71/2003

Raport Badawczy
Research Report

RB/68/2003

Partitioning schemes for quicksort and quickselect

Krzysztof C. Kiwiel

Instytut Badań Systemowych
Polska Akademia Nauk

Systems Research Institute
Polish Academy of Sciences

POLSKA AKADEMIA NAUK

Instytut Badań Systemowych

ul. Newelska 6

01-447 Warszawa

tel.: (+48) (22) 8373578

fax: (+48) (22) 8372772

Kierownik Pracowni zgłaszający pracę:
Prof. dr hab. Krzysztof C. Kiwiel

Warszawa 2003

Partitioning schemes for quicksort and quickselect

Krzysztof C. Kiwiel*

December 20, 2003

Abstract

We introduce several modifications of the partitioning schemes used in Hoare's quicksort and quickselect algorithms, including ternary schemes which identify keys less or greater than the pivot. We give estimates for the numbers of swaps made by each scheme. Our computational experiments indicate that ternary schemes allow quickselect to identify all keys equal to the selected key at little additional cost.

Key words. Sorting, selection, quicksort, quickselect, partitioning.

1 Introduction

Hoare's quicksort [Hoa62] and quickselect (originally called FIND) [Hoa61b] are among the most widely used algorithms for sorting and selection. In our context, given an array $x[1\colon n]$ of n elements and a total order $<$, sorting means permuting the elements so that $x_i \leq x_{i+1}$ for $i = 1\colon n-1$, whereas for the simpler problem of selecting the kth smallest element, the elements are permuted so that $x_i \leq x_k \leq x_j$ for $1 \leq i \leq k \leq j \leq n$.

Both algorithms choose a pivot element, say v, and partition the input into a left array $x[1\colon a-1] \leq v$, a middle array $x[a\colon b] = v$, and a right array $x[b+1\colon n] \geq v$. Then quicksort is called recursively on the left and right arrays, whereas quickselect is called on the left array if $k < a$, or the right array if $k > b$; if $a \leq k \leq b$, selection is finished.

This paper introduces useful modifications of several partitioning schemes. First, we show that after exchanging x_1 with x_n when necessary, the classic scheme of Sedgewick [Knu98, §5.2.2] no longer needs an artificial sentinel. Second, it turns out that a simple modification of another popular scheme of Sedgewick [BeM93, Prog. 3] allows it to handle equal keys more efficiently; both schemes take n or $n+1$ comparisons. Third, we describe a scheme which makes just the $n-1$ necessary comparisons, as well as the minimum number of swaps when the elements are distinct. This should be contrasted with Lomuto's scheme [BeM93, Prog. 2], [CLRS01, §7.1], which takes $n-1$ comparisons but up to $n-1$ swaps. Hence we analyze the average numbers of swaps made by the four schemes when the elements are distinct and in random order. The first three schemes take at most $n/4$ swaps on average, whereas Lomuto's scheme takes up to $n-1$. Further, for the pivot selected

---

*Systems Research Institute, Newelska 6, 01–447 Warsaw, Poland (`kiwiel@ibspan.waw.pl`)

as the median of a sample of $2t+1$ elements, the first three schemes make asymptotically $n/6$ swaps for $t=0$, $n/5$ for $t=1$, etc. (cf. §3.3.1), while Lomuto's scheme takes $(n-1)/2$; the swap counts are similar when the pivot is Tukey's ninther [BeM93, CHT02, Dur03].

When equal keys occur, one may prefer a *ternary* scheme which produces a left array with keys $< v$ and a right array with keys $> v$, instead of $\le v$ and $\ge v$ as do *binary* schemes. Here only the Bentley-McIlroy scheme [BeM93] looks competitive, since Dijkstra's "Dutch national flag" scheme [Dij76, Chap. 14] and Wegner's schemes [Weg85] are more complex. However, the four schemes discussed above also have attractive ternary versions. Our first scheme omits pointer tests in its key comparison loops, keeping them as fast as possible. Our second scheme improves on another scheme of Sedgewick [Sed98, Chap. 7, quicksort] (which needn't produce true ternary partitions; cf. §5.2). Our third scheme is a simple modification of the Bentley-McIlroy scheme which makes $n-1$ comparisons; the original version takes $n-1/2$ on average (cf. Lem. 5.1), although $n-1$ was assumed in [Dur03]. Ternary versions of Lomuto's scheme seem to be less attractive. When many equal keys occur, the Bentley-McIlroy scheme tends to make fewer swaps than the other schemes, but it may swap needlessly equal keys with themselves and its inner loops involve pointer tests. Hence we introduce hybrid two-phase versions which eliminate vacuous swaps in the first phase and pointer tests in the second phase.

Ternary schemes, although slower than their simpler binary counterparts, have at least two advantages. First, quicksort's recursive calls aren't made on the equal keys isolated by partitioning. Second, quickselect can identify *all* keys equal to the kth smallest by finding two indices $k_- \le k \le k_+$ such that $x[1\colon k_- - 1] < x_k = x[k_-\colon k_+] < x[k_+ + 1\colon n]$ on output.

Our fairly extensive computational tests with quickselect (we left quicksort for future work) were quite suprising. First, the inclusion of pointer tests in the key comparison loops didn't result in significant slowdowns; this is in sharp contrast with traditional recommendations [Knu98, Ex. 5.2.2-24], [Sed78, p. 848], but agrees with the observation of [BeM93] that Knuth's MIX cost model needn't be appropriate for modern machines. Second, the overheads of ternary schemes relative to binary schemes were quite mild. Third, Lomuto's binary scheme was hopeless when many equal keys occured, since its running time may be quadratic in the number of keys equal to the kth smallest.

More information on theoretical and practical aspects of quicksort and quickselect can be found in [BeS97, Grü99, HwT02, KMP97, MaR01, Mus97, Val00] and references therein.

The paper is organized as follows. The four bipartitioning schemes of interest are described in §2 and their average-case analysis is given in §3. In §4 we present tuned versions (cf. [MaR01, §7]) for the case where the pivot is selected from a sample of several elements. Tripartitioning schemes are discussed in §5. Finally, our computational results are reported in §6.

2 Bipartitioning schemes

Each invocation of quicksort and quickselect deals with a subarray $x[l\colon r]$ of the input array $x[1\colon n]$; abusing notation, we let $n := r-l+1$ denote the size of the current subarray. It is convenient to assume that the pivot $v := x_l$ is placed first (after a possible exchange with another element). Each *binary* scheme given below partitions the array into three blocks, with $x_m \le v$ for $l \le m < a$, $x_m = v$ for $a \le m \le b$, $x_m \ge v$ for $b < m \le r$, $l \le a \le b \le r$. We suppose that $n > 1$ (otherwise partitioning is trivial: set $a := b := l$).

2.1 Safeguarded binary partition

Our first modification of the classic scheme of Sedgewick [Knu98, §5.2.2, Algorithm Q] proceeds as follows. After comparing the pivot $v := x_l$ to x_r to produce the initial setup

$$\begin{array}{|c|c|c|c|c|}\hline x = v & x < v & ? & x > v & x = v \\ \hline \end{array} \\ l \qquad\quad p \quad i \qquad\qquad j \quad q \qquad\qquad r \tag{2.1}$$

with $i := l$ and $j := r$, we work with the three inner blocks of the array

$$\begin{array}{|c|c|c|c|c|}\hline x = v & x \le v & ? & x \ge v & x = v \\ \hline \end{array} \\ l \qquad\quad p \qquad\quad i \;\; j \qquad\quad q \qquad\quad r \tag{2.2}$$

until the middle part is empty or just contains an element equal to the pivot

$$\begin{array}{|c|c|c|c|c|}\hline x = v & x \le v & x = v & x \ge v & x = v \\ \hline \end{array} \\ l \qquad\quad p \quad j \qquad\qquad i \quad q \qquad\qquad r \tag{2.3}$$

(i.e., $j = i - 1$ or $j = i - 2$), then swap the ends into the middle for the final arrangement

$$\begin{array}{|c|c|c|}\hline x \le v & x = v & x \ge v \\ \hline \end{array} . \\ l \qquad\quad a \quad b \qquad\quad r \tag{2.4}$$

Scheme A (Safeguarded binary partition).

A1. [Initialize.] Set $i := l$, $p := i + 1$, $j := r$ and $q := j - 1$. If $v > x_j$, exchange $x_i \leftrightarrow x_j$ and set $p := i$; else if $v < x_j$, set $q := j$.

A2. [Increase i until $x_i \ge v$.] Increase i by 1; then if $x_i < v$, repeat this step.

A3. [Decrease j until $x_j \le v$.] Decrease j by 1; then if $x_j > v$, repeat this step.

A4. [Exchange.] (Here $x_j \le v \le x_i$.) If $i < j$, exchange $x_i \leftrightarrow x_j$ and return to A2. If $i = j$ (so that $x_i = x_j = v$), increase i by 1 and decrease j by 1.

A5. [Cleanup.] Set $a := l + j - p + 1$ and $b := r - q + i - 1$. If $l < p$, exchange $x_l \leftrightarrow x_j$. If $q < r$, exchange $x_i \leftrightarrow x_r$.

Step A1 ensures that $x_i \le v \le x_j$, so steps A2 and A3 don't need to test whether $i \le j$. In other words, while searching for a pair of elements to exchange, the previously sorted data (initially, $x_l \le x_r$) are used to bound the search, and the index values are compared only when an exchange is to be made. This leads to a small amount of overshoot in the search: in addition to the necessary $n - 1$ comparisons, scheme A makes two spurious comparisons or just one (when $i = j + 1$ or $i = j$ at A4 respectively). Step A4 makes at most $n/2$ index comparisons and at most $n/2 - 1$ swaps (since $j - i$ decreases at least by 2 between swaps); thus A1 and A4 make at most $n/2$ swaps. To avoid vacuous swaps, step

A5 may use the tests $l < \min\{p, j\}$ and $\max\{q, i\} < r$; on the other hand, A5 could make unconditional swaps without impairing (2.4).

Of course, scheme A could be described in other equivalent ways. For instance, A1 and A5 can be written in terms of binary variables $i_l := p - l$ and $i_r := r - q$; then A5 may decrease j by 1 if $i_l = 1$ and increase i by 1 if $i_r = 1$ to have $a = j + 1$, $b = i - 1$ in (2.4).

A more drastic simplification could swap $x_l \leftrightarrow x_r$ if $v > x_r$ at A1, omit the second instruction of A4, set $a := b := j$ at A5 and swap $x_l \leftrightarrow x_j$ if $x_l = v$, $x_j \leftrightarrow x_r$ otherwise.

2.2 Single-index controlled binary partition

It is instructive to compare scheme A with a popular scheme of Sedgewick [BeM93, Progs. 3 and 4], based on the arrangements (2.2)–(2.3) with $p := l + 1$, $q := r$.

Scheme B (Single-index controlled binary partition).

B1. [Initialize.] Set $i := l$ and $j := r + 1$.

B2. [Increase i until $x_i \geq v$.] Increase i by 1; then if $i \leq r$ and $x_i < v$, repeat this step.

B3. [Decrease j until $x_j \leq v$.] Decrease j by 1; then if $x_j > v$, repeat this step.

B4. [Exchange.] (Here $x_j \leq v \leq x_i$.) If $i \leq j$, exchange $x_i \leftrightarrow x_j$ and return to B2.

B5. [Cleanup.] Exchange $x_l \leftrightarrow x_j$.

The test $i \leq r$ of step B2 is necessary when v is greater than the remaining elements. If $i = j$ at B4, a vacuous swap is followed by one or two unnecessary comparisons; hence B4 may be replaced by A4 to achieve the same effect at no extra cost. With this replacement, scheme B makes $n + 1$ comparisons or n if $i = j$ or $i = r + 1$ at B4, and at most $(n + 1)/2$ index comparisons and $(n - 1)/2$ swaps at B4. Usually scheme B is used as if $a := b := j$ in (2.4), but in fact B5 may set $a := j$, $b := i - 1$ (note that the final arrangement of [BeM93, p. 1252] is wrong when $j = i - 2$). Therefore, from now on, we assume that scheme B incorporates our suggested modifications of steps B4 and B5.

2.3 Double-index controlled binary partition

The following scheme compares both scanning indices i and j in their inner loops.

Scheme C (Double-index controlled binary partition).

C1. [Initialize.] Set $i := l + 1$ and $j := r$.

C2. [Increase i until $x_i \geq v$.] If $i \leq j$ and $x_i < v$, increase i by 1 and repeat this step.

C3. [Decrease j until $x_j \leq v$.] If $i < j$ and $x_j > v$, decrease j by 1 and repeat this step. If $i \geq j$, set $j := i - 1$ and go to C5.

C4. [Exchange.] Exchange $x_i \leftrightarrow x_j$, increase i by 1, decrease j by 1 and return to C2.

C5. [Cleanup.] Set $a := b := j$. Exchange $x_l \leftrightarrow x_j$.

Thanks to its tight index control, scheme C makes just $n - 1$ comparisons and at most $(n - 1)/2$ swaps at C4. Suprisingly, we have not found this scheme in the literature.

2.4 Lomuto's binary partition

We now consider Lomuto's partition [BeM93, Prog. 2], based on the arrangements

$$\begin{array}{|c|c|c|c|}\hline v & x < v & x \geq v & ? \\ \hline \multicolumn{1}{c}{l} & \multicolumn{1}{c}{p} & \multicolumn{1}{r}{i} & \multicolumn{1}{r}{r} \end{array} \quad \longrightarrow \quad \begin{array}{|c|c|c|}\hline v & x < v & x \geq v \\ \hline \multicolumn{1}{c}{l} & \multicolumn{1}{c}{p} & \multicolumn{1}{r}{r} \end{array} . \tag{2.5}$$

Scheme D (Lomuto's binary partition).

D1. [Initialize.] Set $i := l + 1$ and $p := l$.

D2. [Check if done.] If $i > r$, go to D4.

D3. [Exchange if necessary.] If $x_i < v$, increase p by 1 and exchange $x_p \leftrightarrow x_i$. Increase i by 1 and return to D2.

D4. [Cleanup.] Set $a := b := p$. Exchange $x_l \leftrightarrow x_p$.

At the first sight, scheme D looks good: it makes just the $n - 1$ necessary comparisons. However, it can make up to $n - 1$ swaps (e.g., vacuous swaps when v is greater than the remaining elements, or $n - 2$ nonvacuous swaps for $x[l\colon r] = [n-1, n, 1, 2, \ldots, n-2]$).

2.5 Comparison of bipartitioning schemes

2.5.1 Swaps for distinct keys

When the elements are distinct, we have strict inequalities in (2.2)–(2.5), $j = i - 1$ in (2.3) and $a = b$ in (2.4). Distinguishing *low* keys $x_m < v$ and *high* keys $x_m > v$, let t be the number of high keys in the input subarray $x[l + 1\colon a]$. Then schemes B and C make the *same* sequence of t swaps to produce the arrangement

$$\begin{array}{|c|c|c|}\hline v & x < v & x > v \\ \hline \multicolumn{1}{c}{l} & \multicolumn{1}{r}{a} & \multicolumn{1}{r}{r} \end{array} \tag{2.6}$$

before the final swap $x_l \leftrightarrow x_a$, and their operation is described by the instruction: until there are no high keys in $x[l + 1\colon a]$, swap the leftmost high key in $x[l + 1\colon a]$ with the rightmost low key in $x[a + 1\colon r]$. Thus schemes B and C make just the necessary t swaps. Scheme A acts in the same way if $x_r > v$ at A1. If $x_r < v$ at A1, let t_l be the number of low keys in $x[a\colon r]$; in this *low* case, after the initial swap $x_l \leftrightarrow x_r$, scheme A makes $t_l - 1$ swaps, each time exchanging the leftmost high key in $x[l + 1\colon a - 1]$ with the rightmost low key in $x[a\colon r - 1]$, to produce the arrangement

$$\begin{array}{|c|c|c|}\hline x < v & x > v & v \\ \hline \multicolumn{1}{l}{l} & \multicolumn{1}{c}{a} & \multicolumn{1}{c}{r} \end{array} \tag{2.7}$$

before the final swap $x_a \leftrightarrow x_r$. Since the number of low keys in $x[a + 1\colon r]$ equals t, we have $t_l = t + 1$ if $x_a < v$, otherwise $t_l = t$. Thus, relative to schemes B and C, scheme A makes an extra swap when both x_a and x_r are low. Note that schemes A, B and C never swap the same key twice while producing the arrangements (2.6)–(2.7). In contrast, scheme D may swap the same high key many times while producing the arrangement (2.6) (usually different from that of B and C). In fact scheme D makes exactly $t_{\rm D} := a - l$ swaps; this is the total number of low keys. Thus the number of extra swaps made by scheme D relative to B and C, $t_{\rm D} - t$, equals the number of low keys in the initial $x[l + 1\colon a]$.

2.5.2 Swaps for equal keys

When equal keys occur, schemes A, B and C perform similarly to Sedgewick's scheme of [Sed77, Prog. 1]; in particular, thanks to stopping the scanning pointers on keys equal to the pivot, they tend to produce balanced partitions. For instance, when all the keys are equal, we get the following partitions: for scheme A, $a = \lfloor (l+r-1)/2 \rfloor$, $b = a+1+(n \bmod 2)$ after $\lceil (n + 1)/2 \rceil$ swaps; for scheme B, $a = \lceil (l + r)/2 \rceil$, $b = a + 1 - (n \bmod 2)$ after $\lceil n/2 \rceil$ swaps; for scheme C, $a = b = \lceil (l + r)/2 \rceil$ after $\lceil n/2 \rceil$ swaps. In contrast, scheme D makes no swaps, but yields $a = b = l$, the worst possible partition.

3 Average-case analysis of bipartitioning schemes

In this section we assume that the keys to be partitioned are distinct and in random order; since the schemes depend only on the relative order of the keys, we may as well assume that they are the first n positive integers in random order. For simplier notation, we suppose that $l = 1$ and $r = n$. It is easy to see that when the keys in $x[l + 1\colon r]$ are in random order, each scheme of §2 *preserves randomness* in the sense of producing $x[l\colon a - 1]$ and $x[a + 1\colon r]$ in which the low and high keys are in random order (since the relative orders of the low keys and the high keys on input have no effect on the scheme).

3.1 Expected numbers of swaps for fixed pivot ranks

For a given pivot $v := x_1$, let j_v denote the number of low keys in the array $x[2\colon n]$; then $a = j_v + 1$ is the rank of v. Once j_v is fixed at j (say), to compute the average number of swaps made by each scheme, it's enough to assume that the keys in $x[2\colon n]$ are in random order; thus averages are taken over the $(n - 1)!$ distinct inputs. Our analysis hinges on the following well-known fact (cf. [Chv02]).

Fact 3.1. *Suppose an array $x[\hat{l}\colon \hat{r}]$ contains $\hat{n} := \hat{r} - \hat{l} + 1 > 0$ distinct keys, of which $\hat{\jmath}$ are low and $\hat{n} - \hat{\jmath}$ are high. If all the $\hat{n}!$ permutations of the keys are equiprobable, then $\hat{\jmath}(\hat{n} - \hat{\jmath})/\hat{n}$ is the average number of high keys in the first $\hat{\jmath}$ positions.*

Proof. List all the $\hat{n}!$ key permutations as rows of an $\hat{n}! \times \hat{n}$ matrix. In each column, each key appears $(\hat{n} - 1)!$ times, so the number of high keys in the first $\hat{\jmath}$ columns is $\hat{\jmath}(\hat{n} - \hat{\jmath})(\hat{n} - 1)!$; dividing by $\hat{n}!$ gives the average number $\hat{\jmath}(\hat{n} - \hat{\jmath})/\hat{n}$. □

Lemma 3.2. *Suppose the number of low key equals j. Let $T_j^{\rm A}$, $T_j^{\rm B}$, $T_j^{\rm C}$, $T_j^{\rm D}$ denote the average numbers of swaps made by schemes* A, B, C *and* D, *excluding the final swaps. Then*

$$T_j^{\rm A} = \frac{j(n-1-j)}{n-1}\frac{n-3}{n-2} + \frac{j}{n-1}, \quad n \geq 3, \tag{3.1a}$$

$$T_j^{\mathrm{A}} = \frac{j}{n-1}, \quad n = 2, \tag{3.1b}$$

$$T_j^{\mathrm{B}} = T_j^{\mathrm{C}} = \frac{j(n-1-j)}{n-1}, \tag{3.2}$$

$$T_j^{\mathrm{D}} = j. \tag{3.3}$$

Proof. By assumption, the arrangements (2.6)–(2.7) involve $l = 1$, $a = j + 1$, $r = n$. The results follow from suitable choices of $\hat{l}$, $\hat{r}$, $\hat{\jmath}$ in Fact 3.1.

For scheme A, assuming $n \geq 3$, let $\hat{l} = 2$, $\hat{r} = n - 1$. Depending on whether $x_n > v$ or $x_n < v$, scheme A produces either (2.6) or (2.7) from the initial configurations

$$\begin{array}{|c|c|c|c|}\hline v & \quad & \quad & x > v \\ \hline \multicolumn{1}{c}{1} & \multicolumn{1}{c}{a} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{n} \end{array} \quad \text{or} \quad \begin{array}{|c|c|c|c|}\hline v & \quad & \quad & x < v \\ \hline \multicolumn{1}{c}{1} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{a} & \multicolumn{1}{c}{n} \end{array}. \tag{3.4}$$

For $x_n > v$, take $\hat{\jmath} = j = a - 1$; then the average number of high keys in $x[2\colon a]$ (i.e., of swaps) equals $j(n-2-j)/(n-2)$. For $x_n < v$, take $\hat{\jmath} = j - 1$; in this case $t_l - 1$, the number of low keys in $x[a\colon n-1]$, equals the number of high keys in $x[2\colon j]$, so the average value of t_l equals $(j-1)(n-1-j)/(n-2)+1$. Since there are j low keys and $n-1-j$ high keys which appear in random order, we have $x_n > v$ with probability $(n-1-j)/(n-1)$ and $x_n < v$ with probability $j/(n-1)$. Adding the contributions of these cases multiplied by their probabilities yields (3.1a). For $n = 2$, A1 makes 1 swap if $j = 1$, 0 otherwise, so (3.1b) holds.

For schemes B and C, take $\hat{l} = 2$, $\hat{r} = n$, $\hat{\jmath} = j$ to get (3.2) in a similar way.

Since scheme D makes $t_{\mathrm{D}} := a - l = j$ swaps, (3.3) follows. $\square$

To compare the average values (3.1)–(3.3), note that we have $0 \leq j \leq n - 1$,

$$T_j^{\mathrm{A}} = T_j^{\mathrm{B}} + \frac{j(j-1)}{(n-1)(n-2)} \quad \text{and} \quad T_j^{\mathrm{D}} = T_j^{\mathrm{A}} + \frac{j + (n-3)j^2}{(n-1)(n-2)} \quad \text{if } n \geq 3, \tag{3.5}$$

$T_j^{\mathrm{B}} = 0$ and $T_j^{\mathrm{A}} = T_j^{\mathrm{D}} = j$ if $n = 2$. Thus $T_j^{\mathrm{B}} \leq T_j^{\mathrm{A}} + 1$ (with equality iff there are no high keys), whereas T_j^{D} is much greater than T_j^{A} when there are relatively many low keys.

3.2 Bounding expected numbers of swaps for arbitrary pivots

From now on we assume that the pivot is selected by an arbitrary rule for which (once the pivot is swapped into x_l if necessary) each permutation of the remaining keys is equiprobable. Let T_{A}, T_{B}, T_{C}, T_{D} denote the average numbers of swaps made by schemes A, B, C and D, *excluding* the final swaps. Of course, these numbers depend on details of pivot selection, but they can be bounded independently of such details. To this end we compute the maxima of the average values (3.1)–(3.3).

Lemma 3.3. *Let* $T_{\max}^{\mathrm{A}}$, $T_{\max}^{\mathrm{B}}$, $T_{\max}^{\mathrm{C}}$, $T_{\max}^{\mathrm{D}}$ *denote the maxima of* T_j^{A}, T_j^{B}, T_j^{C}, T_j^{D} *over* $0 \leq j < n$. *Then*

$$T_{\max}^{\mathrm{A}} = \begin{cases} \dfrac{n}{4} - \dfrac{(n-5)(n \bmod 2)}{4(n-1)(n-2)} & \textit{if } n \geq 5, \\ 1 & \textit{if } n \leq 4, \end{cases} \tag{3.6}$$

$$T^{\mathrm{B}}_{\max} = T^{\mathrm{C}}_{\max} = \frac{n-1}{4} - \frac{(n+1) \bmod 2}{4(n-1)}, \tag{3.7}$$

$$T^{\mathrm{D}}_{\max} = n - 1. \tag{3.8}$$

Proof. The maximum of (3.1) is attained at $j = \lceil n/2 \rceil$ if $n \geq 4$, $j = n-1$ otherwise. The maximum of (3.2) is attained at $j = \lfloor n/2 \rfloor$. The rest follows by simple computations. □

Corollary 3.4. *The average numbers of swaps T_{A}, T_{B}, T_{C}, T_{D} made by schemes* A, B, C, D *are at most $T^{\mathrm{A}}_{\max}$, $T^{\mathrm{B}}_{\max}$, $T^{\mathrm{C}}_{\max}$, $T^{\mathrm{D}}_{\max}$ for the values given in* (3.6)–(3.8). *In particular, T_{A}, T_{B} and T_{C} are at most $n/4$ for $n > 3$.*

3.3 The case where pivots are chosen via sampling

3.3.1 Pivots with fixed sample ranks

We assume that the pivot v is selected as the $(p+1)$th element in a sample of size s, $0 \leq p < s \leq n$. Thus p and $q := s - 1 - p$ are the numbers of low and high keys in the sample, respectively. Recall that v has rank $j_v + 1$, where j_v is the total number of low keys. We shall need the following two expected values for this selection:

$$\mathrm{E} j_v = E(n, s, p) := (p+1)(n+1)/(s+1) - 1, \tag{3.9}$$

$$\mathrm{E}\left[\frac{j_v(n-1-j_v)}{n-1}\right] = T(n, s, p) := \frac{(p+1)(q+1)}{(s+1)(s+2)} \frac{(n+1)(n+2)}{n-1} - \frac{n}{n-1}. \tag{3.10}$$

Here (3.9) follows from [FlR75, Eq. (10)] and (3.10) from the proof of [MaR01, Lem. 1].

Theorem 3.5. *For $E(n,s,p)$ and $T(n,s,p)$ given by* (3.9)–(3.10), *the average numbers of swaps T_{A}, T_{B}, T_{C}, T_{D} made by schemes* A, B, C, D *are equal to, respectively,*

$$T_{\mathrm{A}}(n, s, p) = \frac{\max\{n-3, 0\}}{\max\{n-2, 1\}} T(n, s, p) + \frac{1}{n-1} E(n, s, p), \tag{3.11}$$

$$T_{\mathrm{B}}(n, s, p) = T_{\mathrm{C}}(n, s, p) = T(n, s, p), \tag{3.12}$$

$$T_{\mathrm{D}}(n, s, p) = E(n, s, p). \tag{3.13}$$

Proof. Take expectations of the averages (3.1)–(3.3) conditioned on $j_v = j$, and use (3.9)–(3.10); the two "max" operations in (3.11) combine the cases of $n = 2$ and $n \geq 3$. □

The average values (3.11)–(3.13) may be compared as follows. First, in the classic case of $s = 1$ ($p = q = 0$), we have $T_{\mathrm{A}} = n/6$ if $n \geq 3$ (else $T_{\mathrm{A}} = 1/2$), $T_{\mathrm{B}} = (n-2)/6$, $T_{\mathrm{D}} = (n-1)/2$; thus scheme D makes about three times as many swaps as A, B and C.

Second, for nontrivial samples ($s > 1$) one may ask which choices of p are "good" or "bad" with respect to swaps. For schemes B and C, the worst case occurs if p is chosen to maximize (3.10) (where $q + 1 = s - p$); we obtain that for all $0 \leq p < s$,

$$T(n, s, p) \leq \kappa(s) \frac{(n+1)(n+2)}{n-1} - \frac{n}{n-1} \leq \frac{n-1}{4} \quad \text{with} \quad \kappa(s) := \frac{s+1}{4(s+2)}, \tag{3.14}$$

where the first inequality holds as equality only for the *median-of-s* choice of $p = (s-1)/2$, and the second one iff $s = n$. Since $T_{\rm A} \le T_{\rm B} + 1$, (3.14) yields $T_{\rm A} \le (n+3)/4$, but we already know that $T_{\rm A} \le n/4$ (Cor. 3.4). For any median-of-s choice with a fixed s, $T_{\rm A}$ and $T_{\rm B}$ are asymptotically $\kappa(s)n$, whereas $E(n,s,p) = (n-1)/2$; thus scheme D makes about $1/2\kappa(s) > 2$ times as many swaps as A, B and C (with $\kappa(3) = 1/5$, $\kappa(5) = 3/14$, $\kappa(7) = 2/9$, $\kappa(9) = 5/22$). On the other hand, for the extreme choices of $p = 0$ or $p = s-1$ which minimize (3.10) (then v is the smallest or largest key in the sample), $T_{\rm A}$ and $T_{\rm B}$ are asymptotically $ns/(s+1)(s+2)$, whereas $T_{\rm D}$ is asymptotically $n/(s+1)$ for $p = 0$ and $ns/(s+1)$ for $p = s-1$. Thus scheme D can't improve upon A and B even for the choice of $p = 0$ which minimizes (3.9).

3.3.2 Pivots with random sample ranks

Following the general framework of [CHT02, §1], suppose the pivot v is selected by taking a random sample of s elements, and choosing the $(p+1)$th element in this sample with probability π_p, $0 \le p < s$, $\sum_{p=0}^{s-1} \pi_p = 1$. In other words, for p_v denoting the number of low keys in the sample, we have $\Pr[p_v = p] = \pi_p$. Hence, by viewing (3.9)–(3.13) as expectations conditioned on the event $p_v = p$, we may take total averages to get

$$\mathrm{E} j_v = \mathrm{E}\left[E(n,s,p_v)\right] = E(n,s) := (\mathrm{E}p_v + 1)(n+1)/(s+1) - 1, \tag{3.15}$$

$$\mathrm{E}\left[\frac{j_v(n-1-j_v)}{n-1}\right] = \mathrm{E}\left[T(n,s,p_v)\right] = T(n,s) := \sum_{0\le p<s} \pi_p T(n,s,p), \tag{3.16}$$

and the following extension of Theorem 3.5.

Theorem 3.6. *For $E(n,s)$ and $T(n,s)$ given by* (3.15)–(3.16), *the average numbers of swaps $T_{\rm A}$, $T_{\rm B}$, $T_{\rm C}$, $T_{\rm D}$ made by schemes* A, B, C, D *are equal to, respectively,*

$$T_{\rm A}(n,s) = \frac{\max\{n-3,0\}}{\max\{n-2,1\}} T(n,s) + \frac{1}{n-1} E(n,s), \tag{3.17}$$

$$T_{\rm B}(n,s) = T_{\rm C}(n,s) = T(n,s), \tag{3.18}$$

$$T_{\rm D}(n,s) = E(n,s). \tag{3.19}$$

Note that in (3.15)–(3.16), we have $\mathrm{E}p_v = \sum_{0\le p<s} \pi_p p \le s-1$ and

$$T(n,s) = \breve{\kappa}(s)\frac{(n+1)(n+2)}{n-1} - \frac{n}{n-1} \quad \text{with} \quad \breve{\kappa}(s) := \sum_{0\le p<s} \pi_p \frac{(p+1)(s-p)}{(s+1)(s+2)}, \tag{3.20}$$

where $\breve{\kappa}(s) \le \kappa(s)$ (cf. (3.14)), and $\breve{\kappa}(s) = \kappa(s)$ iff $\pi_p = 1$ for $p = (s-1)/2$. Thus again $T_{\rm A}$ and $T_{\rm B}$ are asymptotically $\breve{\kappa}(s)n$, whereas $T_{\rm D}$ can be much larger.

As an important example, we consider *Tukey's ninther*, the median of three elements each of which is the median of three elements [BeM93]. Then $s = 9$ and $\pi_p = 0$ except for $\pi_3 = \pi_5 = 3/14$, $\pi_4 = 3/7$ [CHT02, Dur03], so $E(n,9) = (n-1)/2$ and $\breve{\kappa}(9) = 86/385 \approx 0.223$. Thus, when the ninther replaces the median-of-3, $T_{\rm A}$ and $T_{\rm B}$ increase by about 12 percent, getting closer to $n/4$, whereas $T_{\rm D}$ stays at $(n-1)/2$.

4 Using sample elements as sentinels

The schemes of §2 can be tuned [MaR01, §7.2] when the pivot v is selected as the $(p+1)$th element in a sample of size s, assuming $0 \le p < s \le n$ and $q := s - 1 - p > 0$.

First, suppose the p sample keys $\le v$ are placed first, followed by v, and the remaining q sample keys $\ge v$ are placed at the end of the array $x[l:r]$. Then, for $\bar l := l + p$ and $\bar r := r - q$, we only need to partition the array $x[\bar l:\bar r]$ of size $\bar n := n - s + 1$. The schemes of §2 are modified as follows.

In step A1 of scheme A, set $i := \bar l$ and $j := \bar r + 1$; in step A5 set $a := j$, $b := i - 1$ and exchange $x_{\bar l} \leftrightarrow x_j$. This scheme makes $\bar n + 1$ comparisons, or just $\bar n$ if $i = j$ at A4. The same scheme results from scheme B by replacing l, r with $\bar l$, $\bar r$, B4 with A4, and omitting the test "$i \le r$" in B2. Similarly, $\bar l$, $\bar r$ replace l and r in schemes C and D, which make $\bar n - 1$ comparisons.

To extend the results of §3 to these modifications, note that for $\bar n = 1$ these schemes make no swaps except for the final ones. For $\bar n > 1$, schemes A, B and C swap the same keys, if any. Therefore, under the sole assumption that the keys in $x[\bar l + 1:\bar r]$ are distinct and in random order, Lemma 3.2 holds with (3.1)–(3.3) replaced by

$$T_j^{\mathrm{A}} = T_j^{\mathrm{B}} = T_j^{\mathrm{C}} = \frac{(j-p)(n-1-q-j)}{n-s} \qquad \text{and} \qquad T_j^{\mathrm{D}} = j - p, \tag{4.1}$$

using $\hat l = \bar l + 1$, $\hat r = \bar r$, $\hat j = j - p$ in Fact 3.1; further, Lemma 3.3 and Corollary 3.4 hold with n replaced by $\bar n$, (3.6) omitted and $T^{\mathrm{A}}_{\max} = T^{\mathrm{B}}_{\max}$ in (3.7). Next, (3.9)–(3.10) are replaced by

$$\mathrm{E} j_v - p = \hat E(n,s,p) := (p+1)(n-s)/(s+1), \tag{4.2}$$

$$\mathrm{E}\left[\frac{(j_v - p)(n-1-q-j_v)}{n-s}\right] = \hat T(n,s,p) := \frac{(p+1)(q+1)}{(s+1)(s+2)}(n-s-1), \quad s < n, \tag{4.3}$$

where (4.3) is obtained similarly to (3.10) [MaR01, §7.2]. In view of (4.1)–(4.3), Theorem 3.5 holds with $E(n,s,p)$, $T(n,s,p)$ replaced by $\hat E(n,s,p)$, $\hat T(n,s,p)$, (3.11) omitted and $T_{\mathrm{A}}(n,s,p) = T_{\mathrm{B}}(n,s,p)$ in (3.12). Finally, (3.14) is replaced by

$$\hat T(n,s,p) \le \kappa(s)(n-s-1) < \frac{n-s-1}{4} \quad \text{with} \quad \kappa(s) := \frac{s+1}{4(s+2)}, \tag{4.4}$$

where the equality holds iff $p = (s-1)/2$, in which case $\hat E(n,s,p) = (n-s)/2$.

Randomness may be lost when the sample keys are rearranged by pivot selection, but it is preserved for the median-of-3 selection with $p = q = 1$. Then the sample keys usually are x_l, x_{l+1}, x_r (after exchanging x_{l+1} with the middle key $x_{\lfloor (l+r)/2 \rfloor}$). Arranging the sample according to Figure 4.1 takes 8/3 comparisons and 7/6 swaps on average for distinct keys. (These counts hold if, for simpler coding, only the left subtree is used after exchanging $a \leftrightarrow c$ when $a > c$; other trees [BeM93, Prog. 5] take 3/2 swaps for such simplifications.)

Even if pivot selection doesn't rearrange the array (except for placing the pivot in x_l), scheme A may be simplified: in step A1, set $i := l$ and $j := r + 1$; in step A5 set $a := j$, $b := i - 1$ and exchange $x_l \leftrightarrow x_j$. The same scheme results from scheme B by replacing B4 with A4, and omitting the test "$i \le r$" in B2. This simplification is justified by the

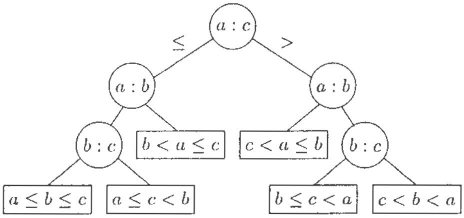

Figure 4.1: Decision tree for median of three

presence of at least one key $\geq v$ in $x[l+1\colon r]$, which stops the scanning index i. Hence the results of §3 remain valid (with (3.1), (3.5), (3.6), (3.11), (3.17) omitted, $T_j^{\mathrm{A}} = T_j^{\mathrm{B}}$ in (3.2), $T_{\max}^{\mathrm{A}} = T_{\max}^{\mathrm{B}}$ in (3.7), $T_{\mathrm{A}}(n,s,p) = T_{\mathrm{B}}(n,s,p)$ in (3.12), $T_{\mathrm{A}}(n,s) = T_{\mathrm{B}}(n,s)$ in (3.18)).

5 Tripartitioning schemes

While bipartitioning schemes divide the input keys into $\leq v$ and $\geq v$, tripartitioning schemes divide the keys into $< v$, $= v$ and $> v$. We now give ternary versions of the schemes of §2, using the following notation for vector swaps (cf. [BeM93]).

A vector swap denoted by $x[a\colon b] \leftrightarrow x[b+1\colon c]$ means that the first $d := \min(b+1-a, c-b)$ elements of array $x[a\colon c]$ are exchanged with its last d elements in arbitrary order if $d > 0$; e.g., we may exchange $x_{a+i} \leftrightarrow x_{c-i}$ for $0 \leq i < d$, or $x_{a+i} \leftrightarrow x_{c-d+1+i}$ for $0 \leq i < d$.

5.1 Safeguarded ternary partition

Our ternary version of scheme A employs the following "strict" analogs of (2.2)–(2.4):

$$\begin{array}{|c|c|c|c|c|}\hline x=v & x<v & ? & x>v & x=v \\ \hline \multicolumn{1}{l}{l} & \multicolumn{1}{l}{p} & \multicolumn{1}{r}{i\ \ j} & \multicolumn{1}{c}{q} & \multicolumn{1}{r}{r} \end{array}, \tag{5.1}$$

$$\begin{array}{|c|c|c|c|c|}\hline x=v & x<v & x=v & x>v & x=v \\ \hline \multicolumn{1}{l}{l} & \multicolumn{1}{l}{p\ \ j} & \multicolumn{1}{c}{} & \multicolumn{1}{l}{i\ \ q} & \multicolumn{1}{r}{r} \end{array}, \tag{5.2}$$

$$\begin{array}{|c|c|c|}\hline x<v & x=v & x>v \\ \hline \multicolumn{1}{l}{l} & \multicolumn{1}{c}{a\ \ \ b} & \multicolumn{1}{r}{r} \end{array}. \tag{5.3}$$

Scheme E (Safeguarded ternary partition).

E1. [Initialize.] Set $i := l$, $p := i+1$, $j := r$ and $q := j-1$. If $v > x_j$, exchange $x_i \leftrightarrow x_j$ and set $p := i$; else if $v < x_j$, set $q := j$.

E2. [Increase i until $x_i \geq v$.] Increase i by 1; then if $x_i < v$, repeat this step.

E3. [Decrease j until $x_j \leq v$.] Decrease j by 1; then if $x_j > v$, repeat this step.

E4. [Exchange.] (Here $x_j \leq v \leq x_i$.) If $i < j$, exchange $x_i \leftrightarrow x_j$; then if $x_i = v$, exchange $x_i \leftrightarrow x_p$ and increase p by 1; if $x_j = v$, exchange $x_j \leftrightarrow x_q$ and decrease q by 1; return to E2. If $i = j$ (so that $x_i = x_j = v$), increase i by 1 and decrease j by 1.

E5. [Cleanup.] Set $a := l + j - p + 1$ and $b := r - q + i - 1$. Exchange $x[l\colon p-1] \leftrightarrow x[p\colon j]$ and $x[i\colon q] \leftrightarrow x[q+1\colon r]$.

Similarly to scheme A, scheme E makes n or $n+1$ key comparisons, and at most $n/2$ index comparisons at E4. Let $n_<$, $n_=$, $n_>$ denote the numbers of low, equal and high keys (here $j - p + 1$, $b - a + 1$, $q - i + 1$). Step E4 makes at most $n/2 - 1$ "usual" swaps $x_i \leftrightarrow x_j$, and $n_= - 1$ or $n_= - 2$ "equal" swaps when $x_i = v$ or $x_j = v$. Step E5 makes $\min\{p - l, n_<\} + \min\{r - q, n_>\}$ swaps; in particular, at most $\min\{n_=, n_< + n_>\}$ swaps.

5.2 Single-index controlled ternary partition

Our ternary version of scheme B also employs the arrangements (5.1)–(5.2).

Scheme F (Single-index controlled ternary partition).

F1. [Initialize.] Set $i := l$, $p := i + 1$, $j := r + 1$ and $q := j - 1$.

F2. [Increase i until $x_i \geq v$.] Increase i by 1; then if $i \leq r$ and $x_i < v$, repeat this step.

F3. [Decrease j until $x_j \leq v$.] Decrease j by 1; then if $x_j > v$, repeat this step.

F4. [Exchange.] (Here $x_j \leq v \leq x_i$.) If $i < j$, exchange $x_i \leftrightarrow x_j$; then if $x_i = v$, exchange $x_i \leftrightarrow x_p$ and increase p by 1; if $x_j = v$, exchange $x_j \leftrightarrow x_q$ and decrease q by 1; return to F2. If $i = j$ (so that $x_i = x_j = v$), increase i by 1 and decrease j by 1.

F5. [Cleanup.] Set $a := l + j - p + 1$ and $b := r - q + i - 1$. Exchange $x[l\colon p-1] \leftrightarrow x[p\colon j]$ and $x[i\colon q] \leftrightarrow x[q+1\colon r]$.

The comparison and swap counts of scheme F are similar to those of scheme E; in particular, step F5 makes $\min\{p - l, n_<\} + \min\{r - q, n_>\}$ swaps, where $p - l + r - q = n_=$ or $n_= - 1$. In contrast, a similar scheme of Sedgewick [Sed98, Chap. 7, quicksort] swaps all the $n_=$ equal keys in its last step. More importantly, Sedgewick's scheme needn't produce *true* ternary partitions (e.g., for $x = [0, 1, 0]$ and $v = 0$, it doesn't change the array).

5.3 Double-index controlled ternary partition

We now present our modification of the ternary scheme of [BeM93], described also in [BeS97, Prog. 1] and [Knu98, Ex. 5.2.2–41]. It employs the loop invariant (5.1), and the cross-over arrangement (5.2) with $j = i - 1$ for the swaps leading to the partition (5.3).

Scheme G (Double-index controlled ternary partition).

G1. [Initialize.] Set $i := p := l + 1$ and $j := q := r$.

G2. [Increase i until $x_i > v$.] If $i \leq j$ and $x_i < v$, increase i by 1 and repeat this step. If $i \leq j$ and $x_i = v$, exchange $x_p \leftrightarrow x_i$, increase p and i by 1, and repeat this step.

G3. [Decrease j until $x_j < v$.] If $i < j$ and $x_j > v$, decrease j by 1 and repeat this step. If $i < j$ and $x_j = v$, exchange $x_j \leftrightarrow x_q$, decrease j and q by 1, and repeat this step. If $i \geq j$, set $j := i - 1$ and go to G5.

G4. [Exchange.] Exchange $x_i \leftrightarrow x_j$, increase i by 1, decrease j by 1, and return to G2.

G5. [Cleanup.] Set $a := l + i - p$ and $b := r - q + j$. Swap $x[l\colon p-1] \leftrightarrow x[p\colon j]$ and $x[i\colon q] \leftrightarrow x[q+1\colon r]$.

Steps G2 and G3 make $n_= - 1$ swaps, step G4 at most $\min\{n_<, n_>\} \leq (n-1)/2$ swaps, and step G5 takes $\min\{p-l, n_<\} + \min\{r-q, n_>\} \leq \min\{n_=, n_< + n_>\}$ swaps.

Scheme G makes $n-1$ comparisons, whereas the versions of [BeM93, Progs. 6 and 7], [BeS97, §5], [Knu98, Ex. 5.2.2–41] make one spurious comparison when $i = j$ at step G3. These versions correspond to replacing step G3 by

G3'. [Decrease j until $x_j < v$.] If $i \leq j$ and $x_j > v$, decrease j by 1 and repeat this step. If $i \leq j$ and $x_j = v$, exchange $x_j \leftrightarrow x_q$, decrease j and q by 1, and repeat this step. If $i \geq j$, go to G5.

Except for making a spurious comparison when $i = j$, step G3' acts like G3: If $i = j$, then, since $x_i > v$ by G2, they exit to G5 with $j = i - 1$, whereas if $i > j$, then the general invariant $i \leq j + 1$ yields $i = j + 1$, and G3 maintains this equality.

Lemma 5.1. *Let $c \in \{0, 1\}$ be the number of spurious comparisons made by scheme G using step G3'. If the keys are distinct and in random order, then $\mathrm{E}[c|j_v = j] = 1 - j/(n-1)$ for $0 \leq j < n$, and $\mathrm{E}c = 1 - \mathrm{E}j_v/(n-1)$, where j_v is the number of keys $< v$. In particular, $\mathrm{E}c = 1/2$ when the pivot v is the median-of-s (for odd $s \geq 1$) or the ninther (cf. §3.3); in these cases scheme G with step G3' makes on average $n - 1/2$ comparisons.*

Proof. For distinct keys, the final $i = a + 1$ and $j = a$ at step G5. If $c = 1$, then $i = j$ and $x_i > v$ at G3' yield $i = a + 1 \leq n$ and $x_{a+1} > v$. Conversely, suppose $a < n$ and $x_{a+1} > v$ on input. If x_{a+1} were compared to v first at G3' for $j = a + 1 > i$, G3' would set $j = a$ and exit to G5 (since G4 would decrease j below a) with $i \leq a$, a contradiction; hence x_{a+1} must be compared to v first at G2 for $i = a + 1 \leq j$, and again at G3'. Thus $c = 1$ iff $a < n$ and $x_{a+1} > v$ on input. Consequently, for $j_v := a - 1 = j < n - 1$, $\mathrm{E}[c|j_v = j] = \Pr[x_{a+1} > v|j_v = j] = (n-1-j)/(n-1)$ since there are $n - 1 - j$ high keys in random order, and $\mathrm{E}[c|j_v = n-1] = 0$; the rest is straighforward. □

5.4 Lomuto's ternary partition

Our ternary extension of scheme D employs the following "strict" version of (2.5):

$$\begin{array}{c} \begin{array}{|c|c|c|c|} \hline x = v & x < v & x > v & ? \\ \hline \end{array} \\ l \quad \bar{p} \qquad\quad p \qquad\qquad\quad i \quad r \end{array} \longrightarrow \begin{array}{c} \begin{array}{|c|c|c|} \hline x = v & x < v & x > v \\ \hline \end{array} \\ l \quad \bar{p} \qquad\quad p \qquad\qquad r \end{array} . \tag{5.4}$$

Scheme H (Lomuto's ternary partition).

H1. [Initialize.] Set $i := l + 1$ and $p := \bar{p} := l$.

H2. [Check if done.] If $i > r$, go to H4.

H3. [Exchange if necessary.] If $x_i < v$, increase p by 1 and exchange $x_p \leftrightarrow x_i$. If $x_i = v$, increase $\bar{p}$ and p by 1 and exchange $x_p \leftrightarrow x_i$ and $x_{\bar{p}} \leftrightarrow x_p$. Increase i by 1 and return to H2.

H4. [Cleanup.] Set $a := l + p - \bar{p}$ and $b := p$. Exchange $x[l{:}\bar{p}] \leftrightarrow x[\bar{p}+1{:}p]$.

Scheme H makes $n_< + 2(n_= - 1) + \min\{n_=, n_<\}$ swaps. Using the arrangements

$$\begin{array}{|c|c|c|c|}\hline x<v & x=v & x>v & ? \\ \hline \multicolumn{4}{c}{l\ \ \bar{p}\qquad\quad p\qquad\qquad i\ \ r}\end{array} \longrightarrow \begin{array}{|c|c|c|}\hline x<v & x=v & x>v \\ \hline \multicolumn{3}{c}{l\ \ \bar{p}\qquad\quad p\qquad\quad r}\end{array}, \tag{5.5}$$

with obvious modifications, scheme H would make $n_= - 1 + 2n_<$ swaps.

5.5 Comparison of binary and ternary schemes

When the keys are distinct, the binary schemes A, B, C, D are *equivalent* to their ternary versions E, F, G, H in the sense that respective pairs of schemes (e.g., A and E) produce identical partitions, making the same sequences of comparisons and swaps. Hence our results of §3 extend to the ternary schemes by replacing A, B, C, D with E, F, G, H, respectively. Since the overheads of the ternary schemes are relatively small, consisting mostly of additional tests for equal keys, the ternary schemes should run almost as fast as their binary counterparts in the case of distinct keys.

Let us highlight some differences when equal keys occur. Although schemes A and E stop the scanning pointers i and j on the same keys, step A4 simply swaps each key to the other side, whereas step E4 additionally swaps equals to the ends. Schemes B and F behave similarly. However, in contrast with scheme C, scheme G never swaps equals to the other side. For instance, when all the keys are equal, scheme E makes $\lfloor n/2 - 1 \rfloor$ usual swaps and $2\lfloor n/2 - 1 \rfloor$ vacuous swaps, scheme F makes $\lfloor (n-1)/2 \rfloor$ usual swaps and $2\lfloor (n-1)/2 \rfloor$ vacuous swaps, scheme G makes just $n - 1$ vacuous swaps, and scheme H makes $2(n-1)$ vacuous swaps.

5.6 Preventing vacuous swaps of equal keys

Steps G2 and G3 of scheme G have two drawbacks: they make vacuous swaps when $i = p$ and $j = q$, and they need the tests "$i \le j$" and "$i < j$". These drawbacks are eliminated in the following two-phase scheme, which runs first a special version of scheme G that doesn't make vacuous swaps until it finds two keys $x_i < v < x_j$. Afterwards no vacuous swaps occur (because $p < i$, $j < q$) and the pointer tests are unnecessary (since $x_j > v$ stops the i-loop, and $x_{i-1} < v$ stops the j-loop).

Scheme I (Hybrid ternary partition).

I1. [Initialize.] Set $i := l + 1$ and $j := q := r$.

I2. [Increase i until $x_i \ne v$.] If $i \le j$ and $x_i = v$, increase i by 1 and repeat this step. Set $p := i$. If $i = j$, set $i := j + 1$ if $x_i < v$, $j := i - 1$ otherwise. If $i \ge j$, go to I12.

I3. [Decrease j until $x_j \neq v$.] If $i < j$ and $x_j = v$, decrease j by 1 and repeat this step. Set $q := j$. If $i = j$, set $i := j + 1$ if $x_i < v$, $j := i - 1$ otherwise, and go to I12.

I4. [Decide which steps to skip.] If $x_i < v$ and $x_j < v$, go to I5. If $x_i > v$ and $x_j > v$, go to I6. If $x_i > v$ and $x_j < v$, go to I7. If $x_i < v$ and $x_j > v$, go to I8.

I5. [Increase i until $x_i > v$.] Increase i by 1. If $i < j$ and $x_i < v$, repeat this step. If $i < j$ and $x_i = v$, exchange $x_p \leftrightarrow x_i$, increase p by 1, and repeat this step. (At this point, $x_j < v$.) If $i < j$, go to I7. Set $i := j + 1$ and go to I12.

I6. [Decrease j until $x_j < v$.] Decrease j by 1. If $i < j$ and $x_j > v$, repeat this step. If $i < j$ and $x_j = v$, exchange $x_j \leftrightarrow x_q$, decrease q by 1, and repeat this step. (At this point, $x_i > v$.) If $i = j$, set $j := i - 1$ and go to I12.

I7. [Exchange.] (At this point, $i < j$ and $x_i > v > x_j$.) Exchange $x_i \leftrightarrow x_j$.

I8. [End of first stage.] (At this point, $x_i < v < x_j$ and $p \leq i < j \leq q$.)

I9. [Increase i until $x_i > v$.] Increase i by 1. If $x_i < v$, repeat this step. If $x_i = v$, exchange $x_p \leftrightarrow x_i$, increase p by 1, and repeat this step.

I10. [Decrease j until $x_j < v$.] Decrease j by 1. If $x_j > v$, repeat this step. If $x_j = v$, exchange $x_j \leftrightarrow x_q$, decrease q by 1, and repeat this step.

I11. [Exchange.] If $i < j$, exchange $x_i \leftrightarrow x_j$ and return to I9.

I12. [Cleanup.] Set $a := l + i - p$ and $b := r - q + j$. Exchange $x[l\colon p - 1] \leftrightarrow x[p\colon j]$ and $x[i\colon q] \leftrightarrow x[q + 1\colon r]$.

Scheme I makes $n + 1$ comparisons, or just $n - 1$ if it finishes in the first stage before reaching step I9. The two extraneous comparisons can be eliminated by keeping the strategy of scheme G in the following modification.

Scheme J (Extended double-index controlled ternary partition).
Use scheme I with steps I8 through I11 replaced by the following steps.

I8. [End of first stage.] Increase i by 1 and decrease j by 1.

I9. [Increase i until $x_i > v$.] If $i \leq j$ and $x_i < v$, increase i by 1 and repeat this step. If $i \leq j$ and $x_i = v$, exchange $x_p \leftrightarrow x_i$, increase p and i by 1, and repeat this step.

I10. [Decrease j until $x_j < v$.] If $i < j$ and $x_j > v$, decrease j by 1 and repeat this step. If $i < j$ and $x_j = v$, exchange $x_j \leftrightarrow x_q$, decrease j and q by 1, and repeat this step. If $i \geq j$, set $j := i - 1$ and go to I12.

I11. [Exchange.] Exchange $x_i \leftrightarrow x_j$, increase i by 1, decrease j by 1, and return to I9.

Schemes I and J are *equivalent* in the sense of producing identical partitions via the same sequences of swaps. Further, barring vacuous swaps, scheme G is equivalent to schemes I and J in the following cases: (a) all keys are equal; (b) $x_r \neq v$ (e.g., the keys are distinct); (c) there is at least one high key $> v$. In the remaining *degenerate* case where the keys aren't equal, $x_r = v$ and there are no high keys, scheme G produces $i = r + 1$ and $j = r$ on the first pass, whereas step I3 finds $j < r$, and either I3 or I5 produce $i = j + 1$ (i.e., scheme G swaps $r - j$ more equal keys to the left end).

If the first stage of schemes I and J is implemented by a more straightforward adaptation of scheme G, we obtain the following variants.

Scheme K (Alternative hybrid ternary partition).
Use scheme I with steps I2 through I6 replaced by the following steps.

I2. [Increase i until $x_i \neq v$.] If $i \leq j$ and $x_i = v$, increase i by 1 and repeat this step. Set $p := i$. If $i \leq j$ and $x_i < v$, increase i by 1 and go to I3; otherwise go to I4.

I3. [Increase i until $x_i > v$.] If $i \leq j$ and $x_i < v$, increase i by 1 and repeat this step. If $i \leq j$ and $x_i = v$, exchange $x_p \leftrightarrow x_i$, increase p and i by 1, and repeat this step.

I4. [Decrease j until $x_j \neq v$.] If $i < j$ and $x_j = v$, decrease j by 1 and repeat this step. Set $q := j$. If $i < j$ and $x_j > v$, decrease j by 1 and go to I5. If $i < j$ and $x_j < v$, go to I7. Set $j := i - 1$ and go to I12.

I5. [Decrease j until $x_j < v$.] If $i < j$ and $x_j > v$, decrease j by 1 and repeat this step. If $i < j$ and $x_j = v$, exchange $x_j \leftrightarrow x_q$, decrease j and q by 1, and repeat this step. If $i \geq j$, set $j := i - 1$ and go to I12.

Scheme L (Two-stage double-index controlled ternary partition).
Use scheme I with steps I2 through I6 replaced by steps I2 through I5 of scheme K, and steps I8 through I11 replaced by steps I8 through I11 of scheme J.

In other words, scheme L is obtained from scheme G by using special versions of steps G2 and G3 on the first pass, with each step split into two substeps to avoid vacuous swaps.

Except for avoiding vacuous swaps, schemes K and L are equivalent to scheme G. Hence schemes G, I, J, K and L are equivalent except for the degenerate case discussed after scheme J; in this case, schemes I and J swap fewer equal keys than schemes K and L. Another significant difference between schemes I and K is that scheme I may be quicker in reaching the second stage where the tests "$i \leq j$" and "$i < j$" aren't needed. (In fact scheme I reaches step I8 faster than scheme K iff $x_i < v < x_j$ occurs at step I4 of scheme I; in the remaining three cases of I4 both schemes act equivalently.)

5.7 Using sample elements in tripartitioning

In parallel with §4, we now show how to tune the ternary schemes when the pivot v is selected as the $(\hat{p}+1)$th element in a sample of size s, assuming $0 \leq \hat{p} < s \leq n$ and $\hat{q} := s - 1 - \hat{p} > 0$.

First, suppose that after pivot selection, we have the following arrangement:

$$\begin{array}{|c|c|c|c|c|}\hline x<v & x=v & \quad ? \quad & x=v & x>v \\ \hline \multicolumn{1}{c}{l} & \multicolumn{1}{c}{\bar{l}} & \multicolumn{1}{c}{p \quad q} & \multicolumn{1}{c}{\bar{r}} & \multicolumn{1}{c}{r} \end{array}, \tag{5.6}$$

with $p := l + \hat{p} + 1$, $q := r - \hat{q}$; then we only need to partition the array $x[p-1\colon q]$ of size $\bar{n} := n - s + 1$. The ternary schemes are modified as follows.

In step E1 of scheme E, set $i := p - 1$ and $j := q + 1$; in step E5 replace l, r by $\bar{l}$, $\bar{r}$. The same scheme results from scheme F after analogous changes and omitting the test "$i \leq r$"

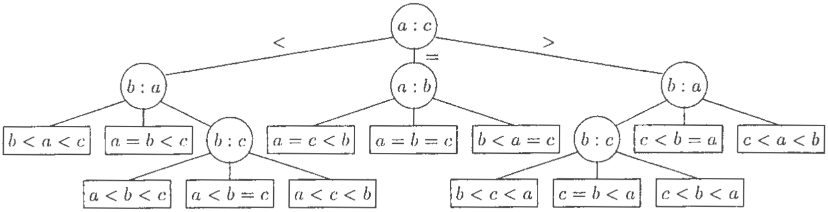

Figure 5.1: Decision tree for median of three

in F2. Similarly, in step G1 of scheme G, set $i := p$ and $j := q$; in step G5 replace l, r by $\bar{l}$, $\bar{r}$. Steps I1 and I11 of schemes I through L are modified in the same way. Finally, in step H1 of scheme H set $i := p$ and $p := \bar{p} := i - 1$; in step H2 replace r by q; in step H4 set $a := \bar{l} + p - \bar{p}$, $b := p - q + \bar{r}$ and exchange $x[\bar{l}\colon \bar{p}] \leftrightarrow x[\bar{p} + 1\colon p]$ and $x[p + 1\colon q] \leftrightarrow x[q + 1\colon \bar{r}]$.

When the keys are distinct, we have $\bar{l} = l + \hat{p}$, $p = \bar{l} + 1$ and $q = \bar{r} = r - \hat{q}$ in (5.6), so that schemes E, F, G, H are equivalent to schemes A, B, C, D as modified in §4 (where p, q correspond to the current $\hat{p}$, $\hat{q}$).

For the median-of-3 selection ($\hat{p} = \hat{q} = 1$, $p = l + 2$, $q = r - 1$), we may rearrange the sample keys x_l, x_{l+1}, x_r and find $\bar{l}$, $\bar{r}$ according to Figure 5.1. (For simplicity, as with Fig. 4.1, the left subtree may be used after exchanging $a \leftrightarrow c$ when $a > c$.)

As in §4, even if pivot selection doesn't rearrange the array except for placing the pivot in x_l, scheme E may be simplified by replacing step E1 with step F1; the same scheme is obtained from scheme F by omitting the test "$i \le r$" in F2.

6 Experimental results

6.1 Implemented algorithms

We now sketch the algorithms used in our experiments, starting with a nonrecursive version of quickselect that employs a random pivot and one of the ternary schemes of §5.

Algorithm 6.1 (QUICKSELECT(x, n, k) for selecting the kth smallest of $x[1\colon n]$).

Step 1 (*Initialize*). Set $l := 1$ and $r := n$.

Step 2 (*Handle small file*). If $l < r$, go to Step 3. If $l > r$, set $k_- := r + 1$ and $k_+ := l - 1$. If $l = r$, set $k_- := k_+ := k$. Return.

Step 3 (*Select pivot*). Pick a random integer $i \in [l, r]$, swap $x_l \leftrightarrow x_i$ and set $v := x_l$.

Step 4 (*Partition*). Partition the array $x[l\colon r]$ to produce the arrangement (5.3).

Step 5 (*Update bounds*). If $a \le k$, set $l := b + 1$. If $k \le b$, set $r := a - 1$. Go to Step 2.

Steps 2 and 5 ensure that on exit $x[1\colon k_- - 1] < x[k_-\colon k_+] < x[k_+ + 1\colon n]$, $k_- \le k \le k_+$.

The median-of-3 version works as follows. If $l = r - 1$ at Step 2, we swap $x_l \leftrightarrow x_r$ if $x_l > x_r$, set $k_- := l$ and $k_+ := r$ if $x_l = x_r$, $k_- := k_+ := k$ otherwise, and return. At

Step 3, we swap x_{l+1}, x_r with random keys in $x[l+1\colon r]$ and $x[l+2\colon r]$, respectively. After sorting the sample keys x_l, x_{l+1}, x_r and finding $\bar{l}$, $\bar{r}$ for (5.6) according to Fig. 5.1, we set $v := x_{l+1}$. Then Step 4 uses one of the modified ternary schemes of §5.7.

When a binary scheme is employed, we omit k_- and k_+, use Fig. 4.1 instead of Fig. 5.1, and the modified schemes of §4 with $\bar{l} := l+1$, $\bar{r} := r-1$ for the median-of-3.

Our implementations of QUICKSELECT were programmed in Fortran 77 and run on a notebook PC (Pentium 4M 2 GHz, 768 MB RAM) under MS Windows XP. We used a double precision input array $x[1\colon n]$, in-line comparisons and swaps; future work should test tuned comparison and swap functions for other data types (cf. [BeM93]).

6.2 Testing examples

We used minor modifications of the input sequences of [Val00], defined as follows:

random A random permutation of the integers 1 through n.

mod-m A random permutation of the sequence i mod m, $i = 1\colon n$, called *binary* (*ternary*, *quadrary*, *quintary*) when $m = 2$ (3, 4, 5, respectively).

sorted The integers 1 through n in increasing order.

rotated A sorted sequence rotated left once; i.e., $(2, 3, \ldots, n, 1)$.

organpipe The integers $(1, 2, \ldots, n/2, n/2, \ldots, 2, 1)$.

m3killer Musser's "median-of-3 killer" sequence with $n = 4j$ and $k = n/2$:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & \ldots & k-2 & k-1 & k & k+1 & \ldots & 2k-2 & 2k-1 & 2k \\ 1 & k+1 & 3 & k+3 & \ldots & 2k-3 & k-1 & 2 & 4 & \ldots & 2k-2 & 2k-1 & 2k \end{pmatrix}.$$

twofaced Obtained by randomly permuting the elements of an m3killer sequence in positions $4\lfloor \log_2 n \rfloor$ through $n/2-1$ and $n/2 + 4\lfloor \log_2 n \rfloor - 1$ through $n-2$.

For each input sequence, its (lower) median element was selected for $k := \lceil n/2 \rceil$.

These input sequences were designed to test the performance of selection algorithms under a range of conditions. In particular, the binary sequences represent inputs containing many duplicates [Sed77]. The rotated and organpipe sequences are difficult for many implementations of quickselect. The m3killer and twofaced sequences are hard for implementations with median-of-3 pivots (their original versions [Mus97] were modified to become difficult when the middle element comes from position k instead of $k+1$).

6.3 Computational results

We varied the input size n from 50,000 to 16,000,000. For the random, mod-m and twofaced sequences, for each input size, 20 instances were randomly generated; for the deterministic sequences, 20 runs were made to measure the solution time.

Table 6.1 summarizes the performance of four schemes used in QUICKSELECT with median-of-3. The average, maximum and minimum solution times are in milliseconds (in

Table 6.1: Performance of schemes A, E, G, I with median-of-3.

Scheme	Sequence	Size	Time [msec]			Comparisons [n]			P_{avg}	S_{avg}	S^0_{avg}	S_{avg}
		n	avg	max	min	avg	max	min	[$\ln n$]	[n]	[n]	[C_{avg}]
A	random	8M	252	360	170	2.59	3.96	1.78	1.64	0.55	0.00	0.21
		16M	494	641	371	2.57	3.46	1.93	1.57	0.53	0.00	0.21
	organpipe	8M	173	250	111	2.64	4.10	1.77	1.53	0.57	0.00	0.22
		16M	355	460	270	2.61	3.49	1.94	1.62	0.60	0.00	0.23
	binary	8M	254	271	250	2.73	2.92	2.68	1.86	1.00	0.00	0.37
		16M	506	521	500	2.70	2.79	2.68	1.87	1.00	0.00	0.37
	ternary	8M	246	321	171	2.44	3.27	1.75	1.33	0.82	0.00	0.34
		16M	452	620	360	2.22	3.11	1.75	1.29	0.76	0.00	0.34
	quadrary	8M	277	340	230	2.78	3.44	2.26	1.83	0.86	0.00	0.31
		16M	537	671	460	2.65	3.37	2.26	1.85	0.84	0.00	0.32
	quintary	8M	231	350	180	2.31	3.56	1.85	1.34	0.69	0.00	0.30
		16M	486	671	330	2.44	3.49	1.67	1.36	0.71	0.00	0.29
E	random	8M	284	391	201	2.59	3.96	1.78	1.64	0.55	0.00	0.21
		16M	550	711	411	2.57	3.46	1.93	1.57	0.53	0.00	0.21
	organpipe	8M	232	321	120	2.73	5.27	1.84	1.54	0.57	0.00	0.21
		16M	421	571	320	2.92	4.62	1.90	1.58	0.59	0.00	0.20
	binary	8M	205	231	170	1.28	1.50	1.00	0.10	1.41	0.61	1.11
		16M	381	471	350	1.13	1.50	1.00	0.08	1.19	0.46	1.06
	ternary	8M	259	281	240	1.47	2.00	1.00	0.12	1.37	0.37	0.93
		16M	505	590	480	1.37	2.00	1.00	0.10	1.25	0.28	0.91
	quadrary	8M	262	331	210	1.60	2.50	1.00	0.12	1.33	0.28	0.83
		16M	559	661	410	1.66	2.25	1.00	0.13	1.35	0.31	0.81
	quintary	8M	283	370	210	1.52	2.40	1.00	0.13	1.14	0.14	0.75
		16M	582	731	420	1.55	2.40	1.00	0.14	1.13	0.14	0.73
G	random	8M	301	411	210	2.59	3.96	1.78	1.64	0.55	0.00	0.21
		16M	587	761	430	2.57	3.46	1.93	1.57	0.53	0.00	0.21
	organpipe	8M	186	250	110	2.88	4.20	1.91	1.55	0.61	0.00	0.21
		16M	378	511	270	2.77	3.93	1.97	1.59	0.59	0.00	0.21
	binary	8M	293	331	250	1.27	1.50	1.00	0.10	1.27	0.27	1.00
		16M	549	671	500	1.12	1.50	1.00	0.08	1.12	0.13	1.00
	ternary	8M	340	420	250	1.47	2.00	1.00	0.12	1.21	0.10	0.82
		16M	646	811	501	1.53	2.00	1.00	0.11	1.26	0.10	0.82
	quadrary	8M	311	450	220	1.42	2.25	1.00	0.12	1.02	0.07	0.72
		16M	665	972	440	1.55	2.50	1.00	0.13	1.13	0.09	0.73
	quintary	8M	319	451	220	1.47	2.00	1.00	0.13	0.96	0.07	0.65
		16M	644	1021	440	1.61	2.80	1.00	0.13	0.97	0.04	0.60
I	random	8M	275	381	190	2.59	3.96	1.78	1.64	0.55	0.00	0.21
		16M	536	681	391	2.57	3.46	1.93	1.57	0.53	0.00	0.21
	organpipe	8M	183	240	110	2.88	4.20	1.91	1.55	0.61	0.00	0.21
		16M	357	461	260	2.77	3.93	1.97	1.59	0.59	0.00	0.21
	binary	8M	245	261	230	1.27	1.50	1.00	0.10	1.00	0.00	0.78
		16M	500	530	480	1.12	1.50	1.00	0.08	1.00	0.00	0.89
	ternary	8M	323	391	230	1.47	2.00	1.00	0.12	1.11	0.00	0.76
		16M	620	761	470	1.53	2.00	1.00	0.11	1.16	0.00	0.76
	quadrary	8M	292	440	200	1.43	2.25	1.00	0.12	0.95	0.00	0.66
		16M	630	922	420	1.55	2.50	1.00	0.13	1.04	0.00	0.67
	quintary	8M	297	431	200	1.47	2.00	1.00	0.13	0.89	0.00	0.60
		16M	614	1042	411	1.61	2.80	1.00	0.13	0.93	0.00	0.58

general, they grow linearly with n, and can't be measured accurately for small inputs; hence only large inputs are included, with 1M $:= 10^6$). The comparison counts are in multiples of n; e.g., column seven gives C_{avg}/n, where C_{avg} is the average number of comparisons made over all instances. Further, P_{avg} is the average number of partitions in units of $\ln n$, S_{avg} and S^0_{avg} are the average numbers of all swaps and of vacuous swaps in units of n, and the final column gives the average number of swaps per comparison. Note that for random inputs with distinct keys, quickselect with median-of-3 takes on average $2.75n + o(n)$ comparisons and $\frac{12}{7} \ln n + o(n)$ partitions [Grü99, KMP97], and thus about $0.55n$ swaps when there are $1/5$ swaps per comparison; e.g., for schemes A, E and G.

For each scheme (and others not included in Tab. 6.1), the results for the twofaced and m3killer inputs were similar to those for the random and organpipe inputs, respectively. The sorted and rotated inputs were solved about twice faster than the random inputs.

Recall that in tuned versions, scheme B coincides with A and scheme F with E.

The run times of schemes C and J were similar to those of schemes A and I, respectively; in other words, the inclusion of pointer tests in the key comparison loops didn't result in significant slowdowns. Also their comparison and swap counts were similar.

Due to additional tests for equal keys, the ternary schemes were slower than their binary counterparts on the inputs with distinct keys. Yet the slowdowns were quite mild (e.g., about ten percent for scheme E vs. A) and could be considered a fair price for being able to identify all keys equal to the selected one. On the inputs with multiple equal keys, the numbers of comparisons made by the binary schemes A and C were similar to those made on the random inputs, but the numbers of swaps increased up to n. In contrast, the ternary schemes E and G took significantly fewer comparisons and more swaps. Scheme E produced the largest numbers of swaps, but was still faster than schemes G and J, whereas scheme J was noticeably faster than scheme G due to the elimination of vacuous swaps.

On the inputs with distinct keys, Lomuto's scheme D was about sixty percent slower than scheme A, making about half as many swaps as comparisons (cf. §§3.3.1 and 4). On the inputs with multiple equal keys, scheme D was really bad: once the current array $x[l\colon r]$ contains only keys equal to the kth smallest, each partition removes two keys, so the running time may be quadratic in the number of equal keys. For instance, on a binary input with $k = n/2$, at least $n(n + 20)/16 - 2$ comparisons are used (if the first $v = 1$, we get $l = 1$, $r = k$, and then l increases by 2 while $r = k$; otherwise the cost is greater).

Our results were similar while using the classic random pivot instead of the median-of-3. Then, for random inputs with distinct keys, quickselect takes on average $2(1+\ln 2)n+o(n)$ comparisons [Knu98, Ex. 5.2.2–32], and thus about $0.564n$ swaps when there are $1/6$ swaps per comparison. Hence, not suprisingly, the running times and comparison counts on the inputs with distinct keys increased by between 14 and 20 percent, but all the schemes had essentially the same relative merits and drawbacks as in the median-of-3 case above.

References

[BeM93] J. L. Bentley and M. D. McIlroy, *Engineering a sort function*, Software–Practice and Experience **23** (1993) 1249–1265.

[Ben98] J. L. Bentley, *Programming Pearls*, Addison-Wesley, Reading, MA, 1998.

[BeS97] J. L. Bentley and R. Sedgewick, *Fast algorithms for sorting and searching strings*, in Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97), SIAM, Philadelphia, 1997, pp. 360–369.

[BFP+72] M. R. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest and R. E. Tarjan, *Time bounds for selection*, J. Comput. System Sci. **7** (1972) 448–461.

[CHT02] H.-H. Chern, H.-K. Hwang and T.-H. Tsai, *An asymptotic theory for Cauchy-Euler differential equations with applications to the analysis of algorithms*, J. Algorithms **44** (2002) 177–225.

[Chv02] V. Chvátal, *Average-case analysis of quicksort*, Lecture notes, Dept. of Computer Science, Rutgers Univ., New Brunswick, 2002.

[CLR90] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, second ed., MIT Press, Cambridge, MA, 2001.

[Dij76] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Dur03] M. Durand, *Asymptotic analysis of an optimized quicksort algorithm*, Information Proc. Letters **85** (2003) 73–77.

[FlR75] R. W. Floyd and R. L. Rivest, *Expected time bounds for selection*, Comm. ACM **18** (1975) 165–172.

[Grü99] R. Grübel, *On the median-of-k version of Hoare's selection algorithm*, Theor. Inform. Appl. **33** (1999) 177–192.

[Hoa61a] C. A. R. Hoare, *Algorithm 63:* PARTITION, Comm. ACM **4** (1961) 321–322.

[Hoa61b] ———, *Algorithm 65:* FIND, Comm. ACM **4** (1961) 321–322.

[Hoa62] ———, *Quicksort*, Computer J. **5** (1962) 10–15.

[HwT02] H.-K. Hwang and T.-H. Tsai, *Quickselect and Dickman function*, Combinatorics, Probability and Computing **11** (2002) 353–371.

[KMP97] P. Kirschenhofer, C. Martínez and H. Prodinger, *Analysis of Hoare's* FIND *algorithm with median-of-three partition*, Random Stuctures and Algorithms **10** (1997) 143–156.

[Knu97] D. E. Knuth, *The Art of Computer Programming. Volume I: Fundamental Algorithms*, third ed., Addison-Wesley, Reading, MA, 1997.

[Knu98] ———, *The Art of Computer Programming. Volume III: Sorting and Searching*, second ed., Addison-Wesley, Reading, MA, 1998.

[MaR01] C. Martínez and S. Roura, *Optimal sampling strategies in quicksort and quickselect*, SIAM J. Comput. **31** (2001) 683–705.

[Mus97] D. R. Musser, *Introspective sorting and selection algorithms*, Software–Practice and Experience **27** (1997) 983–993.

[Pre00] B. R. Preiss, *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*, John Wiley & Sons, Chichester, 2000.

[Sed77] R. Sedgewick, *Quicksort with equal keys*, SIAM J. Comput. **6** (1977) 240–287.

[Sed78] ———, *Implementing quicksort programs*, Comm. ACM **21** (1978) 847–857.

[Sed98] ———, *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching*, third ed., Addison-Wesley, Reading, MA, 1998.

[Val00] J. D. Valois, *Introspective sorting and selection revisited*, Software–Practice and Experience **30** (2000) 617–638.

[Weg85] L. M. Wegner, *Quicksort for equal keys*, IEEE Trans. Computers **C-34** (1985) 362–367.