Evaluation of permutation algorithms

M. K. Roy*

Computer Centre, Jadavpur University, Calcutta 700032, India

Six non-recursive algorithms which appear to be the best in respect of the permutation sequences they generate, have been considered for evaluation. Since the machine/compiler used has significant effect on the permutation algorithms, in this paper a different approach has been taken instead of the usual timing experiments. We have obtained the number of times certain constructs would be obeyed in terms of formulae related to n, the number of marks. A comparison based on these formulae shows that Ives' algorithm is the best of all.

(Received October 1976)

The algorithms for the generation of permutation sequences have received much attention in literature. Earlier, Ord-Smith (1970; 1971) presented a pioneering review of these algorithms and compared execution times of six very fast permutation algorithms. However, since then two new algorithms (Fike, 1975; Ives, 1976) and some improvements of the previously published ones (Ehrlich, 1973; Lenstra, 1973; Rohl, 1976) have been published. Therefore, it is worthwhile to make a comparative study of these algorithms.

It is usual in the literature to compare permutation algorithms by means of timing experiments. However, many authors have noted that the machine/compiler used has significant effect on the performance of an algorithm. Ives (1976) has indicated an approach in which he counts the number of times certain constructs—assignments, arithmetic operations, comparisons and subscript references—are executed for a particular value of n, the number of marks. A better approach would be to find out the number of times these operations are needed in terms of formulae related to n, and to use these formulae to obtain the figures-of-merit of an algorithm. This approach not only removes the confusions of the timing experiments but also gives an insight into the algorithms.

The present paper is divided into two parts. In the first part the algorithms from various considerations have been reviewed and six algorithms which appear to be the best in their respective categories have been selected. The second part is devoted to the analysis of the algorithms.

1. The six best algorithms

1. *Permutation sequences*

An important consideration during the selection of a permutation algorithm for an application is the order in which the configurations are generated and not merely the speed of the algorithm. Keeping in view the best algorithms considered by Ord-Smith (1971) and the algorithms published subsequently, it may be observed that there are six different permutation sequences. A representative of each is shown in **Table 1.** It may not be out of place to mention the various combinatorial advantages of these sequences.

It is well known that the Trotter-Johnson and Wells sequences have the property that a configuration can be generated from its predecessor by a single interchange of two marks. It may be observed that all other sequences require more transpositions. The added advantage in the Trotter-Johnson sequence is that the marks to be transposed are always adjacent in the preceding permutation.

Another property which may be called the *reflection-free* property is that either half of the sequence should not contain the reflection of any configuration belonging to that half. Two configurations are said to be reflections of each other if one read from left-to-right is identical with the other read from right-to-left. Both the Trotter-Johnson and the Ives sequences share this property (Roy, 1973; 1977) which is an advantage in many applications (Lenstra, 1973).

The Wells, lexicographic and pseudo lexical sequences share the combinatorial advantage that the k-th ($k < n$) intransitive subgroup of permutations is generated before the $(k + 1)$th mark is moved (Ord-Smith, 1968; Wells, 1971). Another advantage of the lexicographic sequence is the 'dictionary' order of its configurations. These properties are of importance in many applications. This author, however, fails to find any combinatorial advantage of the Fike sequence.

It may be noted that the properties mentioned above are for the sequences as shown in Table 1. A sequence may have many related sequences such as the sequences of reflections or inverse permutations of its configurations. An algorithm that generates a sequence requires only minor modifications to generate a related sequence. However, a related sequence may not always retain the properties of the original sequence. For example, it is easy to notice that the reflection-free property is not retained in the sequence of inverse permutations.

2. *A classification of the algorithms*

Observation shows that there are only two fundamental generating schemes. The first scheme generates the $n!$ permutations of the n marks from the knowledge of the $(n - 1)!$ permutations of the first $(n - 1)$ marks. Each of these $(n - 1)$ permutations yields n of the n-permutations. In the algorithms this is accomplished by moving the n-th mark through the arrangement of the $(n - 1)$ marks according to certain rules (these rules differ with the algorithms). The $(n - 1)!$ permutations of the first $(n - 1)$ marks are generated exactly in the same way from the $(n - 2)!$ permutations of the first $(n - 2)$ marks. The process is thus repeated until the trivial case of 1-permutation of the first mark is reached. On the other hand, the second scheme is as follows. Suppose we have a procedure that can generate only the $(k - 1)!$ permutations of $k - 1$ ($k < n$) given marks. The $k!$ permutations of the first k marks can then be generated by repeating the said procedure k times by taking $(k - 1)$ of the k marks at a time and the remaining mark occupying the k-th position. Starting with the trivial case of $k = 1$, we may repeat the technique with increasing k until $k = n$. For both the schemes the positions of the marks in the given arrangement may be considered from either end.

Both the schemes require backtracking and their differences are best illustrated by means of search trees (Wells, 1971). The search tree for the algorithms based on the first scheme is shown in **Fig. 1.** For easy reference we shall call these *A*-

*Author's present address, Regional Computer Centre, Calcutta, Jadavpur University Campus, Calcutta, 700 032, India.

Table 1 Six different permutation sequences of the marks 1, 2, 3, 4

Ives	*Trotter-Johnson*	*Fike*	*Wells*	*Pseudo-lexical*	*Lexicographic*
1234	1234	1234	1234	1234	1234
2134	1243	1243	2134	2134	2134
2314	1423	1432	2314	3124	1324
2341	4123	4231	3214	1324	3124
1342	4132	1324	3124	2314	2314
3142	1432	1342	1324	3214	3214
3412	1342	1423	1342	4123	1243
3421	1324	4321	3142	1423	2143
1423	3124	3214	3412	2413	1423
4123	3142	3241	4312	4213	4123
4213	3412	3412	4132	1243	2413
4231	4312	4213	1432	2143	4213
1324	4321	2134	1423	3412	1342
3124	3421	2143	4123	4312	3142
3214	3241	2431	4213	1342	1432
3241	3214	4132	2413	3142	4132
1243	2314	2314	2143	4132	3412
2143	2341	2341	1243	1432	4312
2413	2431	2413	3241	2341	2341
2431	4231	4312	2341	3241	3241
1432	4213	3124	2431	4231	2431
4132	2413	3142	4231	2431	4231
4312	2143	3421	4321	3421	3421
4321	2134	4123	3421	4321	4321

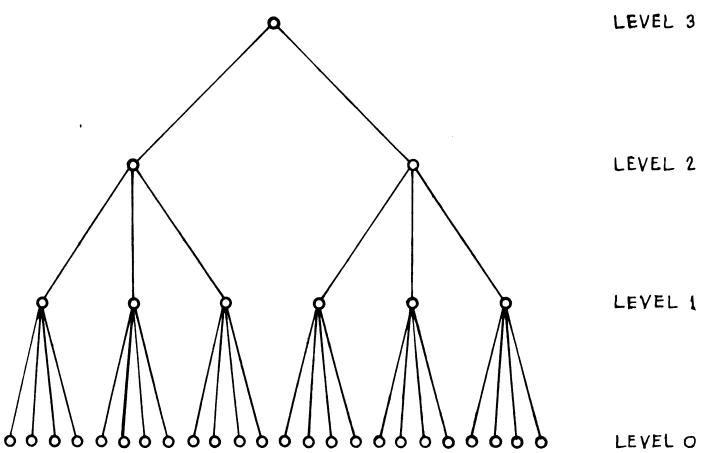

Fig. 1 Search tree of *A*-algorithm for $n = 4$

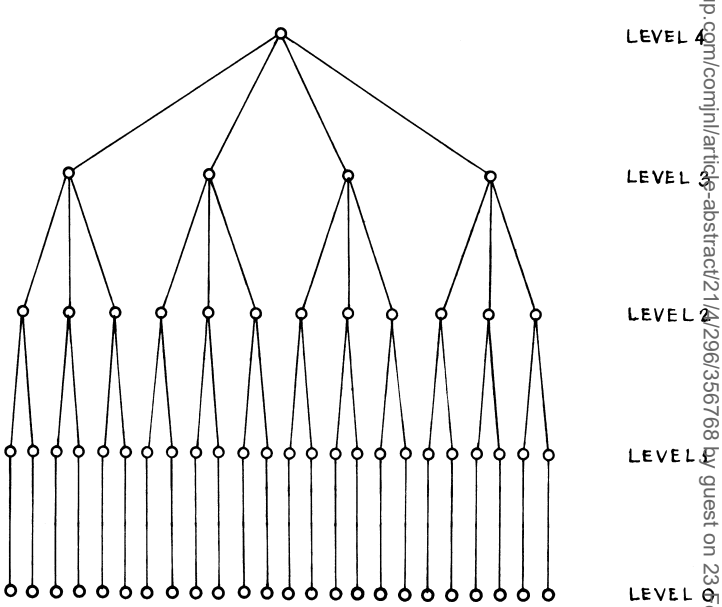

Fig. 2 Search tree of *B*-algorithm for $n = 4$

algorithms. **Fig. 2** shows the search tree for the algorithms using the second scheme. These will be called *B*-algorithms. The levels of nodes in these search trees correspond to the iteration levels. We shall mark these levels as 0, 1, 2, . . . , n (or $n - 1$) starting from the terminal nodes going upwards. It may be observed that algorithms generating the Trotter-Johnson, Ives and Fike sequences are *A*-algorithms whereas algorithms that generate the Wells, lexicographic and pseudo-lexicographic sequences are *B*-algorithms.

3. *Improvements to the algorithms*

A very common technique for improving the efficiency of a permutation algorithm is to deal separately with the iterations at the bottom of the search tree. Since the majority of the iterations take place at the bottom it is obvious that this will improve the efficiency, Still better would be to deal separately with the iterations at levels 0 and 1 and in this way one might like to handle the iterations at levels 0, 1, 2, . . . , k separately. However, for a large k, the implementation may become extremely difficult. This brings us to the question: what should be the optimum value of k?

In case of *A*-algorithms, for a large n (this is the situation when we are most concerned with efficiency), the number of iterations decreases sharply as the iteration levels increase. Therefore, the additional advantage when $k > 0$ is not much compared to the advantage when $k = 0$. As an illustration, an analysis similar to that described in Section 2 shows that the saving in the total number of operations in case of Trotter-Johnson algorithm is $12\,n! - 5.5\,(n-1)!$ when $k = 0$ and is $12n! + 5.5\,(n-1)! - 5.5\,(n-2)!$ when $k = 1$. Note that when $k = 1$, the average saving per configuration decreases as n is increased. Possibly this justifies the experimental observation of Ehrlich (1973a). Therefore, in case of *A*-algo-

rithms, we decide to restrict ourselves to the separate handling of iterations at level 0 only.

Among the available *A*-algorithms, PERM1 shown in **Fig. 3** may be considered as an exception to this rule. This algorithm which generates the Ives sequence, besides the iterations at level 0 also eliminates the iterations at all odd numbered levels by handling them separately. This has been possible without any implementation complexity because of the nature of the generation rules. However, the above decision of explicit handling of level 0 iterations in case of *A*-algorithms, gives us the scope for improving Algorithm 115 (Trotter, 1962; Ord-Smith, 1971) by exploiting this. A PL/I description of this algorithm called PERM is shown in **Fig. 4.** Of the algorithms that generate the Fike sequence, the two tuned procedures due to Rohl (1976) conforms to the above decision. One of them, the non-recursive tuned procedure described in PL/I with minor changes (see paragraph 4 for details) is shown in **Fig. 5.** This algorithm will be referred to by the name FIKE.

For the *B*-algorithms, the number of iterations decreases rather slowly as compared to *A*-algorithms particularly when n is large. Therefore, for the *B*-algorithms k should only be limited by the implementation difficulty. Boothroyd (1967) has shown a technique of dealing separately with the four lower levels of iterations for his algorithm which generates

```
PERM1:  PROCEDURE;
        DCL X(*) BINARY FIXED, BUSY BIT(1);
        DCL (N,N1,MAX,L1,K1,M,L,U,J,LL,X1,XL) BINARY FIXED STATIC,
            (LOC (2:MAX),K(2:MAX)) BINARY FIXED CONTROLLED;
FIRST:  ENTRY(X,BUSY);
        N=DIM(X,1); N1=N-1;
        MAX=N/2; X1=X(1); L1=1; K1=1;
        ALLOCATE LOC,K;
        DO M=2 TO MAX;
           LOC(M)=M; K(M)=M;
        END;
        BUSY='1'B;
NEXT:   ENTRY(X,BUSY);
        IF L1<N
        THEN DO; J=L1+1;
                 X(L1)=X(J); X(J)=X1;
                 L1=J; RETURN;
             END;
        ELSE DO;
             X(N)=X(1); X(1)=X1; L1=1; K1=K1+1;
             IF K1<N THEN RETURN;
             K1=1; L=2; U=N1;
             DO WHILE(U>L);
                LL=LOC(L);
                IF LL=U
                THEN DO; J=L; K(L)=K(L)+1; END;
                ELSE J=LL+1;
                XL=X(LL); X(LL)=X(J); X(J)=XL; LOC(L)=J;
                IF K(L)<U THEN RETURN;
                K(L)=L; L=L+1; U=U-1;
             END;
             BUSY='0'B; RETURN;
             END;
        END;  /* END OF PERM1 */
```

Fig. 3 PERM1: Generates the Ives sequence

```
PERM:   PROCEDURE;
        DCL X(*) BINARY FIXED, BUSY BIT(1);
        DCL (N,J,K,Q,T) BINARY FIXED STATIC,
            (M,L,LL,KN,PN,DN,XN) BINARY FIXED STATIC,
            (P(2:L),D(2:L)) BINARY FIXED CONTROLLED;
FIRST:  ENTRY(X,BUSY);
        N=DIM(X,1); L=N+1; LL=N+1;
        ALLOCATE P,D;
        DN=-1; PN=N; KN=0; XN=X(N); M=1;
        DO K=2 TO L; P(K)=K; D(K)=-1; END;
        BUSY='1'B;
NEXT:   ENTRY(X,BUSY);
        IF PN ¬= M THEN
        DO; Q=PN+DN;
            X(PN)=X(Q); X(Q)=XN;
            PN=Q; RETURN;
        END;
        M=LL-M; DN=-DN; J=L; K,KN=1-KN;
INDEX:  P(J),Q=P(J)+D(J);
        IF Q=J THEN
        DO; D(J)=-1; GO TO LOOP; END;
        IF Q¬=0 THEN GO TO TRANSPOSE;
        D(J)=1; K=K+1;
LOOP:   J=J-1; IF J>1 THEN GO TO INDEX;
        BUSY='0'B;
TRANSPOSE:
        Q=Q+K; K=Q+1;
        T=X(Q); X(Q)=X(K); X(K)=T;
        END; /*END OF PERM */
```

Fig. 4 PERM: Generates the Trotter-Johnson sequence (ACM 115 improved)

```
FIKE:   PROCEDURE;
        DCL X(*) BINARY FIXED, BUSY BIT(1);
        DCL (N,N1,I,K,DN,TEMP) BINARY FIXED STATIC,
            D(N1) BINARY FIXED CONTROLLED;
FIRST:  ENTRY(X,BUSY);
        N=DIM(X,1); N1=N-1;
        ALLOCATE D;
        DO I=1 TO N1; D(I)=I; END;
        BUSY='1'B;
NEXT:   ENTRY(X,BUSY);
        IF DN=0
        THEN DO; TEMP=X(N); DN=N1;
                 X(N)=X(DN); X(DN)=TEMP;
                 RETURN;
             END;
        ELSE DO; X(DN)=X(N); DN=DN-1;
                 IF DN¬=0
                 THEN DO;
                      X(N)=X(DN); X(DN)=TEMP;
                      RETURN;
                      END;
                 ELSE DO;
                      X(N)=TEMP; TEMP=X(D(N1));
                      X(D(N1))=X(N1); X(N1)=TEMP;
                      K=N1;
                      DO WHILE(K>2 & D(K)=1);
                         D(K)=K; K=K-1;
                         TEMP=X(D(K)); X(D(K))=X(K); X(K)=TEMP;
                      END;
                      IF D(K)=1
                      THEN BUSY='0'B;
                      ELSE DO;
                           D(K)=D(K)-1;
                           TEMP=X(K); X(K)=X(D(K)); X(D(K))=TEMP;
                           END;
                      END;
             END;
        END; /* END OF FIKE */
```

Fig. 5 FIKE: Generates the Fike sequence

```
BOOTH:  PROCEDURE;
        DCL X(*) BINARY FIXED, BUSY BIT(1);
        DCL (N,K,KLESS1,DK,L) BINARY FIXED STATIC,
            D(4:L) BINARY FIXED CONTROLLED,
            (D3,TEMP) BINARY FIXED STATIC,
            ODD BIT(1) STATIC;
FIRST:  ENTRY(X,BUSY);
            N=DIM(X,1);
            ODD='0'B;
            IF N>2 THEN
              DO; BUSY='1'B;
                  IF N=3 THEN L=4; ELSE L=N;
                  ALLOCATE D;
                  DO K=4 TO L; D(K)=0; END;
                  D3=0;
                  IF N=3 THEN D(4)=3;
              END;
NEXT:   ENTRY (X,BUSY);
        IF ODD
        THEN  DO; ODD='0'B;
              IF D3<2
              THEN    DO; D3=D3+1; X(2)=X(3); X(3)=TEMP; END;
              ELSE    DO; KLESS1=3; K=4; D3=0;
              COUNT:  DK=D(K);
                      IF DK ¬= KLESS1 THEN GO TO SWAP;
                      D(K)=0;
                      IF K<N  THEN  DO;
                                    KLESS1=K; K=K+1;
                                    GO TO COUNT;
                                    END;
                      BUSY='0'B; RETURN;
              SWAP:   DK,D(K)=DK+1;
                      IF DK>2 THEN
                         DO; IF MOD(K,2)=0 THEN KLESS1=K-DK; END;
                      TEMP=X(K); X(K)=X(KLESS1); X(KLESS1)=TEMP;
                      END;
              END;
        ELSE  DO; TEMP=X(1); X(1)=X(2);
                  X(2)=TEMP;  ODD='1'B;
              END;
        END; /* END OF BOOTH */
```

Fig. 6 BOOTH: Generates the Wells sequence (CJ 29 improved)

the Wells sequences. But many authors (Lenstra, 1973; Ives, 1976) have observed that the technique is sensitive to the compiler/machine used because of its heavy dependence on *switch* variables. In the absence of any other elegant technique which deals with $k \geqslant 3$, we decide to restrict ourselves to $k = 2$ in the case of the Wells algorithm as well as other *B*-algorithms. It may be remarked that the implementation of separate handling of the three lowest levels can be done easily for any *B*-algorithm. Since among the existing *B*-algorithms only CJ 28 (Phillips, 1968) exploits this, we have improved the other algorithms, namely CJ 29, ACM 308 and ACM 323 (Ord-Smith, 1971). The improved algorithms called BOOTH, PSEUDO and BESTLEX generate the respective sequences for $n > 1$. **Figs. 6 to 8** give PL/I descriptions of BOOTH, PSEUDO and BESTLEX.

The idea of avoiding nesting of iterations by means of exten-

```
PSEUDO:   PROCEDURE;
          DCL X(*) BINARY FIXED, BUSY BIT(1);
          DCL (N,K,M,L) BINARY FIXED STATIC,
              (Q3,T) BINARY FIXED STATIC,
              Q(4:L) BINARY FIXED CONTROLLED,
              ODD BIT(1) STATIC;
FIRST:    ENTRY(X,BUSY);
          N=DIM(X,1);
          ODD='1'B;
          IF N>2 THEN
          DO; BUSY='1'B;
              IF N=3 THEN L=4; ELSE L=N;
              ALLOCATE Q;
              DO M=4 TO L; Q(M)=1; END;
              Q3=1;
              IF N=3 THEN Q(4)=4;
          END;
NEXT:     ENTRY(X,BUSY);
          IF ODD THEN
          DO;ODD='0'B;
              T=X(2); X(2)=X(1); X(1)=T;
              RETURN;
          END;
          ODD='1'B;
          IF Q3<3 THEN
          DO; Q3=Q3+1;
              X(1)=X(3); X(3)=T;
              RETURN;
          END;
          Q3=1; K=4;
LOOP:     M=Q(K);
          IF M=K THEN
          DO; IF K<N THEN DO; Q(K)=1; K=K+1; GO TO LOOP; END;
                      ELSE DO; BUSY='0'B;
                               IF N=3 THEN K=3;
                               GO TO TRINIT;
                           END;
          END;
          Q(K)=M+1;
TRINIT:   M=1;
TRANSPOSE:
          T=X(M); X(M)=X(K); X(K)=T;
          M=M+1; K=K-1;
          IF M<K THEN GO TO TRANSPOSE;
          END; /* END OF PSEUDO */
```

Fig. 7 PSEUDO: Generates the Pseudo-lexicographic sequence (ACM 308 improved)

```
BESTLEX:  PROCEDURE;
          DCL X(*) BINARY FIXED, BUSY BIT(1);
          DCL (N,K,M,L) BINARY FIXED STATIC,
              (Q3,T) BINARY FIXED STATIC,
              Q(4:L) BINARY FIXED CONTROLLED,
              ODD BIT(1) STATIC;
FIRST:    ENTRY(X,BUSY);
          N=DIM(X,1);
          ODD='1'B;
          IF N>2 THEN
          DO; BUSY='1'B;
              IF N=3 THEN L=4; ELSE L=N;
              ALLOCATE Q;
              DO M=4 TO L; Q(M)=1; END;
              Q3=1; IF N=3 THEN Q(4)=4;
          END;
NEXT:     ENTRY(X,BUSY);
          IF ODD THEN
          DO; ODD='0'B;
              T=X(2); X(2)=X(1); X(1)=T;
              RETURN;
          END;
          ODD='1'B;
          IF Q3<3 THEN
          DO; IF Q3=1
              THEN DO;
                   X(1)=X(2); X(2)=X(3); X(3)=T;
                   Q3=2; RETURN;
                   END;
              ELSE DO;
                   X(1)=X(3); X(3)=X(2); X(2)=T;
                   Q3=3; RETURN;
                   END;
          END;
          Q3=1; K=4;
LOOP:     M=Q(K);
          IF M=K THEN
          DO; IF K<N
              THEN DO; Q(K)=1; K=K+1; GO TO LOOP; END;
              ELSE DO; BUSY='0'B;IF N=3 THEN K=3;
                       GO TO TRINIT;
                   END;
          END;
          T=X(M); X(M)=X(K); X(K)=T;
          Q(K)=M+1; K=K-1;
TRINIT:   M=1;
TRANSPOSE:T=X(M); X(M)=X(K); X(K)=T;
          M=M+1; K=K-1;
          IF M<K THEN GO TO TRANSPOSE;
          END; /* END OF BESTLEX */
```

Fig. 8 BESTLEX: Generates the Lexicographic sequence (ACM 323 improved)

sive bookkeeping is due to Ehrlich (1973). He also claimed that the high speed of his algorithm PERMU was partly due to its loop-free implementation. Ives (1976) observes that such loopless implementation does not contribute to the average

Table 2 Average number of operations required to generate a configuration for eight algorithms

Algorithm	*Average operations/configuration*
PERM1	$9 + 1/n + 20/\{n(n-1)\} + R1$
PERM	$9 + 19/n + 13.5/\{n(n-1)\} + R2$
PERMU	$9 + 32/n + 23/\{n(n-1)\} + R3$
FIKE	$11 + 22/n + 17/\{n(n-1)\} + R4$
BOOTH	$5e - 8\ \text{Cosh}\ 1 + 10.83333 = 12.08010$
PSEUDO	$8e + 12\ \text{Sinh}\ 1 - 18.83333 = 14.34867$
BESTLEX	$8e + 12\ \text{Cosh}\ 1 - 25 = 15.26323$
CJ 28	$10.5e + 12\ \text{Cosh}\ 1 - 30.66657 = 16.39226$

where

$R1 = (3\phi'(n-3) + 23\phi'(n-4)n!$

$R2 = 13.5\phi(n-3)/n!$

$R3 = (4(n-3)! + 12\phi(n-3))/n!$

$R4 = 17\phi(n-3)/n!$

and ϕ, ϕ' are same as in Table 2.

performance of his algorithms. We find that this is also true in the case of PERMU. The algorithm which generates the Trotter-Johnson sequence is similar to PERM except that it is loop-free while PERM uses loops. On the IBM 370/155, PERMU has been found to be slightly slower (for $n = 8$, the execution time for PERMU is 6.0 secs. against 5.7 secs, for PERM). Our analysis in Section 2 also shows that (see **Table 2**) PERM is better than PERMU in terms of the average number of operations required. Thus the suggestion that loop-free implementation should contribute to the performance of an algorithm does not seem to be convincing. This excludes the loopless algorithms from our consideration.

4. *Algorithm structures and the six best algorithms*

Permutation algorithms have traditionally been described as non-recursive procedures, but recently it has been reported that algorithms described as recursive procedures run faster than the corresponding non-recursive versions on some systems (Lenstra, 1973; Fike, 1975; Rohl, 1976). Rohl has also observed performances contradictory to this on a CDC 7600. However, the effect of recursion on permutation algorithms needs further investigation and we propose to report our findings in a subsequent paper. Here, we wish to concentrate mainly on non-recursive algorithms because:

(*a*) Most algorithms are available in non-recursive description.

(*b*) Non-recursive descriptions are more universal in the sense that some programming languages like FORTRAN, COBOL and many implementations of PL/I and ALGOL do not permit recursive calls.

Keeping in view only the non-recursive algorithms which are not made loop-free, it may be observed that there are only seven algorithms which conform to our earlier decision on the separate handling of iteration levels at the bottom of the search trees. These algorithms are PERM1, PERM, FIKE, BOOTH, PSEUDO and BESTLEX shown in Figs. 3 to 8 and CJ 28 (Phillips, 1967; Ord-Smith, 1971). Among these both BESTLEX and CJ 28 generate lexicographic sequence. Our analysis in Section 2 will show that BESTLEX is superior to CJ 28 in terms of the total number of operations required (see Table 2).

The six best algorithms shown in Figs. 3 to 8 have been described in the traditional form such that each call generates the next configuration. Of these algorithms, PERM1 is a previously published algorithm (Ives, 1976) reproduced here with a minor improvement. FIKE is a PL/I translation of non-recursive tuned procedure of Rohl (1976). However, FIKE in its original form was described in such a way that a

Table 3 Formulae representing the figures-of-merit for six best permutation algorithms

Algorithm	*Permutation sequence*	*Expressions giving the number of times the different operations are required*	*Counts for $n = 7$ obtained theoretically*	*Actual counts for $n = 7$*
PERM1	Ives	$a = 4n! + 9\,\phi'(n-2) + \phi'(n-3)$	21,320	21,324
		$b = n! + (n-2)! + 3\,\phi'(n-4)$	5,178	5,180
		$c = n! + (n-1)! + 3\phi'(n-2)$	6,138	6,139
		$d = 3n! + 7(n-2)! + 2\phi'(n-3) + 8\phi'(n-4)$	16,060	16,061
PERM	Trotter-Johnson	$a = 4n! + 8(n-1)! + 4.5\phi(n-2)$	26,604	26,608
		$b = n! + 5(n-1)! + 2.5\phi(n-2)$	9,020	9,022
		$c = n! + 2(n-1)! + 2.5\phi(n-2)$	6,860	6,861
		$d = 3n! + 4\phi(n-1)$	18,608	18,609
FIKE	Fike	$a = 4n! + 8(n-1)! + 6\phi(n-2)$	26,832	26,827
		$b = n! + \phi(n-2)$	5,192	5,192
		$c = n! + 2\phi(n-1)$	6,784	6,782
		$d = 5n! + 12(n-1)! + 8\phi(n-2)$	35,056	35,047
BOOTH	Wells	$a = (e + 4\,\text{Sinh}\,1 - 13/6)n! = 5.25242\,n!$	26,472	26,463
		$b = (37/3 - 4\,\text{Cosh}\,1 - 2e)n! = 0.72445\,n!$	3,651	3,647
		$c = 2n!$	10,080	10,077
		$d = (2e - 4/3)n! = 4.10323\,n!$	20,080	20,674
PSEUDO	Pseudo-lexicographic	$a = (3e + 5\,\text{Sinh}\,1 - 47/6)n! = 6.19752\,n!$	31,235	31,227
		$b = (e + 2\,\text{Sinh}\,1 - 23/6)n! = 1.23535\,n!$	6,226	6,222
		$c = (2e + \text{Sinh}\,1 - 4.5)n! = 2.11176\,n!$	10,643	10,642
		$d = (2e + 4\,\text{Sinh}\,1 - 16/3)n! = 4.80404\,n!$	24,212	24,205
BESTLEX	Lexicographic	$a = (3e + 5\,\text{Cosh}\,1 - 28/3)n! = 6.53692\,n!$	32,946	32,938
		$b = (e + 2\,\text{Cosh}\,1 - 5)n! = 0.80444\,n!$	4,054	4,051
		$c = (2e + \text{Cosh}\,1 - 14/3)n! = 2.31298\,n!$	11,657	11,656
		$d = (2e + 4\,\text{Cosh}\,1 - 6)n! = 5.60889\,n!$	28,269	28,261

a = assignments
b = arithmetic operations
c = comparisons
d = subscript references

$\phi(n) = n! + (n-1)! + (n-2)! + \ldots + 2!$
$\phi'(n) = n! + (n-2)! + (n-4)! + \ldots + 3!(\text{or } 2!)$

single call generates all configurations successively. Therefore, it has been modified so as to present it in the classical form in which the other algorithms have been described. As already mentioned the remaining four, namely PERM, BOOTH, PSEUDO and BESTLEX are improvements of previously published algorithms.

In the accompanying figures, each algorithm has been described as a PL/I procedure. The parameters are the array X containing the marks and a BIT(1) variable BUSY. At first entry BUSY is set to '1'B and only when all the configurations have been generated is it returned as '0'B. The first entry is CALL FIRST (X,BUSY) and the subsequent entries are CALL NEXT (X,BUSY).

2. Analysis of algorithms

1. *Theoretical evaluation: figures-of-merit*

As has been stated in the introduction, the number of assignments, arithmetic operations, comparisons and subscript references that are required for the generation of all the $n!$ configurations will be considered as the figures-of-merit of an algorithm. The expressions for each of these figures-of-merit for the six best algorithms are given in **Table 3**. The actual derivation of these formulae is quite involved and is not necessary. Here, we shall only indicate how these have been

obtained. The description of the algorithm was considered as divided into a number of sections of code by the control statements involving comparisons. With reference to the search tree, the number of times these sections would be executed was found next. The rest is obvious. However, the following points deserve mention in this connection.

(*a*) The formulae do not include the operations needed during initialisation (the section entered on first entry) as these are executed only once.

(*b*) Some variables (such as the one representing n) do not change their value during execution. Unary minus (−) signs preceeding such variables or constants and binary arithmetic operations involving such variables and constants have not been considered as arithmetic operations. The reason is that one can always replace these constructs by a variable in the initialisation section, thus avoiding the operations during generation.

(*c*) In the case of a loop, the components of assignment, arithmetic operation and comparison have been obtained by determining the requirements of these operations, if the statements controlling the loop were written using arithmetic and simple IF statements only. Thus the dissection of a loop control statement of the form DO I = 1 TO N; ...; END; into the said components is given by one assignment outside the loop and one assignment, one arithmetic operation and one comparison for the number of times the loop is repeated

(*d*) In BOOTH, the use of the built-in function MOD(K,2) has been considered equivalent to two arithmetic operations

(*e*) In certain cases, the expressions given in Table 3 have been approximated under the assumption that n is large.

As a verification of these formulae, counts computed from these and the actual counts obtained through a computer when $n = 7$ have been included in Table 3.

The number of comparisons required by FIKE deserves a note. As already mentioned, FIKE was recast in the classical form. However because of this modification, as implemented in Fig. 5, $n!$ additional comparisons are required. Thus if all the algorithms were recast in a form such that the procedure is called once and generates all permutations, it would favour FIKE. It may be observed that in the case of other algorithms, the necessary change would require no extra operation. Therefore, for the sake of fairness the additional $n!$ comparisons have not been considered in the corresponding formula for FIKE in Table 3.

2. *Observations and comparisons of algorithms*

The four types of operation considered above may not require equal amounts of time for execution. Therefore, to get an estimate of the execution time, the formulae given in Table 3 need to be used with proper weights depending on the computer/compiler combination. Moreover, by simple changes in coding, it is possible to reduce one type of operation at the cost of another. For example, the following PL/I statements TEMP = X (D(K)); X (D(K)) = X(K); X(K) = TEMP; using three assignments and six subscript references could be replaced by

DK = D(K); TEMP = X (DK); X(DK) = X(K); X(K) = TEMP; which uses four assignments and five subscript references. Situations like this are quite common in permutation algorithms. Therefore, depending on the system, the algorithms should be optimised in favour of less expensive operations.

However, we have noticed that although the individual count for different types of operations may vary depending on the coding, their total remains the same in most cases. Thus, for a machine independent comparison of these algorithms, we feel, the total of the four operations needed provides a good measure of an algorithm's performance. Therefore, the average of these four operations required to generate a configuration from its predecessor will be used as an indicator of the relative merit of an algorithm. Table 2 gives the average number of operations for eight algorithms. Algorithms PERMU and CJ 28 have also been considered here to support the claims made in Section 1.

From Table 2, it appears that PERM1 and BOOTH are the best *A* and *B*-algorithms respectively. PERM1 which generates Ives sequences seems to be the best of all and as such should be used unless a specific sequence is required. It may also be noted that in general, *A*-algorithms are superior to *B*-algorithms.

Acknowledgement

Appreciation is expressed to the referee for providing valuable suggestions for significant improvement of this paper. The author is also indebted to Professor J. S. Chatterjee for his guidance and encouragement and to Mr S. Sarkar for his help during the timing tests on IBM 370/155.

References

BOOTHROYD, J. (1967). Algorithms 29, 30, *The Computer Journal*, Vol. 10, p. 310.
EHRLICH, G. (1973a). Loopless algorithms for generating permutations, combinations and other combinatorial configurations, *JACM*, Vol. 20, p. 500.
EHRLICH, G. (1973b). Four combinatorial algorithms, *CACM*, Vol. 16, p. 690.
FIKE, C. T. (1975). A permutation generation method, *The Computer Journal*, Vol. 18, p. 21.
IVES, F. M. (1976). Permutation enumeration: four new permutation algorithms. *CACM*, Vol. 19, p. 68.
JOHNSON, S. M. (1963). An algorithm for generating permutations, *Math. Comp*, Vol. 17, p. 28.
LENSTRA, J. K. (1973). Recursive algorithms for enumerating subsets, lattice-points, combinations and permutations, Report BW 28/73, Mathematisch Centrum, Amsterdam.
ORD-SMITH, R. J. (1967). Generation of permutations in pseudo-lexicographic order, Algorithm 308, *CACM*, Vol. 10, p. 452.
ORD-SMITH, R. J. (1968). Generation of permutations in lexicographic order, Algorithm 323, *CACM*, Vol. 11, p. 117.
ORD-SMITH, R. J. (1970). Generation of permutation sequences: part 1, *The Computer Journal*, Vol. 13, p. 152.
ORD-SMITH, R. J. (1971). Generation of permutation sequences: part 2, *The Computer Journal*, Vol. 14, p. 136.
PHILLIPS, J. P. N. (1967). Algorithm 28, *The Computer Journal*, Vol. 10, p. 311.
ROHL, J. S. (1976). Programming improvements to Fike's algorithm for generating permutations, *The Computer Journal*, Vol. 19, p. 156.
ROY, M. K. (1973). Reflection-free permutations, rosary permutations, and adjacent transposition algorithms, *CACM*, Vol. 16, p. 312.
ROY, M. K. (1977). A note on reflection-free permutation enumeration, *CACM*, Vol. 20, p. 823.
TROTTER, H. F. (1962). Perm., Algorithm 115, *CACM*, Vol. 5, p. 434.
WELLS, M. B. (1971). *Elements of combinatorial computing*, Pergamon Press, New York.