On the Efficiency of Algorithm SELECT by Floyd and Rivest

by

James R. Bitner
Department of Computer Science
University of Texas
Austin Texas, 78712

TR 183

October, 1981

ABSTRACT

We study algorithm SELECT by Floyd and Rivest for finding the Kth largest out of n elements. The dominant cost of this algorithm is a 3-Way Partitioning step where an array of "small", "medium", and "large" elements is permuted so that all small elements precede all medium elements which precede all large elements. Floyd and Rivest could not find an efficient and direct method to accomplish this 3-Way Partitioning and used a (supposedly inefficient) two step method. We show their two step method is not inefficient, but in fact, optimal and develop a simple algorithm for directly solving the 3-Way Partitioning and prove it to be optimal.

1. INTRODUCTION

Algorithm SELECT by Floyd and Rivest [1] is the most practical algorithm known for finding the kth largest out of n elements. In the average case, $n + \min(k, n-k) + o(n)$ comparisons are required. This is better average case complexity than algorithm FIND by Hoare [4], and the algorithm is simpler to implement and more practical than that of Blum, et.al. [5]. (A simple and efficient implementation of the algorithm is given in [1]). Finally, a lower bound on the average complexity of the problem is proved [1] which is within 9% of the performance of algorithm SELECT.

The dominant cost of algorithm SELECT is a 3-way partitioning step (defined below). Floyd and Rivest could not find a method to accomplish this partitioning directly and used a two-step method we describe later in this section. They termed direct 3-Way Partitioning an "inherently inefficient" operation and stated that any reduction in the complexity of their (supposedly inefficient) partitioning phase would show up as a significant increase in the efficiency of the whole algorithm.

The purpose of this paper is to:

(1) Show the two-step partitioning method used by Floyd and Rivest is not inefficient, but in fact, optimal.

(2) Develop a direct algorithm for solving the 3-way partitioning problem which is also optimal.

We now discuss the Floyd-Rivest algorithm in sufficient detail to allow us to describe how the partitioning phase interfaces with the rest of the algorithm and the reasons for our probabilistic assumptions (given later). The algorithm is given a segment of an array A[first..last] and a number k. It permutes the elements of A such that:

$A[i] \leq A[k]$ for $first \leq i < k$ and
$A[i] \geq A[k]$ for $k < i \leq last$

Thus, it finds the element which would be in position A[k] if the array were sorted. This is somewhat more convenient than finding the kth smallest element in A[first]..A[last]. Let size(= last - first + 1) be the number of elements in the segment. The algorithm begins by choosing a sample of size s(size) centered about position k (s is a carefully chosen function, see below). Let corresp be the position in the sample corresponding to position k in the whole segment (i.e. a fraction k/size of the way across the sample). SELECT is called twice recursively on the sample to find the correct values for positions corresp - d(size) and corresp + d(size). These values are chosen as u and v. The algorithm then partitions the entire segment into regions of "small" (less than u), "medium" (between u and v), and "large" (greater than v) elements. Finally SELECT is called recursively on the region containing position k.

The algorithm is very fast because we assume A is filled with random numbers each independently chosen from a uniform distribution over [0,1] (or, equivalently, A is filled with a random permutation of n elements). The functions s and d are carefully chosen (see [1])

such that the size of the sample is o(n), and the size of the medium region after the partitioning is also o(n), yet with probability approaching 1 the element we want will be in the medium region.

The authors could not find an efficient algorithm to do 3-way partitioning, so they used the following partitioning strategy instead: First partition about u using the quicksort partitioning algorithm [4], then partition the set of elements greater than u about v. (Actually, the partitioning is done about v first if we expect there will be more elements above v than below u, i.e. $k > (last + first)/2$.)

We now formally define the three related problems we will be studying.

The Selection Problem

Input: An array A[1..n] and an integer k with $1 \leq k \leq n$.

Output: A permutation of A such that

$$A[i] \leq A[k] \quad \text{for } 1 \leq i < k \text{ and}$$
$$A[i] \geq A[k] \quad \text{for } k < i \leq n$$

Analysis: Number of comparison and swaps in the average case where each of the n! input permutations is equally likely (or, equivalently where each element is independently chosen from a uniform distribution over [0,1]).

Defining the problem in this way allowed Floyd and Rivest to directly compare execution time with algorithm FIND. In addition, the Floyd-Rivest algorithm constructs this permutation as it finds the kth element, so it is obtained at no extra cost.

The dominant cost of algorithm SELECT is a 3-way partitioning step defined as follows:

The 3-Way Partitioning Problem

Input: An array and two values u and v $(u \leq v)$.

Output: A permutation of the array such that all elements less than u (called small elements) occur first, followed by all those between u and v (called medium elements) then followed by all those greater than v (called large elements). The order of the elements within these three segments is immaterial.

Analysis: Average number of swaps and comparisons where each element is chosen independently with probabilities P_S, P_M and P_L of being small, medium, and large respectively. Actually, P_S, P_M and P_L are functions of n, the size of the array, and $P_M(n) = o(n)$. We usually assume $P_S \leq P_L$.

This partitioning problem is very similar to the Dutch National Flag problem [2], which is defined as follows:

The Dutch National Flag Problem [2]

Input: A sequence of n pebbles. Each is either red, white, or blue.

Output: A permutation of the sequence such that all the red pebbles occur first followed by all the white pebbles then followed by all the blue pebbles.

Restrictions: We are restricted to using the following two primitive functions in accessing the sequence: buck(i) which gives the color of the ith pebble in the sequence, and swap(i,j), which interchanges the ith and jth pebbles. Buck is deemed a very expensive operation and hence may be applied only once to each pebble in the sequence. Also, the algorithm must operate using a constant amount of space, independent of the length of the sequence.

Analysis: The average number of swaps where each of the 3^n initial sequences is equally likely. (Or, equivalently, where each pebble has probability 1/3 of being either red, white, or blue.)

The only difference between the 3-Way Partitioning Problem and the Dutch National Flag Problem is in the assumptions about the probability of the different kinds of elements. This difference is significant; though a good algorithm for the Dutch National Flag Problem would correctly solve the selection problem, it might do so very inefficiently.

The paper is organized as follows: in Section 2 we state some pertinent results from [3], where we developed an asymptotically optimal algorithm to solve the Dutch National Flag Problem and proved a theorem which allows us to calculate the minimum number of swaps required to permute one sequence into another. We use these results in Section 3 to prove a lower bound on the number of comparisons and interchanges required to solve the 3-Way Partitioning Problem and use it to prove algorithm SELECT's partitioning strategy is optimal. Finally, in Section 4, we develop a new, direct algorithm for the 3-Way Partitioning Problem and prove it to be optimal.

2. APPLICABLE PREVIOUS RESULTS

In this section we state some results from [3] concerning the Dutch National Flag Problem which are applicable. In [3] we studied a generalized version of the problem where we are given a sequence of pebbles each with one of c different colors and must order the sequence such that all pebbles of color i precede all those of color i + 1 (for i = 1,...,c - 1). We developed a correspondence between sequences of pebbles and eulerian digraphs (see below for definition), and used it to derive a lower bound on the number of swaps required to order any sequence.

Definition: An eulerian digraph is a directed graph in which every vertex has its indegree equal to its outdegree. We allow a digraph to have multiple edges and self-loops. For an eulerian digraph G, let e(G) be the number of edges in G, and index(G) be e(G)-M(G) where M(G) is the number of cycles in a maximal decomposition of G into edge disjoint cycles. A cycle is a path in the digraph whose initial and final vertices are identical. It may pass through a vertex more than once.

To develop a correspondence between sequences of pebbles and eulerian digraphs, we first divide a sequence into "regions". If there are a total of x_i pebbles of color i in the sequence, let the first x_1 positions be region 1, the next x_2 in region 2, and so on. The digraph corresponding to this sequence has c vertices and is created by adding one edge from vertex i to vertex j for every pebble

of color i in region j. Note that a sequence is completely ordered if and only if for all i, all pebbles of color i are in region i. Hence, the digraph corresponding to the completely ordered sequence consists solely of self-loops.

Theorem 2.1 [3]: A digraph constructed from a sequence as described above is an eulerian digraph.

To order a sequence, given any decomposition of the corresponding digraph (call it G) into edge-disjoint cycles, we ignore the self loops and sequence through the remaining cycles in any order. If the current cycle is $v_{i_1}, v_{i_2}, \ldots, v_{i_K}$, there must be a pebble of color i_j in region i_{j+1} for $j = 1,\ldots,K-1$ and a pebble of color i_K in region i_1. Clearly $K - 1$ swaps can be used to put each of these K pebbles in the correct region. Also note that after processing all the cycles, the sequence will be ordered. If the decomposition has K cycles with the ith having length L_i, the total number of swaps done is $\sum_{i=1}^{K} [L_i - 1] = e(G) - K$. Clearly, this quantity is minimized by using a maximal decomposition.

Further, using the above procedure with a maximal decomposition uses the minimal number of swaps over all possible algorithms, not just those using this strategy. To prove this an "entropy argument" was used, where index(G) is the entropy function. (G is the digraph corresponding to the current sequence in the execution of some algorithm.) Index(G) must be decreased from its initial value down to zero. Proving that each swap decreases index(G) by at most one gives the following result:

Theorem 2.2 [3]: Given a sequence S_0, let G_0 be the corresponding digraph. Then at least index(G_0) swaps must be used in ordering S_0, and this lower bound is achievable.

To calculate index(G) we must find a maximal decomposition. Though finding a maximal decomposition for arbitrary c appears to be a hard problem, for "small" c we can use a simple, "greedy" algorithm, the shortest cycle first algorithm, which, at each step, arbitrarily removes any of the shortest cycles remaining in the digraph.

Theorem 2.3 [3]: For an eulerian digraph with at most five vertices (i.e. $c \leq 5$) the shortest cycle first algorithm finds a maximal decomposition.

After using the shortest cycle first algorithm to find a maximal decomposition, the following lemma gives an easy way to calculate index(G).

Lemma 2.1: Given an eulerian digraph, G, with at most 3 vertices, if a maximal decomposition has S self-loops, T 2-cycles and U 3-cycles then

$$\text{index}(G) = e(G) - (S + T + U) = T + 2*U$$

Proof: The first equality clearly holds because S + T + U is the number of cycles in the decomposition. The second follows from the first and the fact that $e(G) = S + 2*T + 3*U$. It can also be seen by observing we are charged k - 1 swaps for removing a cycle of length k.

□

The preceding has assumed we were given a particular arrangement. We will be interested in the expected value of index(G) where G is a digraph corresponding to a "random" arrangement. The next theorem extends Theorem 2.3 from [3] to apply to cases where a pebble is not equally likely to be red, white, or blue.

Theorem 2.4: Let G be the digraph corresponding to an arrangement in which each pebble is red with probability P_R, white with probability P_W and blue with probability P_B, then the expected value of index(G)=

$$(P_RP_W + P_RP_B + P_WP_B)n + o(n)$$

for large n, which is the optimal number of swaps required to order the arrangement.

Proof: We give the intuition behind the proof; an exact argument can be constructed along the lines of Theorem 2.3 from [3]. The only probable class of digraphs corresponding to an initial arrangement is shown in Figure 2.1. We only need to consider digraphs of this class, because the probability of a "random" digraph G being in this class approaches 1. The probability G is not in the class in o(1), and the contribution of these digraphs to index(G) is o(n) and hence can be ignored.

Figure 2.1 is obtained by noting the final red, white, and blue regions will be of size (approximately) P_Rn, P_Wn, and P_Bn respectively and that, originally, each contained red, white, and blue pebbles in the ratio $P_R:P_W:P_B$. By Theorem 2.3 and Lemma 2.1, $E(\text{index}(G)) = 1*(P_RP_Wn + P_RP_Bn + P_WP_Bn + o(n)) + 2*o(n)$

$$= (P_R P_W + P_R P_B + P_W P_B)n + o(n)$$

and by Theorem 2.2 this is the optimum. □

3. A LOWER BOUND

In this section we determine the optimal number of comparisons and swaps required to solve the 3-Way Partitioning Problem in the average case. We then show the Floyd-Rivest partitioning strategy is optimal.

Theorem 3.1: The optimal number of swaps to solve the 3-Way Partitioning Problem is $P_S P_L n + o(n)$, and the optimal number of comparisons is $(2 - P_L)n + o(n)$. (Assuming $P_S \leq P_L$.)

Proof: By Theorem 2.4 the optimal number of swaps is

$$(P_S P_M + P_S P_M + P_M P_L)n + o(n) = P_S P_L n + o(n)$$

because $P_M n = o(n)$.

To compute the number of comparisons required, we note that n buck operations must be performed. The optimal way of accomplishing this is by first comparing with v (or u if $P_S > P_L$), then u if necessary. This requires $[1*P_L + 2*(1-P_L)]*n + o(n) = (2-P_L)n + o(n)$ comparisons to perform n buck operations. □

We use the following lemma to calculate the cost of the Floyd-Rivest partitioning scheme.

Lemma 3.1: To partition a random arrangement of n elements about value u using the quicksort partitioning algorithm requires $(1-p)pn + o(n)$ swaps and n comparisons where p is the probability an element is less than u. Further, these figures are optimal.

Proof: Clearly n comparisons are used by the algorithm, and this is optimal. To calculate the number of swaps performed, note that the final left and right regions are of size $pn + o(n)$ and $(1-p)n + o(n)$ respectively, and each originally contained elements less than u and greater than u in the ratio $p : 1-p$. Therefore $p(1-p)n + o(n)$ elements are "out of place" in the left region and are swapped. Since the algorithm swaps an element at most once, $p(1-p)n + o(n)$ swaps are performed. To show this is optimal, use Theorem 2.4 with $P_R = p$, $P_W = 1-p$, $P_B = \emptyset$. □

Theorem 3.2: The Floyd-Rivest partitioning algorithm requires $P_S P_L n + o(n)$ swaps and $(2-P_L)n + o(n)$ comparisons which is optimal.

Proof: By Lemma 3.1, the first quicksort partitioning requires $P_S(1-P_S)n + o(n) = P_S P_L + o(n)$ swaps (since $p = P_S$) and n comparisons. The second step operates on $(1-P_S)n + o(n)$ elements and uses $P_M/(P_M + P_L)*(1-P_M/(P_M+P_L))n + o(n) = o(n)$ swaps (elements can only be medium or large so $p = P_M/(P_M+P_L)$) and $(1-P_S)n + o(n) = P_L + o(n)$ comparisons. □

4. A NEW ALGORITHM

In this section we develop a simple algorithm to solve the 3-way Partitioning Problem and prove it to be optimal in the average case. The algorithm is developed using techniques from [3] where an optimal algorithm for the Dutch National Flag problem was developed. This algorithm would not be optimal if it were used to solve the 3-way Partitioning Problem because of the difference in the probabilistic assumption in the two problems. However, an algorithm is constructed along similar lines.

The following predicates are convenient (u and v are given and are not changed):

issmall(x)	iff $x < u$
ismedium(x)	iff $u \leq x \leq v$
islarge(x)	iff $x > v$

In addition to the specifications given in Section 1, we assume we are given two variables, ML and MR such that ismedium(A[i]) for all i such that $ML < i < MR$. Further, we assume this region contains at least 2 elements (i.e. $ML + 1 \leq MR - 2$) since it simplifies our arguments. Finally, we assume the region is a fraction P_S across the array. All these conditions are met when the 3-way partitioning routine is called in algorithm SELECT.

The existence of a medium region having at least 2 elements significantly simplifies our discussion because what we do to elements to the left of this region never affects those to the right, and vice versa. Having 2 elements also is convenient because a swap can never

make the region between ML + 1 and MR - 1 be empty (even temporarily). Our development of the algorithm assumes a medium region of at least 2 elements at all times. This is a simplification; the region may temporarily contain only one element, but ignoring this possibility greatly simplifies our discussion. We introduce this complication when writing assertions for the final version of the algorithm.

Our loop invariant (I) will be

```
first  ≤ i ≤ L - 1  --> issmall(A[i]) and
ML + 1 ≤ i ≤ MR - 1 --> ismedium(A[i]) and
R + 1  ≤ i ≤ last   --> islarge(A[i]) and
```

and the following predicates I1 and I2 are useful:

```
I1 = first  ≤ i ≤ L - 1   --> issmall(A[i]) and
     ML + 1 ≤ i ≤ MR - 2  --> ismedium(A[i]) and
     R + 1  ≤ i ≤ last    --> islarge(A[i])

I2 = first  ≤ i ≤ L - 1   --> issmall(A[i]) and
     ML + 2 ≤ i ≤ MR - 2  --> ismedium(A[i]) and
     R + 1  ≤ i ≤ last    --> islarge(A[i])
```

The initializations are obvious from I. The end test for the loop is "(L ≤ ML) or (MR ≤ R)" since its negation will imply there are no unclassified elements. The body of the loop first increments L and decrements R over small and large elements respectively. I clearly holds after each iteration. Note that the existence of a non-empty region of medium elements in the middle of the array segment guarantees that each loop will terminate before either L or R becomes an invalid subscript.

Now A[L] is medium or large and A[R] is small or medium. Rather than immediately considering each of the four possible cases, we do

only what is required when A[L] is large and A[R] is small. Since this is the only case with non-negligible probability, we must make sure it is executed efficiently. To handle this case, we first verify that A[L] is large and A[R] is small. If this is the case, (and nearly always it will be), we swap A[L] and A[R]. After that, we "advance" L and R.

These "advances" are not quite as simple as incrementing or decrementing. After swapping A[L] and A[R] we have:

```
I2 and not islarge(A[L]) and not issmall(A[R])
     and L ≠ ML + 1 --> ismedium(A[ML+1])
     and R ≠ MR - 1 --> ismedium(A[MR+1])
```

so "advance L" must be designed so that

```
{I2 and issmall(A[L]) and L ≠ ML+1 --> ismedium(A[ML+1])}
                    "advance L"
                {I2 and ismedium(A[ML+1])}
                        {I1}
```

Clearly, we start with L := L + 1 to include this small element in the small region. This is followed by a statement(s) S such that

```
{I2 and L-1 ≠ ML+1 --> ismedium(A[ML+1])}
                    S
                {I2 and ismedium(A[ML+1])}
```

If L - 1 $\leq$ ML we need do nothing because then we have L - 1 ≠ ML + 1 and the precondition becomes I2 and ismedium(A[ML+1]). On the other hand, if L - 1 > ML we must "retreat" ML to keep the regions from overlapping. This works because

```
{I2} ML := ML + 1 {I2 and ismedium(A[ML+1])}.
```

(Actually we choose to write "advance L" as "IF L > ML THEN ML := ML +

1; L := L + 1"; these two forms are clearly equivalent.) Writing "advance R" is similar. This gives an incomplete version of the algorithm which is shown in Figure 4.1.

We can now analyze the average case complexity of the algorithm because this path through the loop is executed with probability 1-o(n). Clearly, as long as ($L \leq ML$) and ($MR \leq R$) the number of times we have ismedium(A[L]) or ismedium(A[R]) is $P_M n = o(n)$. In addition once one of "$L \leq ML$" or "$MR \leq R$" becomes false, there will be only o(n) unclassified elements because of the initial placement of the medium region. Since we will handle the three unlikely cases in a constant amount of time, the amount of effort expended on medium elements and on "cleaning up" after the small or large region contacts the medium region is o(n).

Based on the above, we restrict our attention to small and large elements. Since a swap will be done for each large pebble in the final small region, $P_S P_L n + o(n)$ swaps are required, which is optimal. To count the number of comparisons, note that one comparison is done for each small element in the final small region and for each large element in the final large region. Two comparisons are done for large elements in the small region and small elements in the large region. Hence $1 * (P_S^2 + P_L^2)n + 2 * (P_S P_L + P_S P_L)n + o(n) = (1 + 2P_S P_L)n + o(n)$ comparisons are done. This is larger than the optimal $(2-P_L)n + o(n) = (1 + P_S)n + o(n)$ unless $P_S = P_L = 1/2$. (Recall we assumed $P_S \leq P_L$.) This is because we did not always compare against v before comparing against u (if necessary). The

algorithm could be revised to do this at the expense of clarity by rewriting the "WHILE A[L] < u..." loop to completely calculate A[L]'s size (by comparing against v first, of course), storing it and refering to it when we need to know A[L]'s size. Note the algorithm is, however, optimal as it stands if the median is sought.

We now consider what to do in the case where A[L] or A[R] is medium after the swap. Again, we only consider L and ML. The statements handling this improbable case must satisfy

$$\{I2 \text{ and ismedium}(A[L]) \text{ and } L \neq ML + 1 \text{ --> ismedium}(A[ML+1])\}$$
$$S$$
$$\{I2 \text{ and ismedium}(A[ML+1])\}$$

Since the precondition implies the postcondition, we do not have to do anything. However, if $L \leq ML$, we swap A[L] and A[ML] in order to make progress. Since $L \leq ML$ this swap does not affect I2 and we have

$$\{I2 \text{ and } L \leq ML \text{ and ismedium}(A[L]) \text{ and ismedium}(A[ML+1])\}$$
$$A[L] \text{ <==> } A[ML]$$
$$\{I2 \text{ and ismedium}(A[ML]) \text{ and ismedium}(A[ML+1])\}$$

and now setting ML := ML - 1 will give us I2 and ismedium(A[ML+1]). This gives a complete algorithm, shown in Figure 4.2. Assertions are included, and it is routine to prove the algorithm from them. (These assertions account for the medium region temporarily having less than 2 elements.)

In the remainder of this section, we first simplify the algorithm then prove termination. The simplification is important because it lies on the probable path through the loop and thus is well worth making. It also makes the proof of termination easier. (We choose to

prove the algorithm, then simplify it because the simplified version appears to be much harder to prove.) The statements in question are "IF L > ML THEN ML := ML + 1" and the corresponding statement for R and MR. Our intuition tells us that once L > ML (i.e. the small and medium regions have run into each other) we will always have ML = L - 1, and it seems senseless to keep updating both variables. We only consider the statement concerning L and ML because the other case is similar.

It is easy to see that the value of ML - L, observed just before "IF A[L] < U THEN" is monotonic non-increasing. Therefore execution can be broken into two phases. In the first, L $\leq$ ML at this point. In this case, the body of "IF L > ML THEN" can never be executed. In the second phase we have L > ML, and the body of "IF L $\leq$ ML THEN" cannot be executed. Since the body of this IF contains the only references to ML (except for comparing it to L), we can safely eliminate the "IF L > ML THEN" statement if we can insure that L > ML from now on. This is guaranteed because ML will never change and L will only be increased. Therefore the "IF ML > L THEN" statement can be eliminated without affecting the execution. This gives us the final version of the algorithm (see Figure 4.3).

Finally, we prove this version terminates by examining the function ML - L + R - MR. Clearly $-4n < ML - L + R - MR < 4n$, so we can prove termination by showing that every path through the loop decreases this function. The only path that does not do this must go through the null ELSE of "IF L $\leq$ ML THEN" and the null ELSE of "IF R

=> MR THEN" (and, in addition, skip the bodies of both WHILE loops). This implies we had L > ML and R < MR at the beginning of this iteration. However this is impossible because it is the negation of the WHILE condition. Therefore all possible paths decrement the function and the algorithm terminates.

The algorithm was implemented on a DEC-10 in PASCAL and was compared with algorithm SELECT. For finding the median of 10,000 elements, the average of 10 trials was 780.1 ms. for algorithm SELECT and 785.6 ms. for our algorithm. The difference of 1% is not significant in view of the wide spread of execution timings (for example the times for algorithm SELECT ranged from 676 ms. to 855 ms.).

The near-equality of execution times is to be expected as both algorithms are optimal and does show that the 3-Way Partitioning Problem can be solved efficiently, in practice as well as theory.

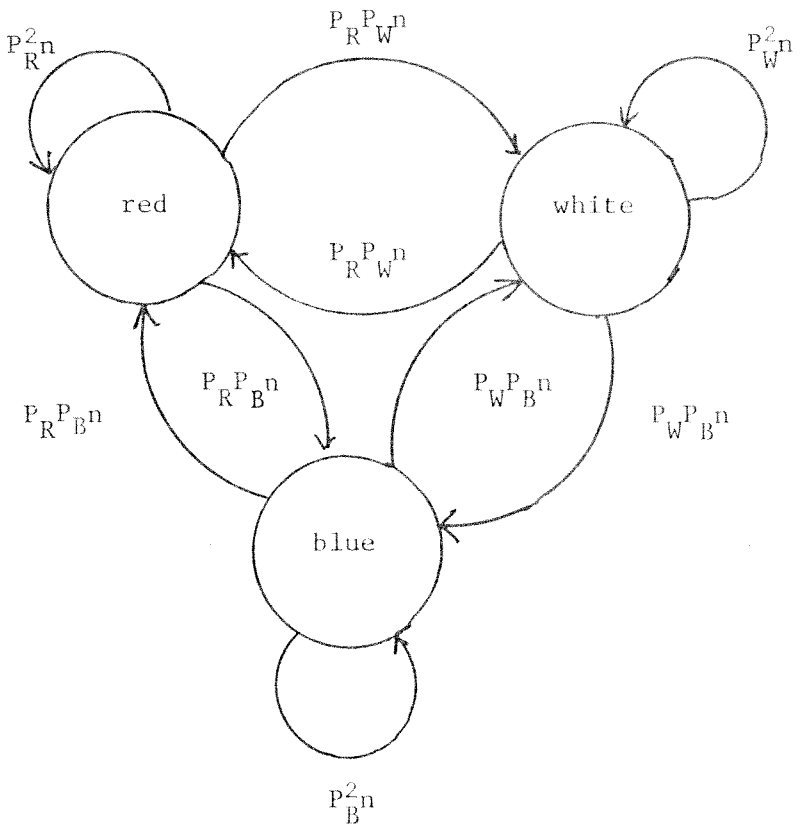

Figure 2.1

A very probable class of digraphs. The number beside each edge gives the number of edges in that direction between the given vertices. Each number is $\pm$ o (n).

```
L := FIRST;
R := LAST;

WHILE (L <= ML) OR (MR <= R) DO BEGIN

        WHILE A[L] < U DO L := L + 1;
        WHILE A[R] > V DO R := R - 1;

        A[L] <==> A[R];     { swap A[L] and A[R] }

        IF A[L] < U
           THEN
              BEGIN
                 { "advance" L }
                 IF L > ML THEN ML := ML + 1;
                 L := L + 1;
              END
           ELSE { almost never executed }

        IF A[R] > V
           THEN
              BEGIN
                 { "advance" R }
                 IF R < MR THEN MR := MR - 1;
                 R := R - 1;
              END
           ELSE { almost never executed }

END;
```

Figure 4.1
A preliminary version of the algorithm

```
{ P0 }

L := FIRST;
R := LAST;

WHILE (L <= ML) OR (MR <= R) DO BEGIN

   { loop invariant : I and ML+1 <= MR-2 }

        WHILE A[L] < U DO L := L + 1;
        WHILE A[R] > V DO R := R - 1;

        { P1 }

        A[L] <==> A[R];     { swap A[L] and A[R] }

        { P2 }

        IF A[L] < U
           THEN
              BEGIN
                 IF L > ML THEN ML := ML + 1;
                 L := L + 1;
              END
           ELSE
              IF L <= ML THEN BEGIN
                    A[L] <==> A[ML];
                    ML := ML - 1;
              END;

        { P3 }

        IF A[R] > V
           THEN
              BEGIN
                 IF R < MR THEN MR := MR - 1;
                 R := R - 1;
              END
           ELSE
              IF R >= MR THEN BEGIN
                 A[MR] <==> A[R];
                 MR := MR + 1;
              END;

END;
```

Figure 4.2
A complete version of the algorithm

The assertions are given on the next page. The predicates I, I1, and I2 are defined in the text.

```
P0 == ML+1 <= MR-2 and ( ML+1 <= i <= MR-1 --> ismedium( A[i] ) )


P1 == I2 and ismedium( A[ ML+1 ] ) and ismedium( A[ MR-1 ] )
        and L <= ML+1 <= MR-2 and MR-1 <= R
        and not issmall( A[L] ) and not islarge( A[R] )
        and ( L > ML --> ismedium( A[L] ) )
        and ( issmall( A[R] ) --> R >= MR )


P2 == I2 and     ( L <> ML + 1 --> ismedium( A[ ML+1 ] )
             and ( R <> MR - 1 --> ismedium( A[ MR-1 ] )
        and L <= ML+1 <= MR-2 and MR-1 <= R
        and not issmall( A[R] ) and not islarge( A[L] )
        and ( L > ML --> ismedium( A[R] ) )
        and ( issmall( A[L] ) --> R >= MR )


P3 == I1 and ( R <> MR-1 --> ismedium( A[ MR-1 ] ) )
        and  MR-1 <= R
        and not issmall( A[R] )
        and      (ismedium( A[R] ) and MR <= R) --> ML+1 <= MR-1
        and  not (ismedium( A[R] ) and MR <= R) --> ML+1 <= MR-2
        and      (islarge(  A[R] ) and MR >  R) --> ML+1 <= MR-3
```

Figure 4.2 (continued)

```
L := FIRST;
R := LAST;

WHILE (L <= ML) OR (MR <= R) DO BEGIN

        WHILE A[L] < U DO L := L + 1;
        WHILE A[R] > V DO R := R - 1;

        A[L] <==> A[R];     { swap A[L] and A[R] }

        IF A[L] < U
           THEN
              L := L + 1;
           ELSE
              IF L <= ML THEN BEGIN
                    A[L] <==> A[ML];
                    ML := ML - 1;
              END;

        IF A[R] > V
           THEN
              R := R - 1;
           ELSE
              IF R >= MR THEN BEGIN
                 A[MR] <==> A[R];
                 MR := MR + 1;
              END;

END;
```

Figure 4.3
The final version of the algorithm

REFERENCES

1. R. Floyd and R. Rivest, "Expected Time Bounds for Selection", CACM 18 (1975), 165-173.

2. E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1976.

3. J. Bitner, "An Asymptotically Optimal Algorithm for the Dutch National Flag Problem", SIAM J. Comput., to appear.

4. C. A. R. Hoare, "Algorithm 63 (PARTITION), Algorithm 64 (QUICKSORT), and Algorithm 65 (FIND)", CACM 4 (1961), 321.

5. M. Blum, R. W. Floyd, V. Pratt, R. Rivest, R. Tarjan, "Time Bounds for Selection", JCSS 7 (1973), 488-461.