SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 8, 3–9 (1978)

A Note on Computing Multiple Sums

BERTRAND MEYER

E.D.F., Direction des Etudes et Recherches, Service I.M.A.,
1 Avenue du Général de Gaulle, 92141 Clamart, France

SUMMARY

In a recent paper,[1] a method was given for computing multidimensional sums in FORTRAN in a particular case. We show that the general problem of computing multidimensional sums is best expressed by a recursive definition which can then be translated into a programming language. The method is applied to an example optimization problem.

KEY WORDS Summations Recursion FORTRAN Combinatorialalgorithms

INTRODUCTION

In a recent paper[1] a method was given to compute multiple sums of the following form in FORTRAN:

$$sum = \sum_{n_1=1}^{k} \sum_{n_2=1}^{k} \cdots \sum_{n_m=1}^{k} f(n_1, n_2, \ldots, n_m) \qquad (1)$$

The method relied upon the fact that in this particular form all indices range over the same domain, namely $\{1, 2, \ldots, k\}$; the whole looping process can thus be controlled by a single m-digit integer in base k.

The purpose of the present paper is to show that the problem above is just an instance of the more general one of computing multiple sums over arbitrary finite sets. A solution to this general problem is best expressed by a simple recursive formula, which can then be made into a FORTRAN program by applying the usual rules for recursion removal. We shall give an application of this method to the solution of an optimization problem which was recently posed by a programmer at our Centre.[2]

Although no new theoretical result is presented in this paper, we feel it useful to show in a particular instance how illuminating a recursive approach can be for certain classes of problems, such as tree traversal. As a matter of fact, many real-world FORTRAN programs contain hidden and in most cases unconscious use of recursion, through the manipulation of some arrays which are but implementations of stacks. It seems to us, for instance, that the array *IVEC* which is used in Reference 1 for the representation of an m-digit integer in base k is actually a recursion stack. A good knowledge of recursive techniques can help FORTRAN programmers to improve and clarify their programs.

THE PROBLEM

A multiple sum can usually be written (using notations similar to Reference 1):

$$sum = \sum_{n_1=a_1}^{b_1} \sum_{n_2=a_2}^{b_2} \cdots \sum_{n_m=a_m}^{b_m} f(n_1, n_2, \ldots, n_m) \qquad (2)$$

0038–6644/78/0108–0003$01.00

Received 5 July 1976
Revised 30 November 1976

where there is no reason for the a_i's and b_i's to be all equal. Formula (2) can be viewed as a summation over the leaves of a tree of depth m, where all internal nodes at a given level i $(1 < i < m)$ have the same number of sons, the branches to which are labelled $a_i, a_i + 1, \ldots, b_i$. This is shown on Figure 1, where, as in later figures, nodes where the summation has to be performed are marked by squares.

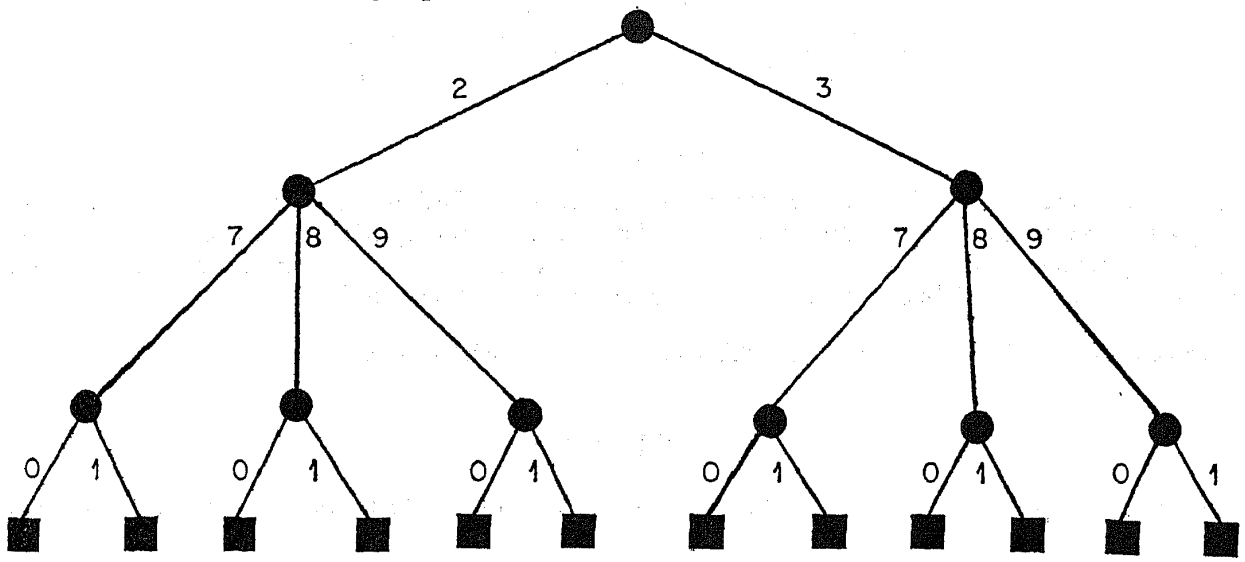

Figure 1. Tree representation of

$$\sum_{n_1=2}^{3} \sum_{n_2=7}^{9} \sum_{n_3=0}^{1} f(n_1, n_2, n_3)$$

Quite frequently, however, formula (2) is still too restrictive, and many practical problems require performing a summation over an arbitrary set of nodes of a given tree. We shall express f as a function of the sequence of labels from the root to a given node; in particular, it can have a variable number of arguments. Thus, Figure 2 represents the sum:

$$f(5, 0, -1) + f(5, 0) + f(5, 1) + f(6) + f(6, 2, 0)$$

The general form of the sum to be performed is:

$$sum = \sum_{x \in T,\, cond(x)} f(x) \tag{3}$$

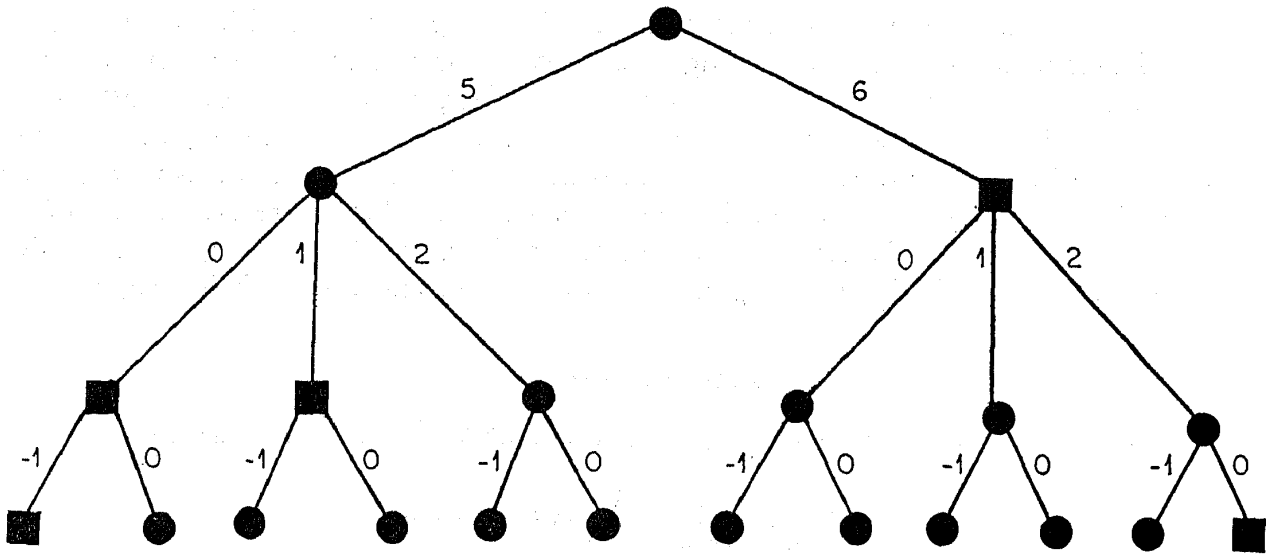

Figure 2. Tree representation of

$f(5, 0, -1) + f(5, 0) + f(5, 1) + f(6) + f(6, 2, 0)$

where T is a tree; x will range over the set of nodes in T and will be expressed by the sequence of labels from the root: $x_1, x_2, \ldots, x_k$; and *cond* is a certain condition on x which selects the nodes to be taken into account for the summation of f. We shall restrict ourselves to trees of a given depth, m, where all branches at level $i\,(1<i<m)$ are labelled $a_i, a_i+1, \ldots, b_i$; the method can be easily generalized to any finite tree.

An interesting case occurs when the predicate *cond* is such that $cond(n_1, \ldots, n_k)$ cannot be true whenever $cond(n_1, \ldots, n_h)$ is true for some $h\,(1 \leqslant h < k \leqslant m)$. We shall say in such a case that *cond* satisfies the 'prefix condition'. In such a case, summation only has to be performed over some of the leaves of some subtree of T (Figure 3). Formulae (1) and (2) trivially satisfy the prefix condition.

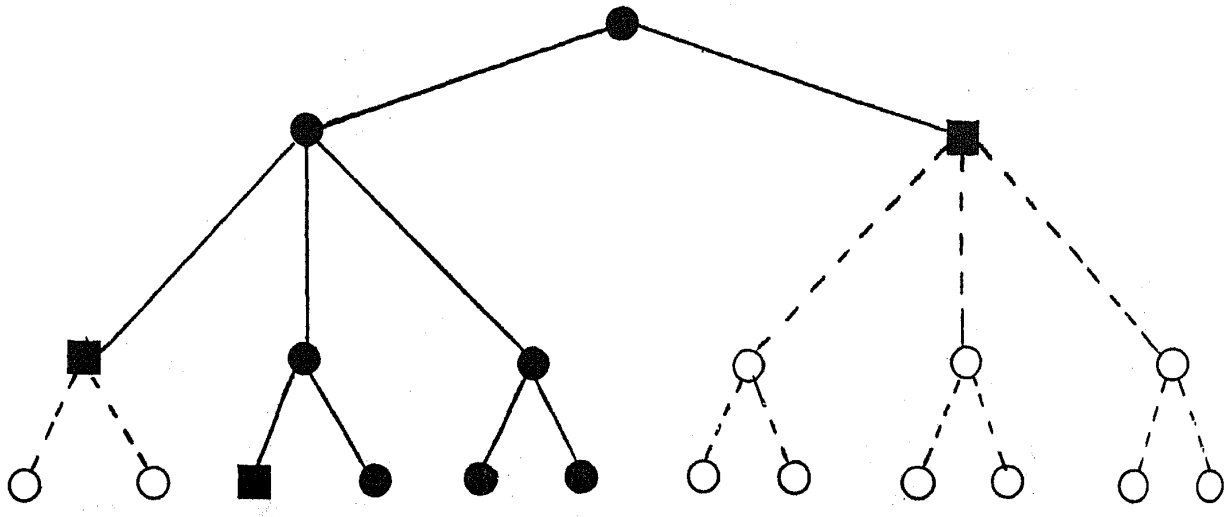

Figure 3. Example of prefix condition

AN EXAMPLE

As an example, consider the following problem,[2] which arose recently. We want to compute the expected cost of operating a thermal power system so as to satisfy a global power requirement R. The system is composed of m 'plants'; plant i operates at cost C_i $(i = 1, 2, \ldots, m)$, and produces power $r_{i,j}$ with probability $P_{i,j}$ $(j = 1, 2, \ldots, t_i)$. Concretely, this means that the different plants will be started in order of increasing cost, and that plant i is composed of t_i 'units' which have a certain probability of breaking down.

Thus, the expected value of operating the system so as to meet the power requirement is given by formula (3) above, where

$$a_i = 1 \;(i = 1, 2, \ldots, m)$$

$$b_i = t_i \quad (i = 1, 2, \ldots, m)$$

$$f(x_1, x_2, \ldots, x_k) = \left(\prod_{i=1}^{k} P_{i,x_i}\right)\left(\sum_{i=1}^{k} C_i r_{i,x_i}\right)$$

and

$$cond(x_1, x_2, \ldots, x_k) \text{ is: } \sum_{i=1}^{k} r_{i,x_i} \geqslant R \quad \text{and} \quad \sum_{i=1}^{k-1} r_{i,x_i} < R$$

(Note that the way *cond* is expressed shows it satisfies the prefix condition).

As a simple illustrative case, consider the situation shown in Figure 4. There are three plants, each of which consists of a single unit. Let us assume that the powers produced are:

$$r_1 = 2 \text{ for plant } 1; \quad r_2 = 1 \text{ for plant } 2; \quad r_3 = 2 \text{ for plant } 3$$

Associated costs are C_1, C_2, C_3, which are such that $C_1 < C_2 < C_3$: the plants will be started in the order 1, 2, 3. If plants 1, 2, 3, i.e. their single units, have the respective probabilities P_1, P_2 and P_3 of being in working order, the powers produced are r_1, r_2 and r_3 with probabilities P_1, P_2 and P_3, and 0 with probabilities $1-P_1$, $1-P_2$ and $1-P_3$.

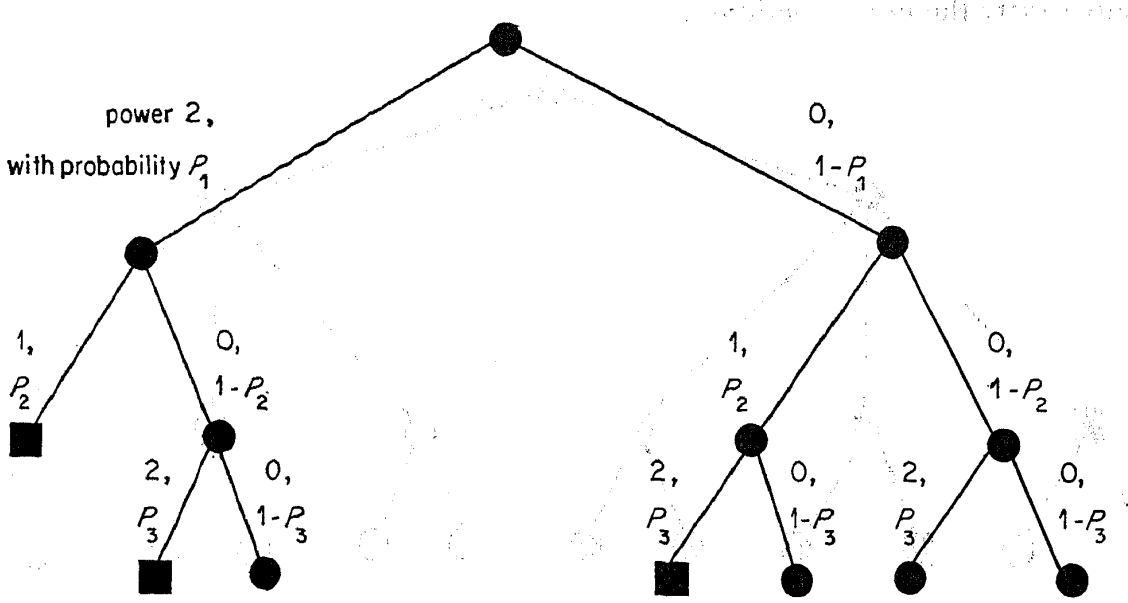

Figure 4. Tree representation of summation required

If we assume that the global power requirement is $R = 3$, the admissible configurations are those represented by square nodes on Figure 4, and the expected operating cost to be computed is:

$$P_1 P_2(2C_1 + C_2) + P_1(1-P_2) P_3(2C_1 + 2C_3) + (1-P_1) P_2 P_3(C_2 + 2C_3)$$

THE METHOD

Formula (3) can be computed by the following recursive function definitions (which make use of list arguments such as *list*, of the special empty list called *NIL*, of the operations *cons*$(x, list)$ which adds object x to list *list*, and of conditional expressions of the form **if** c **then** e_1 **else** e_2, with value e_1 if condition c is true and e_2 otherwise):

$$sum = s(1, NIL)$$

where function s is defined as

$$s(i, list) = \textbf{if } i > m \textbf{ then } 0 \textbf{ else } \sum_{j=a_i}^{b_i} ss(i, cons(j, list))$$

and ss as

$$ss(i, list) = [\textbf{if } cond(list) \textbf{ then } f(list) \textbf{ else } 0] + s(i+1, list)$$

If *cond* satisfies the prefix condition, *ss* can be written more simply as

$$\textbf{if } cond(list) \textbf{ then } f(list)$$
$$\textbf{else } s(i+1, list)$$

We shall not give a formal proof of these formulae, which are easy to justify by considering the way the tree is traversed.

In the case of our example, f itself has a recursive definition, so that the definition for the expected cost *expected* can be written as

$$expected = expect(1, R)$$

where

$$expect(i, x) = \textbf{if } i > m \textbf{ then } 0$$
$$\textbf{else } \sum_{j=1}^{t_i} P_{i,j}\, exp\,(i, j, x)$$

and

$$exp(i, j, x) = \textbf{if } x \leqslant r_{i,j} \textbf{ then } C_i r_{i,j}$$
$$\textbf{else if } expect(i+1, x - r_{i,j}) > 0 \textbf{ then } C_i r_{i,j} + expect(i+1, x - r_{i,j})$$
$$\textbf{else } 0$$

PROGRAMMING THE METHOD IN FORTRAN

In a language allowing recursion, the above formulae are readily translated into mutually recursive function definitions (the $\sum$s being replaced by loops in ALGOL, PASCAL, PL/I, etc, or by further recursive definitions in LISP). In FORTRAN, recursion must be explicitly programmed using a stack. More precisely:

—mutually recursive function definitions are merged into a single recursive definition;

—a variable is added to control the nesting level of recursive calls. Here it already exists, *viz. i.*;

—each variable, function parameter of function result is replaced by an array, and each array by an array with one more dimension; the extra index will indicate the recursion level;

—each recursive call is replaced by instructions incrementing the recursion level, initializing array elements which represent function arguments, and then branching to the beginning of the function;

—each function return is replaced by instructions decrementing the recursion level, updating the stack element representing the function result, exiting if the recursion level becomes 0, otherwise leading to the instruction which follows the last recursive call. In general, return labels would have to be stacked (in an array of integers), and function return would be implemented as a computed GOTO; here this will not be necessary since all functions are called from exactly one location.

The application of these rules, with some obvious simplifications, leads to the programs below for the general case (program 1) and for our example case (program 2). F (the function f, which has been assumed to be of type REAL), and COND (the predicate *cond*, of type LOGICAL) have been treated as external FUNCTION subprograms. The second program is more efficient since it uses the prefix condition.

The use of recursion in programming and the methods for transforming recursive programs into iterative ones are further discussed in Reference 3.

ACKNOWLEDGEMENT

We would like to thank Mr. Michel Gondran for helping to render more accessible a more mathematically-oriented first version of this paper.

REFERENCES

1. A. J. Guttmann, 'Multi-dimensional summations in FORTRAN', *Software–Practice and Experience*, **6**, 221–224 (1976).
2. B. Meyer, 'Un problème récursif', *Bulletin du Centre de Calcul des Etudes et Recherches*, **40**, EDF, Clamart (1976).
3. B. Meyer and C. Baudoin, *Méthodes de Programmation*, Paris, Eyrolles, to appear, 1978.

APPENDIX

Programs

```
      REAL FUNCTION SUM (M, F, COND, A, B, LIST)
C  **  COMPUTE THE SUM OF FUNCTION F OVER THE SET OF I-TUPLES (I < = M) ***
C  **  WITH CHARACTERISTIC FUNCTION COND AND LIMITS A(I), B(I)           ***
      INTEGER M, A(M), B(M), LIST(M)
      EXTERNAL F, COND
      REAL F
      LOGICAL COND
C  **  ARRAY LIST WILL REPRESENT THE RECURSION STACK
C  **  COND(LIST, I) IS COND APPLIED TO THE FIRST I ELEMENTS OF ARRAY LIST
C  **  F(LIST, I)       IS F       APPLIED TO THE FIRST I ELEMENTS OF ARRAY LIST
C
      SUM = 0
      I = 0
C
C     ***** DEPTH-FIRST DESCENT *****
100   IF (I.EQ.M) GOTO 200
          I = I+1
          LIST(I) = A(I)
          GOTO 100
C
C     ***** BACKTRACK *****
200   CONTINUE
C         ***** HERE I < = M AND LIST(I) < = B(I) *****
          IF (COND(LIST, I)) SUM = SUM+F(LIST, I)
          LIST(I) = LIST(I)+1
          IF (LIST(I).LE.B(I)) GOTO 100
              I = I-1
              IF (I.GT.0) GOTO 200
C
1000  RETURN
      END
```

```
      REAL FUNCTION EXPCOS (M, R, T, P, COST, POWER, LIST, EXPECT, REMAIN)
C     *****  COMPUTE THE EXPECTED COST OF OPERATING A THERMAL      *****
C     *****  POWER SYSTEM SO AS TO SATISFY POWER REQUIREMENT R     *****
C
      INTEGER M, T, LIST(M)
      REAL R, P, COST, POWER, EXPECT(M), REMAIN(M)
      INTEGER INDEX
      EXTERNAL R, T, P, COST
C
C     *****  R, T, P, COST, POWER ARE THE PARAMETERS OF THE PROBLEM    *****
C     *****  THEY ARE ASSUMED HERE TO BE FUNCTIONS SUBPROGRAMS,        *****
C     *****  BUT COULD BE ARRAYS AS WELL.                             *****
C     *****      FOR 1 < = I < = M, T(I) IS THE "SIZE" OF PLANT I,    *****
C     *****                         AND COST(I) IS ITS OPERATING COST.*****
C     *****      FOR 1 < = I < = M AND 1 < = J < = T(I), P(I, J) IS THE*****
C     *****      PROBABILITY THAT UNIT J OF PLANT I IS STARTED,        *****
C     *****      AND POWER(I, J) IS THE POWER IT PRODUCES.             *****
C     *****                                                           *****
C
C     *****  ARRAYS LIST, EXPECT AND REMAIN REPRESENT THE STACK;       *****
C     *****  THEY CORRESPOND TO NAMES J, EXPECT AND X IN THE           *****
C     *****  RECURSIVE FORMULATION                                    *****
C
      INTEGER INDEX
C
      I = 1
      LIST(1) = 1
      INDEX = 1
      EXPECT(1) = 0
C
C     *****  DEPTH-FIRST DESCENT  *****
100   CONTINUE
C           THE SPECIAL CASE I = 1 BELOW IS MOTIVATED BY THE FACT
C           THAT ARRAY "REMAIN" CANNOT BE INDEXED BY 0
            IF (I.EQ.1) REMAIN(I) = R-POWER(I, INDEX)
            IF (I.GT.1) REMAIN(I) = REMAIN(I-1)-POWER(I, INDEX)
            IF (REMAIN(I).LE.0) GOTO 150
            IF (I.EQ.M) GOTO 200
                  I = I+1
                  LIST(I) = 1
                  INDEX = 1
                  EXPECT(I) = 0
                  GOTO 100
C           *****  UPDATE EXPECTED VALUE  *****
150         EXPECT(I) = EXPECT(I)+P(I)*COST(I)*POWER(I, INDEX)
C           *****  BACKTRACK  *****
200         CONTINUE
                  INDEX = INDEX+1
                  LIST(I) = INDEX
                  IF (INDEX.LE.T(I)) GOTO 100
                        I = I-1
                        IF (I.EQ.0) GOTO 1000
                              IF (EXPECT(I+1).NE.0) EXPECT(I) =
     1                          EXPECT(I)+P(I, INDEX)*
     2                          (EXPECT(I+1)+COST(I)*POWER(I, INDEX))
                              GOTO 200
C
1000  EXPCOS = EXPECT(1)
      RETURN
      END
```