T and s, the problem of finding the s–th smallest element of T is known as the *selection* problem. In particular, the *median* of $T[1..n]$ is defined as its $\lceil n/2 \rceil$–th smallest element. When n is odd and the elements of T are distinct, the median is simply that element in T such that there are as many items in T smaller than it as there are items larger than it. For instance, the median of $[3,1,4,1,5,9,2,6,5]$ is 4 since 3, 1, 1 and 2 are smaller than 4 whereas 5, 9, 6 and 5 are larger.

What could be easier than to find the smallest element of T or to calculate the mean of all the elements? However, it is not obvious that the median can be found so easily. The naive algorithm for determining the median of $T[1..n]$ consists of sorting the array and then extracting its $\lceil n/2 \rceil$–th entry. If we use *heapsort* or *mergesort*, this takes a time in $\Theta(n \log n)$. Can we do better? To answer this question, we study the interrelation between finding the median and selecting the s–th smallest element.

It is obvious that any algorithm for the selection problem can be used to find the median: simply select the $\lceil n/2 \rceil$–th smallest. Interestingly, the converse holds as well. Assume for now the availability of an algorithm *median*$(T[1..n])$ that returns the median of T. Given an array T and an integer s, how could this algorithm be used to determine the s–th smallest element of T? Let p be the median of T. Now pivot T around p, much as for *quicksort*, but using the *pivotbis* algorithm introduced at the end of the previous section. Recall that a call on *pivotbis*$(T[i..j], p;$ **var** $k, l)$ partitions $T[i..j]$ into three sections: T is shuffled so the elements in $T[i..k]$ are smaller than p, those in $T[k+1..l-1]$ are equal to p, and those in $T[l..j]$ are larger than p. After a call on *pivotbis*(T, p, k, l), we are done if $k < s < l$ as the s–th smallest element of T is then equal to p. If $s \leq k$, the s–th smallest element of T is now the s–th smallest element of $T[1..k]$. Finally, if $s \geq l$, the s–th smallest element of T is now the $(s-l+1)$–st smallest element of $T[l..n]$. In any case, we have made progress since either we are done, or the subarray to be considered contains less than half the elements, by virtue of p being the median of the original array.

There are strong similarities between this approach and binary searching (Section 7.3), and indeed the resulting algorithm can be programmed iteratively rather than recursively. The key idea is to use two variables i and j, initialized to 1 and n respectively, and to ensure that at every moment $i \leq s \leq j$ and the elements in $T[1..i-1]$ are smaller than those in $T[i..j]$, which are in turn smaller than those in $T[j+1..n]$. The immediate consequence of this is that the desired element resides in $T[i..j]$. When all the elements in $T[i..j]$ are equal, we are done.

Figure 7.5 illustrates the process. For simplicity, the illustration assumes *pivotbis* is implemented in a way that is intuitively simple even though a really efficient implementation would proceed differently.

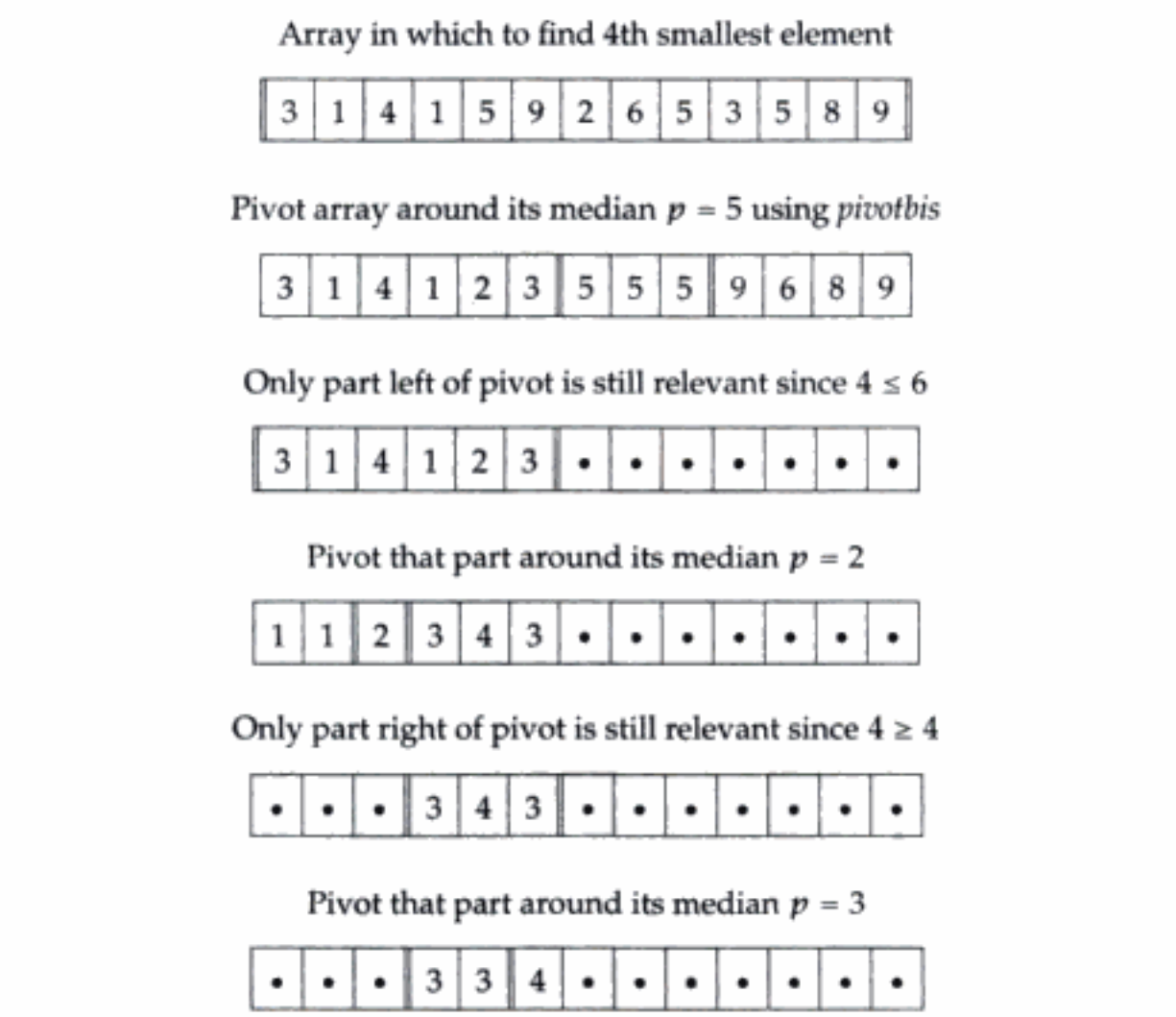

Figure 7.5. Selection using the median

```
function selection(T[1..n], s)
    {Finds the s-th smallest element in T, 1 ≤ s ≤ n}
    i ← 1; j ← n
    repeat
        {Answer lies in T[i..j]}
        p ← median(T[i..j])
        pivotbis(T[i..j], p, k, l)
        if s ≤ k then j ← k
        else if s ≥ l then i ← l
            else return p
```

By an analysis similar to that of binary search, the above algorithm selects the required element of T after going round the **repeat** loop a logarithmic number of times in the worst case. However, trips round the loop no longer take constant

time, and indeed this algorithm cannot be used until we have an efficient way to find the median, which was our original problem. Can we modify the algorithm to avoid resort to the median?

First, observe that our algorithm still works regardless of which element of T is chosen as pivot (the value of p). It is only the *efficiency* of the algorithm that depends on the choice of pivot: using the median assures us that the number of elements still under consideration is at least halved each time round the **repeat** loop. If we are willing to sacrifice speed in the worst case to obtain an algorithm reasonably fast on the average, we can borrow another idea from quicksort and simply choose $T[i]$ as pivot. In other words, replace the first instruction in the loop with

$$p \leftarrow T[i] .$$

This causes the algorithm to spend quadratic time in the worst case, for example if the array is in decreasing order and we wish to find the smallest element. Nevertheless, this modified algorithm runs in *linear* time on the average, under our usual assumption that the elements of T are distinct and that each of the $n!$ possible initial permutations of the elements is equally likely. (The analysis parallels that of *quicksort*; see Problem 7.18). This is much better than the time required on the average if we proceed by sorting the array, but the worst-case behaviour is unacceptable for many applications.

Happily, this quadratic worst case can be avoided without sacrificing linear behaviour on the average. The idea is that the number of trips round the loop remains logarithmic provided the pivot is chosen reasonably close to the median. A good *approximation* to the median can be found quickly with a little cunning. Consider the following algorithm.

```
function pseudomed(T[1..n])
    {Finds an approximation to the median of array T}
    if n ≤ 5 then return adhocmed(T)
    z ← ⌊n/5⌋
    array Z[1..z]
    for i ← 1 to z do Z[i] ← adhocmed(T[5i − 4..5i])
    return selection(Z, ⌈z/2⌉)
```

Here, *adhocmed* is an algorithm specially designed to find the median of at most five elements, which can be done in a time bounded above by a constant, and *selection*$(Z, \lceil z/2 \rceil)$ determines the exact median of array Z. Let p be the value returned by a call on *pseudomed*(T). How far from the true median of T can p be when $n > 5$?

As in the algorithm, let z be $\lfloor n/5 \rfloor$, the number of elements in the array Z created by the call on *pseudomed*(T). For each i between 1 and z, $Z[i]$ is by definition the median of $T[5i-4..5i]$, and therefore at least three elements out of the five in this subarray are less than or equal to it. Moreover, since p is the true median of Z, at least $z/2$ elements of Z are less than or equal to p. By transitivity ($T[j] \leq Z[i] \leq p$ implies that $T[j] \leq p$), at least $3z/2$ elements of T are less than or equal to p. Since $z = \lfloor n/5 \rfloor \geq (n-4)/5$, we conclude that at least $(3n-12)/10$ elements of T are

less than or equal to p, and therefore at most the $(7n + 12)/10$ remaining elements of T are strictly larger than p. Similar reasoning applies to the number of elements of T that are strictly *smaller* than p.

Although p is probably not the exact median of T, we conclude that its rank is approximately between $3n/10$ and $7n/10$. To visualize how these factors arise, although nothing in the execution of the algorithm *pseudomed* really corresponds to this illustration, imagine as in Figure 7.6 that the elements of T are arranged in five rows, with the possible exception of at most four elements left aside. Now suppose each of the $\lfloor n/5 \rfloor$ columns as well as the middle row are sorted by magic, the smallest elements going to the top and to the left, respectively. The middle row corresponds to the array Z in the algorithm and the element in the circle corresponds to the median of this array, which is the value of p returned by the algorithm. Clearly, each of the elements in the box is less than or equal to p. The conclusion follows since the box contains approximately three-fifths of one-half of the elements of T.

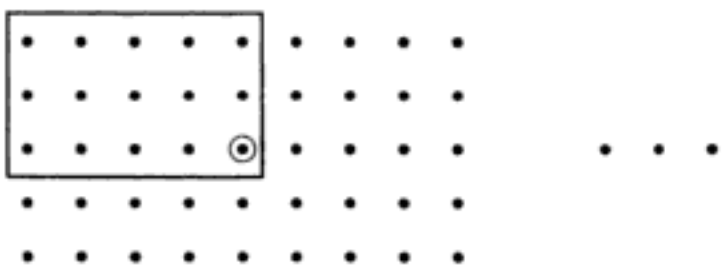

Figure 7.6. Visualization of the pseudomedian

We now analyse the efficiency of the *selection* algorithm presented at the beginning of this section when the first instruction in its **repeat** loop is replaced by

$$p \leftarrow pseudomed(T[i..j]) .$$

Let $t(n)$ be the time required in the worst case by a call on $selection(T[1..n], s)$. Consider any i and j such that $1 \leq i \leq j \leq n$. The time required to complete the **repeat** loop with these values for i and j is essentially $t(m)$, where $m = j - i + 1$ is the number of elements still under consideration. When $n > 5$, calculating $pseudomed(T)$ takes a time in $t(\lfloor n/5 \rfloor) + O(n)$ because the array Z can be constructed in linear time since each call to *adhocmed* takes constant time. The call to *pivotbis* also takes linear time. At this point, either we are finished or we have to go back round the loop with at most $(7n + 12)/10$ elements still to be considered. Therefore, there exists a constant d such that

$$t(n) \leq dn + t(\lfloor n/5 \rfloor) + \max\{t(m) \mid m \leq (7n + 12)/10\} \qquad (7.6)$$

provided $n > 5$.

Equation 7.1 does not help us solve this recurrence, so once again we resort to constructive induction. This time, even guessing the answer in asymptotic notation requires insight. (Recall that the obvious try when we analysed *quicksort* was

$O(n \log n)$ because we had analysed *mergesort* already, and it worked; no such luck this time.) With some experience, the fact that $\frac{1}{5} + \frac{7}{10} < 1$ is a telltale that $t(n)$ may well be linear in n, which is clearly the best we could hope for; see Problem 7.19.

> Theorem 7.5.1 *The selection algorithm used with pseudomed finds the s–th smallest among n elements in a time in $\Theta(n)$ in the worst case. In particular, the median can be found in linear time in the worst case.*

Proof Let $t(n)$ and d be as above. Clearly, $t(n) \in \Omega(n)$ since the algorithm must look at each element of T at least once. Thus it remains to prove that $t(n) \in O(n)$. Let us postulate the existence of a constant c, unknown as yet, such that $t(n) \leq cn$ for all $n \geq 1$. We find an appropriate value for this constant in the process of proving its existence by generalized mathematical induction. Constructive induction will also be used to determine the constant n_0 that separates the basis case from the induction step. For now, our only constraint is $n_0 \geq 5$ because Equation 7.6 only applies when $n > 5$. (We shall discover that the obvious choice $n_0 = 5$ does not work.)

- *Basis*: Consider any integer n such that $1 \leq n \leq n_0$. We have to show that $t(n) \leq cn$. This is easy since we still have complete freedom to choose the constant c and the number of basis cases is finite. It suffices to choose c at least as large as $t(n)/n$. Thus, our first constraint on c is

$$c \geq t(n)/n \text{ for all } n \text{ such that } 1 \leq n \leq n_0. \tag{7.7}$$

- *Induction step*: Consider any integer $n > n_0$. Assume the induction hypothesis that $t(m) \leq cm$ when $1 \leq m < n$. We wish to constrain c so that $t(n) \leq cn$ follows from the induction hypothesis. Starting with Equation 7.6, and because $1 \leq (7n + 12)/10 < n$ when $n > n_0 \geq 5$,

$$\begin{aligned} t(n) &\leq dn + t(\lfloor n/5 \rfloor) + \max\{t(m) \mid m \leq (7n + 12)/10\} \\ &\leq dn + cn/5 + (7n + 12)c/10 \quad \text{by the induction hypothesis} \\ &= 9cn/10 + dn + 6c/5 \\ &= cn - (c/10 - d - 6c/5n)\, n. \end{aligned}$$

It follows that $t(n) \leq cn$ provided $c/10 - d - 6c/5n \geq 0$, which is equivalent to $(1 - 12/n)\, c \geq 10d$. This is possible provided $n \geq 13$ (so $1 - 12/n > 0$), in which case c must be no smaller than $10d/(1 - 12/n)$. Keeping in mind that $n > n_0$, any choice of $n_0 \geq 12$ is adequate, provided c is chosen accordingly. More precisely, all is well provided $n_0 \geq 12$ and

$$c \geq \frac{10d}{1 - \frac{12}{n_0 + 1}} \tag{7.8}$$

which is our second and final constraint on c and n_0. For instance, the induction step is correct if we take $n_0 = 12$ and $c \geq 130d$, or $n_0 = 23$ and $c \geq 20d$, or $n_0 = 131$ and $c \geq 11d$.

Putting together the constraints given by Equations 7.7 and 7.8, and choosing $n_0 = 23$ for the sake of definiteness, it suffices to set

$$c = \max(20d, \max\{t(m)/m \mid 1 \leq m \leq 23\})$$

to conclude the proof by constructive induction that $t(n) \leq cn$ for all $n \geq 1$. ■

7.6 Matrix multiplication

Let A and B be two $n \times n$ matrices to be multiplied, and let C be their product. The classic matrix multiplication algorithm comes directly from the definition

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} .$$

Each entry in C is calculated in a time in $\Theta(n)$, assuming that scalar addition and multiplication are elementary operations. Since there are n^2 entries to compute, the product AB can be calculated in a time in $\Theta(n^3)$.

Towards the end of the 1960s, Strassen caused a considerable stir by improving this algorithm. From an algorithmic point of view, this breakthrough is a landmark in the history of divide-and-conquer, even though the equally surprising algorithm for multiplying large integers (Section 7.1) was discovered almost a decade earlier. The basic idea behind Strassen's algorithm is similar to that earlier one. First we show that two 2×2 matrices can be multiplied using less than the eight scalar multiplications apparently required by the definition. Let

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \text{ and } B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

be two matrices to be multiplied. Consider the following operations, each of which involves just one multiplication.

$$\left.\begin{aligned} m_1 &= (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}) \\ m_2 &= a_{11} b_{11} \\ m_3 &= a_{12} b_{21} \\ m_4 &= (a_{11} - a_{21})(b_{22} - b_{12}) \\ m_5 &= (a_{21} + a_{22})(b_{12} - b_{11}) \\ m_6 &= (a_{12} - a_{21} + a_{11} - a_{22}) b_{22} \\ m_7 &= a_{22}(b_{11} + b_{22} - b_{12} - b_{21}) \end{aligned}\right\} \quad (7.9)$$

We leave the reader to verify that the required product AB is given by the following matrix.

$$C = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix} \quad (7.10)$$

It is therefore possible to multiply two 2×2 matrices using only seven scalar multiplications. At first glance, this algorithm does not look very interesting: it uses