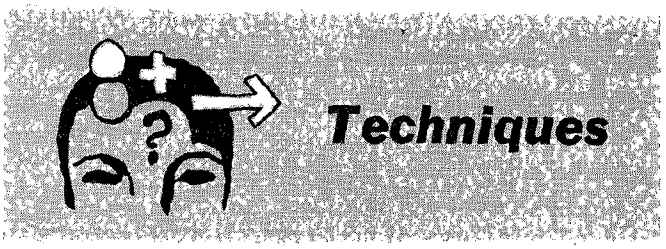
running time for the longest problem (6×10 box) is about 10 minutes.

A program using the same technique was written to solve the hexiamond (cf. *di*amond) problem. Hexiamonds consist of six contiguous equal equilateral triangles and are 12 in number. It was found, for example, that there is no way to place them in a rhomboidal box 3×12.

APPENDIX. Listing of the Program

```
*
        COUNT   81      THIS PROGRAM GENERATES 765 INSTRUCTIONS.
        TITLE           IT SOLVES THE PENTOMINO PROBLEM BY MACRO RECURSION.
*
TEST    MACRO   C,NEXT,XX CONSIDER SQUARE C (RELATIVE TO LEAD).
        ZET     ARENA-C,1 SEE WHETHER IT IS EMPTY.
        IFF     K-1,0,0   (FOR OTHER THAN LEAD (C NOT 0), DO NEXT ITEM.)
        TRA     XX+1      IF IT IS NOT EMPTY, TRY NEXT CHOICE FOR C.
        IFF     K-1,0,1   (FOR LEAD (C IS 0), DO NEXT ITEM.)
        TCO               DO AS DESCRIBED IN TCO MACRO.
        STQ     ARENA-C,1 INDICATE SQUARE C TO BE OCCUPIED.
K       SET     K+1       (ADVANCE COUNT OF DEPTH OF MACRO NESTING.)
        IRP     NEXT      (ITERATE ON ITEMS IN NEXT, WHICH IS A LIST.)
        IFF     K-5,0,0   (FOR DEPTH OF NESTING NOT 5, DO NEXT ITEM.)
        TEST    NEXT      DO AS IN TEST MACRO WITH ARGUMENTS FROM NEXT.
        IFF     K-5,0,1   (FOR DEPTH OF NESTING EQUAL 5, DO NEXT ITEM.)
        FTEST   NEXT      DO AS IN FTEST MACRO WITH ARGUMENTS FROM NEXT.
        IRP               (CEASE ITERATING ON ITEMS IN NEXT.)
K       SET     K-1       (REDUCE COUNT OF DEPTH OF MACRO NESTING.)
XX      STZ     ARENA-C,1 RESET SQUARE C TO EMPTY.
TEST    END               SQUARE C AND ITEMS IN NEXT NOW ALL CONSIDERED.
*
TCO     MACRO             LEAD SQUARE HAS BEEN TESTED.
        TXI     STEM,1,1  IF NOT EMPTY, TRY NEXT SQUARE AS LEAD.
        PXD     ,1        IF EMPTY, RECORD ITS NUMBER
        STD     LEAD,2      AT PROPER PLACE IN LEAD ARRAY, AND
TCO     END               CONTINUE.
*
FTEST   MACRO   C,M       CONSIDER SQUARE C AND PENTOMINO M.
        ZET     ARENA-C,1 SEE WHETHER SQUARE C IS EMPTY.
        TRA     *+9       IF NOT, TRY NEXT CHOICE FOR C. OTHERWISE,
        ZET     MINO-M    SEE WHETHER PENTOMINO M IS UNUSED.
        TRA     *+7       IF NOT, TRY NEXT CHOICE FOR C AND M. OTHERWISE,
        STQ     ARENA-C,1 INDICATE SQUARE C TO BE OCCUPIED.
        STQ     MIND-M    INDICATE PENTOMINO M TO BE USED.
        STL     PATT,2    SAVE CURRENT POINT IN SEARCH OF PATTERNS, AND
        TXI     ROOT,2,1  PROCEED TO TRY TO PLACE ANOTHER PATTERN.
        STZ     ARENA-C,1 RESET SQUARE C TO EMPTY.
        STZ     MINO-M    RESET PENTOMINO M TO UNUSED STATUS.
FTEST   END               SQUARE C AND PENTOMINO M HAVE BEEN CONSIDERED.
*
        CALL    INPUT,ARENA,MINO    INITIALIZE ARENA AND MINO ARRAYS.
        AXT     1,1           INDEX 1 = NO. OF CURRENT LEAD SQUARE.
        AXT     1,2           INDEX 2 = NO. OF PATTERNS ALREADY PLACED + 1.
        AXC     1,4           INDEX 4 = -1 ALWAYS.
K       SET     1             (K = DEPTH OF MACRO NESTING.)
*
ROOT    TXH     SOLVED,2,12 SOLUTION FOUND IF 12 PATTERNS ALREADY PLACED.
        PXD     ,2          SET DECREMENT
        XCA                   OF MQ TO NEW VALUE OF INDEX 2.
*
STEM    TEST    00((01((02($        BEGIN SEARCH OF PATTERNS.
        ETC     (03((04,02)(16,03)(17,11)(18,11)(19,03)))$
        ETC     (16((15,04)(17,05)(18,07)(32,08)))$
        ETC     (17((18,05)(33,06)))$
        ETC     (18((19,04)(34,08)))))(16($
        ETC     (15((14,04)(31,09)(17,05)(32,01)))$
        ETC     (17((18,05)(32,05)(33,05)))$
        ETC     (32((31,12)(33,07)(48,03)))))(17($
        ETC     (18((19,04)(33,01)(34,09)))$
        ETC     (33((32,07)(34,12)(49,03)))))(16((15($
        ETC     (14((13,03)(30,12)(31,01)(17,11)(32,06)))$
        ETC     (31((30,09)(17,01)(32,05)(47,04)))$
        ETC     (17((18,11)(32,10)(33,01)))$
        ETC     (32((33,01)(48,11)))))(17$
        ETC     (18((02,07)(19,03)(32,06)(33,01)(34,12)))$
        ETC     (32((31,01)(33,05)(48,11)))$
        ETC     (33((34,09)(49,04)))))(32($
        ETC     (31((30,08)(47,04)(33,06)(48,11)))$
        ETC     (33((34,08)(48,11)(49,04)))$
        ETC     (48((47,03)(49,03)(64,02)))))))
*
        TIX     STEMX,2,1   ALL PATTERNS TRIED WITH CURRENT LEAD, SO
        CALL    FINISH      IF NO PATTERNS ALREADY PLACED, EXIT, ELSE
STEMX   CAL     LEAD,2      RECALL PREVIOUS NUMBER OF LEAD SQUARE,
        PDX     ,1          RESTORE IT TO STATUS AS CURRENT LEAD,
        PXD     ,2          RESET DECREMENT
        XCA                   OF MQ TO REFLECT REDUCTION IN INDEX 2, AND
        TRA*    PATT,2      RETURN TO EXAMINATION OF PATTERNS.
*
SOLVED  CALL    OUTPUT,ARENA,MINO   SOLUTION OBTAINED, SO OUTPUT IT,
        AXC     1,4         RESET INDEX 4 TO -1, AND
        TXI     STEMX,2,-1  RETURN TO SEARCH FOR NEW 12TH PATTERN.
*
ARENA   BES     512     THIS ARRAY CORRESPONDS TO THE SQUARES.
MINO    BES     12      THIS ARRAY CORRESPONDS TO THE PENTOMINOS.
LEAD    BES     12      THIS IS A PUSHDOWN LIST OF LEAD SQUARES.
        OUP     1,12    (SINCE INDEX 4 = -1, INDIRECT ADDRESSING THRU THE
        PZE     ,4        PATT ARRAY TRANSFERS TO STORED LOCATION + 1.)
PATT    BES     0       THIS IS A PUSHDOWN LIST OF PATTERNS.
        END             SUBROUTINES REQUIRED ARE INPUT, OUTPUT, FINISH.
*
```

RECEIVED FEBRUARY, 1965



R. M. GRAHAM, Editor

A Fast Storage Allocator

KENNETH C. KNOWLTON
Bell Telephone Laboratories, Inc., Murray Hill, N. J.

A fast storage bookkeeping method is described which is particularly appropriate for list-structure operations and other situations involving many sizes of blocks that are fixed in size and location. This scheme, used in the LLLLLL or L[6] (Bell Telephone Laboratories Low-Level List Language), makes available blocks of computer registers in several different sizes: the smaller blocks are obtained by successively splitting larger ones in half, and the larger blocks are reconstituted if and when their parts are simultaneously free.

Introduction

A storage allocating and bookkeeping mechanism is described which makes available blocks of computer registers in several different sizes, the smaller blocks being obtained by successively splitting larger ones in half, and the larger blocks being reconstituted if and when their parts are simultaneously free. Each block contains a few bits of information indicating its size and whether it is presently free or in use; the rest of the block may be formatted as the user wishes. The operations involved in obtaining blocks from and returning them to the free storage lists are very fast, making this scheme particularly appropriate for list structure operations and for other situations involving many sizes of blocks which are fixed in size and location. This is in fact the storage bookkeeping method used in the Bell Telephone Laboratories Low-Level List Language[1] (LLLLLL or L[6], pronounced "L-sixth").

Free Storage Lists

The method to be described is illustrated by an example in which block sizes are one, two, four and eight machine words.[2] Figure 1a depicts the original layout of storage,

[1] KNOWLTON, K. C. A programmer's description of LLLLLL, Bell Telephone Laboratories low-level list language. Bell Telephone Laboratories, Inc., Mar. 1965. This language provides for blocks of the following lengths: 1, 2, 4, 8, 16, 32, 64 and 128 words.

[2] Upon completing this paper the author has learned that a similar storage handling system is used in SIMSCRIPT where block sizes are in fact one, two, four and eight words. For a description of the language, see MARKOWITZ, H. M., HAUSNER, B., and KARR, H. W. *SIMSCRIPT—A Simulation Programming Language*. Prentice-Hall, Englewood Cliffs, N. J., 1963.

with the entire storage area divided into blocks of the largest sizes. These blocks are initially arranged in a doubly connected loop of which one element is the system register. The register's pointers point to the beginning and end of the list of blocks, as shown in Figure 1b. Pointers for such purposes could be either addresses of the first words of blocks pointed to or the 2's complements of these addresses. In this example addresses will be assumed.

There is a free storage list for each size of block. Since the lists of smaller blocks are initially null the system pointers of each of these lists point "to each other," that is, to the same register in which both are located. As is customary with this arrangement of pointers a block is removed from its list by making its predecessor and successor point to each other; a block is inserted at the beginning of the list by appropriate setting of the four pointers involved. ("Special cases," such as removing the only remaining block or inserting a block when none is there, do not require special treatment because the system pointers occupy the same locations within their words as do the other pointers.)

The storage bookkeeper being illustrated has the following points of entry from the outside: one for setting up in a prescribed region the initial list of largest-size blocks, four for obtaining blocks of the four different sizes and one for returning a block to free storage. (The bookkeeper looks at the block's size indicator to determine what to do with it).

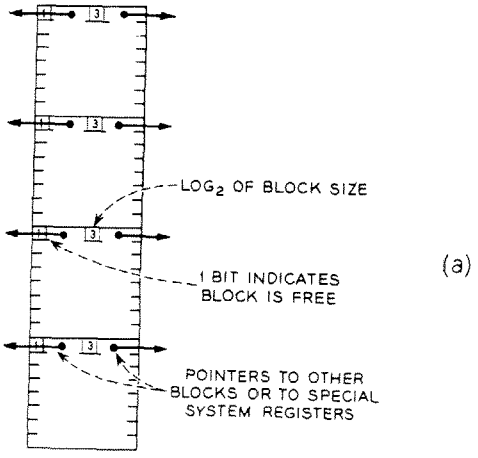


FIG. 1a. Example of the original layout of free storage: four 8-word blocks.

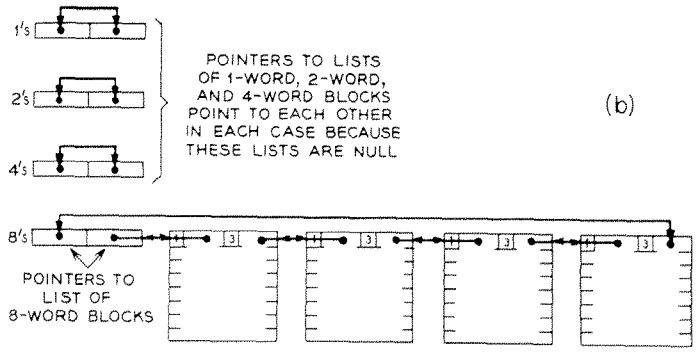


FIG. 1b. Original arrangement of free storage lists for condition shown in Figure 1a.

Splitting of Blocks

It may frequently happen that a block of the desired size is not directly available from the list for blocks of that size. This would be noted by system pointers pointing to each other or by a zero count in an optional counter for the current number of blocks of this size. In this case a block of the next larger size is obtained and split in two; one of its halves is "returned" to free storage and the other half is presented as the requested block. The process works recursively up to the largest size of block. Thus given the situation depicted in Figure 1 and a request for a one-word block the resulting free storage lists would contain three eight-word blocks, and one block each of sizes one, two and four words. A number of additional operations for obtaining and returning blocks of different sizes could, for example, lead to the condition illustrated in Figure 2.

The procedure is clearly one which divides available storage into blocks of different sizes exactly in proportion to the requirements of the current computer run. One difficulty not yet discussed is the possibility that many small blocks are required early in the run and are subsequently returned and that later many larger blocks are requested, thus requiring the reconstitution of large blocks from their individually used and freed parts.

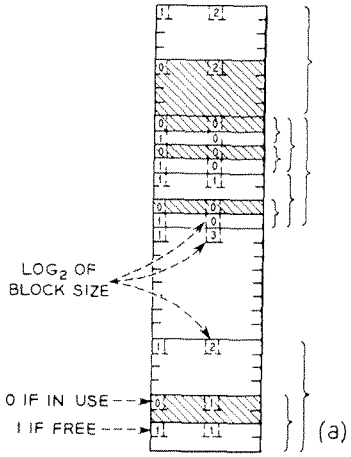


FIG. 2a. Possible later condition of the storage area shown in Figure 1a showing blocks of various sizes in use (shaded) and free (clear). Braces show how three of the original 8-word blocks have been successively subdivided to form the present blocks.

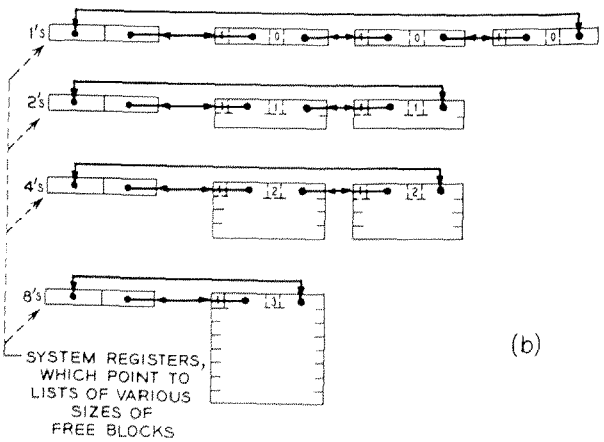


FIG. 2b. Arrangement of free storage lists for condition shown in Figure 2a.

Recombination of Blocks

The system under discussion does in fact recombine blocks according to these rules: When a block is being returned to free storage, a check is made to determine whether its mate (the block of the same size from which it was previously split) is presently in free storage and not further subdivided. If so the mate is removed from its list (this is why backward as well as forward pointers are required in the free storage lists), the two blocks are combined and the resulting block is placed on the list for blocks of this next larger size. This operation can be done quickly since a block's mate can be found quickly (it is immediately above or below—the question is which). One way to do this is simply to complement the nth order bit of the 2^n-block's address relative to the beginning of the storage region. Thus if the relative address of the beginning of a two-block ends in binary . . .011001, then the relative address of the beginning of its mate ends in . . .011011. Similarly if the relative address of a four-word block ends in . . .010110 then the relative address of its mate ends in . . .010010.

It is worth noting that a similar procedure can be used, for whatever purpose, to find the beginning of the block *above* a certain block in storage in the sense implied by Figure 2a. (Finding the block below it is trivial.) The method depends again upon the regular way in which the original blocks have been subdivided and upon the information as to size always found within each block.

Recombination, like splitting, is recursive. Thus in Figure 2a if the lowest two-word block in use (lowest shaded block) is returned to free storage, not only will it be combined with the two-word block below it but the resulting four-word block will also be combined with the free four-word block above it, thus reconstituting one of the original eight-word blocks.

Heuristics

In the interest of further improving the speed of the system it may not be desired to recombine free blocks every time recombination is possible. The system may be designed to use a simple strategy to decide when to try to recombine: e.g., when there are already a certain number of blocks of the size being returned, or when there are fewer than the desired minimum number of blocks of the largest size or when some other simple function of these numbers is satisfied. The precise form of the heuristic used to govern recombination will depend upon how early the system should begin to anticipate trouble which could result from a proportionally increased demand for the larger blocks.

RECEIVED JULY, 1965

A Note on the Use of a Digital Computer for Doing Tedious Algebra and Programming

GEORGE VERONIS
Massachusetts Institute of Technology, Cambridge, Mass.

A special purpose compiler was written with FORTRAN II language and made possible the writing of very long programs by the computer. The procedure is based on a straightforward use of FORMAT statements for generating machine-written programs.

1. Introduction

All computer users are familiar with the advantages afforded by the use of compilers such as FORTRAN II for the writing of programs. It is also possible for individual users to write simple, special purpose compilers which can turn tasks of extreme drudgery into straightforward programming tasks of relatively short duration. In fact, only a moderate amount of experience and facility with the use of FORTRAN is needed to write such a special purpose compiler. An example of this is presented below. The method is both straightforward and simple. It is reported here primarily because the effort was greeted with skepticism by a surprisingly large number of professional programmers. Yet it was possible to use the special purpose compiler to write programs which would have required many weeks of drudgery if they were to be done by hand.

The particular case for which the procedure was carried out involved a mathematical problem that was reduced to a system of $M \times N$ simultaneous ordinary differential equations of the form

$$\begin{aligned}\frac{dP_{rs}}{dt} = \frac{1}{r^2+s^2}\Bigg\{ & \sum_{n=1}^{r}\sum_{m=1}^{s}(n^2+m^2)(ns-mr)P_{nm}P_{r-n,s-m} \\ & + \sum_{n=1}^{N-r}\sum_{m=1}^{M-s}[(2nr+r^2-2sm-m^2) \\ & \qquad \cdot(ns+rs+rm)P_{n+r,m}P_{n,m+s} \\ & - (2nr+2ms+r^2+s^2)(mr-ns)P_{nm}P_{r+n,m+s}] \\ & + \sum_{n=1}^{r}\sum_{m=1}^{M-s}[(n^2+(s+m)^2)(mr+sr-sn) \\ & \qquad \cdot P_{n,s+m}P_{r-n,m} \\ & - (n^2+m^2)(ns+mr)P_{nm}P_{r-n,s+m}] \\ & + \sum_{n=1}^{N-r}\sum_{m=1}^{s}[(n^2+m^2)(ns+mr)P_{nm}P_{r+n,s-m} \\ & - ((n+r)^2+m^2)(ns+rs-rm) \\ & \qquad \cdot P_{n+r,m}P_{n,s-m}]\Bigg\}\end{aligned} \tag{1}$$

This research was supported by the Office of Naval Research under contract number N-ONR-2196.