ITEM NUMBER 6309-0164 (T.I.E.)
24 pages

DATE July 29, 1963

AUTHOR Edward C. Knapp, Jr.

TITLE TRUE BINARY TABLE SEARCH FOR THE IBM 1400 SERIES

SOURCE IBM CORPORATION
205 Whitney Avenue
New Haven, Connecticut

6309 0164

DISTRIBUTED BY
THE PROGRAM INFORMATION DEPARTMENT (TIE)
IBM CORP.
112 EAST POST ROAD
WHITE PLAINS, NY

Table of Contents

TITLE: True Binary Table Search

AUTHOR: Edward C. Knapp

DATE: July 24, 1963

DIRECT INQUIRIES TO: Edward C. Knapp
IBM Corporation
205 Whitney Avenue
New Haven 10, Connecticut

ABSTRACT: This paper describes a new theory of a true binary table search that may be used on any size sequential table. Originally developed for the IBM 1410, it is most attractive on 1400 series and other computers not equipped with the table look-up feature. This technique proves to be superior in both speed and memory requirements to previously used programmed table look-up routines.

213-4

Why the True Binary Search is Needed

It is a rare computer program which does not at some point search through an ordered table or group of records to find a number or name that matches some input. We are confronted with very similar problems in looking up an electricity rate, sorting tape records internally, converting codes, finding a disk record within a blocked track, distributing charges to general ledger accounts, or searching a symbol table.

Table look-up instructions with which some computers are equipped generally provide a simple and efficient method of accomplishing such searches. However, they require that certain rules be observed, such as the placement of word marks, which may not be desirable in all cases. Then too, the straight table look-up becomes very time-consuming when directed at large tables or when long functions must be interspersed between the table arguments.

Computers such as the IBM 1401, 1440, 1460, and 1620 do not possess any table look-up commands, but are frequently called upon to perform these functions with whatever instructions are available. Programmers usually take the most straight-forward approach and search a table item by item by means of indexing or address modification. This can be costly on large volume jobs where the operation must be done repeatedly. Alternative methods, used where timing is the main consideration, usually prove to be rather complicated and tend to use a large amount of memory for instructions.

The true binary search described in this paper was developed by the author for use on the IBM 1410 in a situation where it was impossible to include the word marks needed by the table look-up instruction. However, its greatest application should prove to be for the computers mentioned above which do not have table look-up ability. It will completely search any size sequential table or group of records in a minimum number of comparisons, but does not require a great deal of storage for the program itself. Once understood, it is found to be quite straight-forward and easily adapted to many table search operations.

Theory of Binary Search

Using a table that is either in ascending or descending sequence, it is possible to compare a search argument against the center table argument. If they are equal, the search is already finished. Otherwise the result of the comparison tells in which half of the table the desired argument may be found. A second comparison at the center of one of the halves can further tell which quarter of the original table might contain it. This procedure can be repeated as long as it is possible to subdivide whatever portion of the table remains.

Obviously a table that can be repeatedly divided in half until only one logical entry is left must in itself be related to some power of 2. It must in fact contain a total of entries equal to one less than some power of 2 in order to simulate a "look-up equal" operation and exactly some power of 2 for "look-up equal-high" or "look-up equal-low." This distinction will be explained later in this paper. For the moment we shall concentrate on the search for equal only so that this topic may be followed through to conclusion.

To illustrate the application of a binary table, let us refer to the following sample table:

Position in Table	Argument	Function
1	015	9463001
2	027	1004076
3	066	3472300
4	094	6875679
5	123	4221842
6	148	3884468
7	159	5123779
8	177	6897212
9	200	2011897
10	251	3675774
11	283	2001480
12	694	7581531
13	733	0175000
14	746	6361792
15	999	*******

213-6

The table consists of 15 entries in ascending sequence. Each entry contains ten digits, three for the argument which is the field against which we must make our comparisons and seven for the function. The other element of our problem is the search argument which must match exactly with one of the table arguments. The object of the search is, of course, to extract the function from the table which lies to the right of the matching table argument.

To search this table by the binary method, the program must compare the search argument against the table argument of the eighth entry. If an equal condition results, the desired item has been found and the search terminates. However, if the search argument is low, the item must be among entries 1 - 7 of the table, and if high, it has to be in the upper seven entries (9 - 15). The next comparison is made on the item which is the next lower power of two entries away from the previously compared item. Thus, we must look at entry 4 (low) or entry 12 (high). Successive comparisons are then made which always reduce the number of possibilities by half until only one entry remains. If that entry is not equal to the search argument, it means that the argument is not in the table.

The course taken by the search is best demonstrated by using an actual input argument such as 123. This is initially compared against item #8 (177) and found to be low. The next comparison against the center item of the lower half of the table, item #4 (094), results in a high condition causing the search to move upward to item #6 (148). The low indication at this point means that only one possibility remains, item #5 (123), which in this case satisfies the equal condition desired.

Had the input search argument been a number like 105 which does not exist in the table, the program would have followed the same course, but would have given a low result on the final comparison. Since it had previously been found to be higher than item #4, this low result means that the search argument falls somewhere between items #4 and #5.

To completely search a 15 item table such as the one on the previous page requires a maximum of only four comparisons. Note that the average number of comparisons is somewhat below this because if an equal condition results at some earlier point, no further comparisons are necessary.

It is easy to see that as a binary table increases in size to one item less than higher powers of 2, the number of comparisons needed for a complete search becomes lower in relation to the size of the table. Thus the following numbers of iterations are all that are needed for various larger tables:

No. of Items In Table	Maximum No. of Comparisons
31	5
63	6
127	7
255	8
511	9
1023	10

The essential value of the binary table lies in the fact that it is perfectly symmetrical. Each successive comparison must move to a point higher or lower on the table which is exactly half the distance travelled by the previous comparison. When this distance has been reduced to the length of one item, the search is completed.

This type of table organization lends itself to computer programming in that a simple loop containing just one compare instruction, one of whose addresses is continually modified by the lengths noted above, can perform the whole search. The key to this loop is a small subsidiary table of values containing an entry for each iteration needed equal to the amount the table address to be compared against must be incremented or decremented at that point of the search. It must terminate with some indicator which tells the program that the last iteration has been completed.

For the 15-item table described above the subsidiary table would look like this:

4 X L	=	40
2 X L	=	20
1 X L	=	10
End	=	**

L equals the length of the table argument plus the function which total 10 digits in the example. After the initial comparison has been made at the table's center, the value 40 is added to the compare address if the result were high and subtracted if low. In this manner the programmed loop can accomplish the search described previously. The program is actually controlled by the subsidiary table which means that merely by changing its values the same program can operate on tables with different sized entries or, by adding more values at the beginning (80, 160, etc.), upon larger binary tables.

For 1401, 2 subsidiary tables are needed. The second would be 16K compliments (or 4K) as you cannot subtract from addresses.

R Mann

213-9

Binary Theory Applied to All Tables

At this point the reader may be wondering what practical application the binary search can have, since it appears to require tables of only certain sizes which of course are not often found in actual practice. The salient fact here is that any sequential table of any size can be thought of as being made up of two overlapping binary tables of the next lower power of two minus one. Consider the following table of 25 items which is an extension of the binary table of 15 items described in the previous section:

	Position in Table		Argument	Function
	1		015	9463001
	2		027	1004076
	3		066	etc.
	4		094	
	5		123	
	6		148	
Lower	7		159	
Binary	8		177	
Table	9		200	
	10		251	
	11		283	
	12		694	
	13		733	
	14		746	
	15		758	
	16		762	
	17	Upper	795	
	18	Binary	796	
	19	Table	811	
	20		853	
	21		866	
	22		904	
	23		913	
	24		957	
	25		999	

It can be seen that this table may be thought of as one 15 - item binary table of entries 1 - 15 and a second one consisting of entries 11-25. The only difference between searching this table and a straight binary table is that an initial comparison at the table's center (item #13 in this case) must force the program to search either the upper or lower table. The identical routine described in the previous section can successfully search this unsymmetrical table. The only change is in the subsidiary table which controls the operation. Here one additional value must be placed at the beginning which will modify the address at item #13 to go next to either items #8 or #18 (the center points of the two binary sub-tables). The subsidiary control table would then appear like this:

```
5 X L   =   50
4 X L   =   40
2 X L   =   20
1 X L   =   10
End     =   **
```

This method is valid for an equal search through any table having an odd number of entries. To handle an even number of entries requires a slight change because of the fact that the initial distances moved up or down after the first comparison would not be the same. This is accomplished by creating two subsidiary tables instead of one. The increment table is referred to if the result of a comparison is high, the decrement table if low. They would look like this if the original table were increased to 26 items:

Increment Table	Decrement Table
60	50
40	40
20	20
10	10
**	**

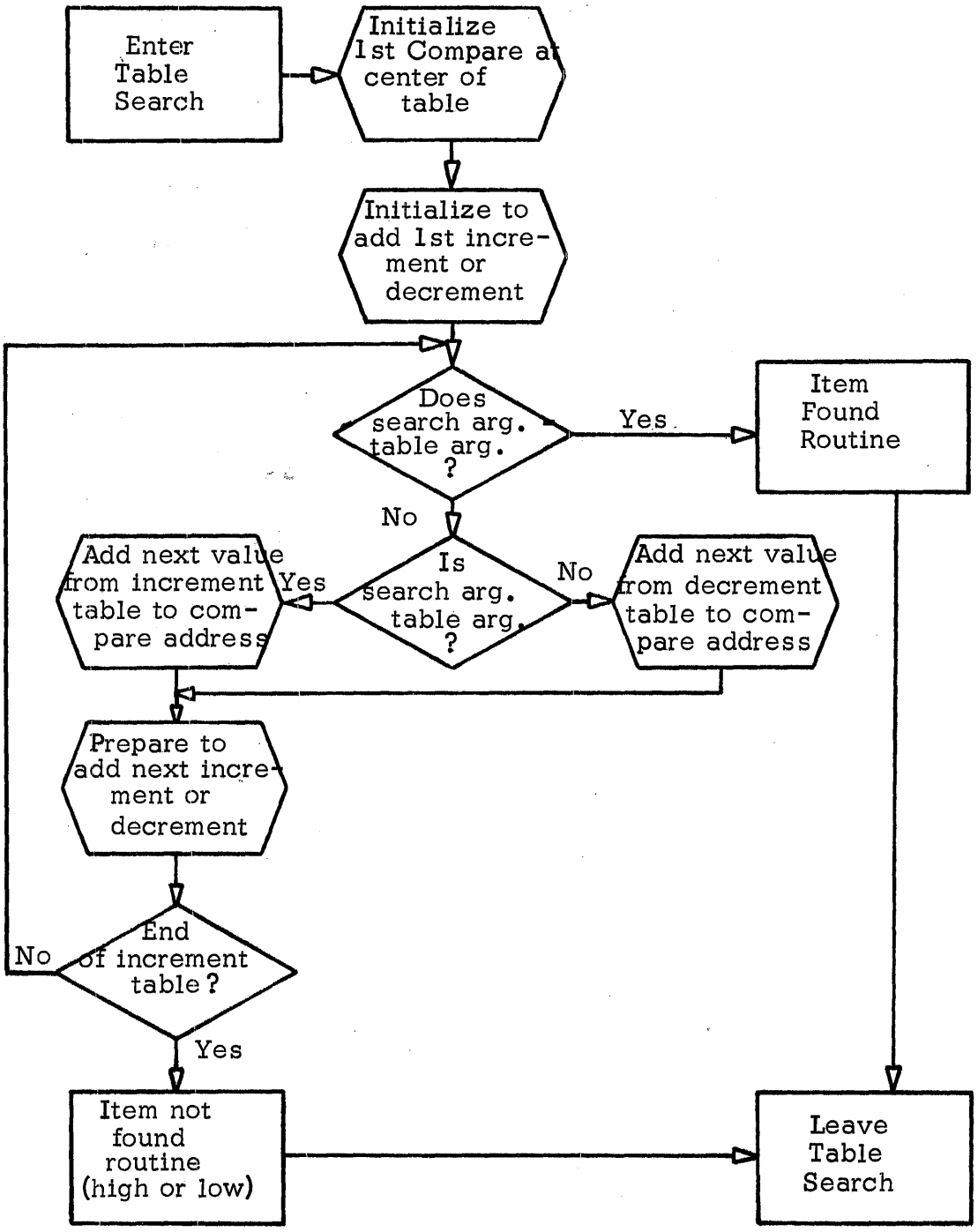
The initial comparison could still be made against item #13, but if the result were high, the next comparison should be made against item #19 which is now the center of the binary sub-table extending from item #12 to #26. Having the two subsidiary tables forces this sequence of operations.

Note that this example requires a maximum of five comparisons or, in other words, the number equal to the exponent of the next power of 2 which is greater than the number of items in the table.

Exactly the same routines can search for an equal argument in any size of table, the only change being to add more values to the subsidiary table (s). In this manner a table of as much as 2000 entries, for example, may be completely searched by comparing against only eleven of these entries or less.

Programming Example of Equal Search:

At this point it is appropriate to show a logic diagram and programming example of an equal binary search. This particular routine was written for and tested on the IBM 1410, but should operate unchanged on the IBM 1401, 1440, and 1460 computers provided they are equipped with index registers and the high-low-equal compare feature. On these machines having three-position addresses, care should be taken that if the total length of the table exceeds 999 characters, modulo 16 complements must be used for the negative values in the subsidiary table, "LOTBL". If it exceeds 1999 characters, the constant, "MIDPT" and possibly some of the positive values ("HITBL") must be given their three-position equivalents. The program steps need not be changed.



Logic Diagram of Binary Table Search

213-13

```
* BINARY SEARCH PROGRAMMING EXAMPLE FOR IBM 1410
START     ZA    MIDPT,X1     INITIALIZE TO MIDDLE ITEM IN TABLE
          ZA    &0,X2                      ZERO X2
COMP      C     TBARG&X1,INARG     COMPARE SEARCH ARGUMENT TO TABLE
          BH    UPPER           BRANCH TO GO HIGHER IN TABLE
          BE    FOUND           BRANCH IF EXACT MATCH
          A     LOTBL&X2,X1     GO LOWER IN TABLE
          B     ADDX2
UPPER     A     HITBL&X2,X1     GO HIGHER IN TABLE
ADDX2     A     &3,X2    UP X2 FOR NEXT VALUE IN SUBSIDIARY TABLES
          BW    COMP,HITBL-2&X2  TEST FOR END OF SUBSIDIARY TABLE
NOFIND    -----IF BRANCH ON WORD MARK NOT TAKEN, ITEM WAS NOT FOUND.
FOUND     -----BEGIN PROCESSING FOUND ITEM AT THIS POINT
* WHENEVER AN EQUAL ARGUMENT HAS BEEN FOUND, INDEX REGISTER 1
* CONTAINS THE HIGH-ORDER RELATIVE ADDRESS OF THE FOUND TABLE ITEM
* WHICH MAY BE PROCESSED AS REQUIRED
*
* DATA AREAS NEEDED
TABLE     DA    10X26       TABLE AREA OF 26 ITEMS OF 10 CHAR. EACH
TBARG           1,3         TABLE ARGUMENT
TFUNC           7,10        TABLE FUNCTION
INPUT     DA    1X80,G      SAMPLE INPUT AREA
INARG           17,19       SEARCH ARGUMENT
* SUBSIDIARY TABLES TO CONTROL AN EQUAL ONLY SEARCH OF A 26 ITEM
* TABLE CONTAINING 3 DIGIT ARGUMENTS AND 7 DIGIT FUNCTIONS.
*
LOTBL     DCW   -050        5 ITEMS LOWER IN TABLE
                -040        4
                -020        2
                -010        1
HITBL     DCW   &060        6 ITEMS HIGHER
                &040        4
                &020        2
                &010        1
          DC    @ @    LACK OF WORD MARK HERE TERMINATES SEARCH
* CONSTANT TO INITIALIZE X1 TO MIDDLE ITEM OF TABLE
MIDPT     DCW   &120
```

Autocoder Program Segment of Binary Table Search

Binary Search for Equal-High

To simulate a "look-up equal-high" machine operation in which the table argument must be found which either equals the search argument or is the next higher one to it requires only a small change in our interpretation of the table. The principal difference is that the ideal size table for this operation exactly equals some power of 2, rather than containing one item less. If we recall that the chief virtue of the binary table is the fact that it is perfectly symmetrical for purposes of the search, we can see that there is a subtle difference between an equal look-up and one for equal-high. In the search for equal, each comparison eliminates one possibility, i.e., the item just compared. The two remaining halves of the table or sub-table must be of equal lengths. For this reason the comparison at the center point of the table or a sub-table must be at the center of a group that is one less item than some power of 2. When the search is for equal or high, the item just compared is not necessarily eliminated from the search since a low result does not indicate that the desired table argument has been found until further comparisons have proven that the next lower item in the table is lower than the search argument.

Because the subsequent comparisons in the lower or upper halves of the table must take the identical course and the item just compared must still be taken into consideration as an entry in the lower half, the table itself must contain a number of items exactly equal to some power of 2.

Although this difference exists, the logic of the search remains unchanged. The only thing that must be altered is the points in the table where comparisons are made. This merely entails placing different values in the subsidiary control tables. The example of a 26 entry table discussed in the previous section was thought of as two overlapping binary tables of 15 items each (items 1-15 and 12-26) for an equal search. To perform a "look-up equal-high" requires it to be thought of as two tables of 16 items each (items 1-16 and 11-26). After a comparison against the center item resulting in the search argument being found low, the next comparison would be made against item #8 in either case. However, if the result were high, the equal search would next look at item #19 whereas the equal-high search would to to item #18.

The logic and programming example shown on pages 9 and 10 are almost identical for both an equal and equal-high search. The only difference is that the test for equal must branch to what was previously the "not found" routine. "Found" and "not found" are, therefore, synonymous for anything but an equal search. The reason for this is that there is no such thing as a "not found" condition following an equal-high search. The program must always find something, which generally means that the last item in the table should be a pad of 9's or some other unique indicator.

The subsidiary tables to control the equal-high search on the same 26 item table would appear like this:

```
LOTBL  DCW  - 050      HITBL  DCW  + 050
            - 040                  + 040
            - 020                  + 020
            - 010                  + 010
            - 000                  + 010
                                DC     @@
```

Note that these tables each contain one more entry than the corresponding tables for an equal search. The reason for this is that if on the last iteration of an equal-high search the search argument is found to be low, the proper "higher than" item is the one just compared, but if the search argument is high, the next higher item in the table is the one desired. Therefore, after the final comparison a low result leaves the table argument's address unchanged (LOTBL entry -000), but a high result causes it to be adjusted upward by one item (HITBL entry + 010).

If a search for an argument just lower than the highest item in the table is followed through, it will be observed that this final table argument is never actually compared in the example given. It is assumed to be the desired item if the search argument is greater than the next-to-last item. This is the reason that it is best to tag this last item with some special indicator (such as 9's) which the program can subsequently test.

The search for equal-high takes the same maximum number of comparisons as the search for equal. However, in many applications the equal condition, which is the only thing that can stop the iterations before maximum, occurs only rarely while looking for equal-high. Here the average number of comparisons per search would be nearly the same as the maximum.

213-16

Binary Search for Equal-Low

A slightly different situation develops when the search required must look for the table argument that is either equal to or the next lower value than the search argument. Here again the logic of the search is exactly the same as what has been previously discussed. Only the subsidiary control tables must be altered with no change in the program instructions.

The difference lies in the fact that comparisons to every level of a binary table or sub-table are made against the left-of-center item for an equal-high search and against the right-of-center item during an equal-low search. Another way of stating this is to say that in looking for an equal or high condition, a comparison to the highest item in the lower half of a binary table determines that if this table item is equal to or higher than the search argument, the desired item must be in the lower half of the table. If it is lower (search argument high), the desired item has to belong to the upper half. The situation is reversed in a search for equal-low where the significant point of comparison is at the lowest item in the upper half of the table segment. When the search argument is low or equal, the search continues in the upper half. If high, the search must shift down to the lower half. This reasoning is valid if the table contains entries totalling any power of 2 or 2 itself. In the latter case the comparison pinpoints the proper item which is what happens on the final iteration of the sample program.

To cause the same sample program to follow the desired sequence of comparisons for the equal-low search on the 26 item table, the following subsidiary control tables would be employed.

```
LOTBL   DCW   - 040      HITBL DCW + 060
              - 040                + 040
              - 020                + 020
              - 010                + 010
              - 010                + 000
                                   @ @
```

This example is based on the initial comparison still being made against item #13 (MIDPT equals + 120). The point of initial comparison is not fixed on one particular item, but on any of those that are part of the overlapping portion of the two binary sub-tables, provided the first increments in the subsidiary tables are adjusted accordingly. Thus, if MIDPT were made + 130 (to compare first against item #14), the first entries in the two subsidiary tables would have to be -050 and +050.

For the same reason that the equal-high search required a special entry in the highest (right-most) table position, the equal-low search needs it in the lowest (left-most) position. The program arrives at this entry without comparing against it if all other items have been found to be higher than the search argument. It can contain asterisks or some other indication that may be tested by the program.

Timing for an equal-low search is the same as for equal-high.

Tables in Descending Sequence

Tables arranged in a descending sequence may be searched by the same program merely by changing the points of comparison so that they are oriented toward the right end of the table in the manner that the ascending table's points are oriented toward the left end. When the result of any one comparison indicates that the next point should be higher in the table (at some higher table argument), the address of this point is arrived at by decrementing the current address. This means that if the initial point of comparison is moved one item to the right, just by changing the signs on the values in the increment and decrement subsidiary tables, a search of an ascending table can apply to the same size descending table.

If we continue with the sample table of 26 items used in previous illustrations, the three types of look-up when applied to descending tables would require the following subsidiary tables (MIDPT is +130 or item #14):

Look-Up Equal

```
LOTBL  DCW  + 050     HITBL  DCW - 060
            + 040                - 040
            + 020                - 020
            + 010                - 010
                                 DC   @@
```

Look-Up Equal-High

```
LOTBL  DCW  + 050     HITBL  DCW - 050
            + 040                - 040
            + 020                - 020
            + 010                - 010
            + 000                - 010
                                 DC   @@
```

Look-Up Equal-Low

```
LOTBL DCW   + 040     HITBL DCW  - 060
            + 040                - 040
            + 020                - 020
            + 010                - 010
            + 010                - 000
                                 DC   @@
```


213-19

How to Construct the Subsidiary Tables

Since the program sample shown on page 10 can perform any of the six searches covered in this paper, the only problem the user has is to calculate the values in the subsidiary tables. This is a four-step proposition consisting of the following:

1. Select initial point of comparison.
2. Determine the first value in both the increment and decrement tables.
3. Fill in the remaining values except the last.
4. Select the last value.

The initial point of comparison is generally at the midpoint of any table, although it may be against any of those items which fall in the overlapping portion of the two binary sub-tables of the next lower power of 2. Let us call this entry number M. From this, the value of the constant, MIDPT, can be calculated relative to zero. Where L = the length of each table entry:

$$MIDPT = L(M - 1)$$

The second points of comparison are the most critical because they are peculiar to the type of search performed. To do this let us call T the total number of items in the table. For an equal-high search, T must include the highest entry which is either the highest possible argument or a special indicator. The same is true of the lowest argument in an equal-low search. Next, figure the value of n which is the power of 2 that is the next lower to the total items, or in other words so that:

$$2^n < T < 2^n + 1$$

Having determined the values of M, L, T, and n, we can now find the items that may be compared against on the second iteration. Let us first deal with tables in ascending sequence. The following formula table shows how the two points are arrived at for the different types of search:

Second Points of Comparison for Ascending Tables

Type of Search	Item # After Low Result (P_1)	Item # After High Result (P_2)
Equal	2^{n-1}	$T - 2^{n-1} + 1$
Equal-High	2^{n-1}	$T - 2^{n-1}$
Equal-Low	$2^{n-1} + 1$	$T - 2^{n-1} + 1$

The above points may be called P. Thus, for example, if a table contains 53 items for an equal-low search, T= 53 and n=5 (i.e., $2^5 < 53 < 2^6$). When the result of the initial comparison is low, the second should be made at item P_1 where:

$$P_1 = 2^{5-1} + 1$$

$$P_1 = 17$$

If the result is high, it must be made at item #38.

$$P_2 = 53 - 2^{5-1} + 1$$

$$P_2 = 38$$

To obtain V_1, the value to be placed in the decrement table (LOTBL), and V_2, to be the first entry in the increment table (HITBL) merely subtract the value of M from the respective P and multiply by the length of each table item. Thus:

$$V_1 = L (P_1 - M)$$

$$V_2 = L (P_2 - M)$$

The same type of formula table for descending tables applies:

213-21

Second Points of Comparison for Descending Tables

Type of Search	Item # After Low Result (P_1)	Item # After High Result (P_2)
Equal	$T - 2^{n-1} + 1$	2^{n-1}
Equal-High	$T - 2^{n-1} + 1$	$2^{n-1} + 1$
Equal-Low	$T - 2^{n-1}$	2^{n-1}

The values of V_1 and V_2 are determined by the same equations shown above.

It can be seen that the values of V_1 and V_2 are functions of (a) the number of items in the table, (b) the length of each item, (c) the sequence of the table, (d) the type of search to be performed and (e) the position of the initial comparison. The assignment of specific values to V_1 and V_2 forces the search's second iteration to look at the center of some size binary table. For an equal only search this is at exactly the center item. When the look-up is for equal-high, it is the highest item of the binary table's lower half; for equal-low, the lowest item of the upper half.

After the second comparison the search assumes a regular pattern for every type of look-up and previous result. Therefore, these portions of the two subsidiary tables are identical except for the signs of the values. They follow this progression:

$$L(2^{n-2}),\ L(2^{n-3}),\ \ldots\ldots\ L(2^{0})$$

When dealing with an ascending table, all signs in the decrement (LOTBL) table are minus and in the increment (HITBL) table, plus. The signs are reversed for a descending table. Note that this portion of a subsidiary table always contains n-2 values.

At this point the subsidiary tables are complete for an equal only search where, in the event of a not-found condition, the programmer is not concerned with the next higher or lower items. All that must be added is the "search end indicator" which in the sample program is the position without a word mark (DC) following the increment table (HITBL).

For an equal-high or equal-low look-up the final element of a subsidiary table is added to the address of the very last item compared causing the program to "find" it or the item immediately to the right or left. If the item in the final comparison is the one desired, this element is zero. Otherwise it is plus or minus the item length (L) causing the search to end one item higher or lower in the table. This value is constant as follows for the different table sequences and types of search:

Final Subsidiary Table Values

Type of Search	Decrement (LOTBL)	Increment (HITBL)
Ascending Tables		
Equal	Not required*	Not required*
Equal-High	Zero	+ L
Equal-Low	- L	Zero
Descending Tables		
Equal	Not required*	Not required*
Equal-High	Zero	- L
Equal-Low	+ L	Zero

*Note: By including the same values required by equal-high or equal-low searches, a not found condition following an equal search can pinpoint the next higher or lower item respectively.

The final increment table item must be followed by a location not containing a word mark to stop the iterations in conformance with the sample program.

By following the above steps the user can easily adapt the sample standard search program to perform any of the usual table look-up operations upon any sequential table containing elements of a fixed length. Since the size of the area being searched is controlled by the size of the subsidiary tables, the latter may be modified by a program such as an internal sort to expand as the table (addresses of already sorted records) increases.

Conclusion

The true binary search will have wide application on any computer not equipped with table look-up instructions. On such a machine the only times some alternative method might be preferable would be in dealing with tables containing only a few items or if the frequency of "hits" is disproportionately large on a very small number of items.

The storage requirements of the program and subsidiary tables are only slightly greater than for the simplest type of indexed search. No other programmed table look-up can completely search an ordered table or group of records in as few iterations as the binary search. Most methods require a great deal more. The number of steps actually executed per iteration is hardly greater than by the usual, less efficient routines.

The binary search, of course, cannot operate upon non-sequential tables.

Even computers such as the 7070 and 1410 that possess table look-up commands can employ the binary search to great advantage under certain circumstances. It can outperform straight table look-up instructions on extremely large tables. Since the length of functions (which may be the information portion of records) interspersed between table arguments does not effect the timing of the binary search, it is superior to 1410 table look-up where these functions are large. As mentioned earlier, the sample search program needs no word marks whatsoever in the table itself whereas 1410 table look-up requires and allows them only in specific places.

Random access systems such as the IBM 1410-1301 can often economically use a single disk input-output area for several formats of records which must be searched. Word marks for one format may not be acceptable for another. A specific instance is in the case of a sequential file accessed by means of index tracks. Using the binary search both index and data tracks may be read into the same area without difficulty. The alternatives are either to reserve two large areas (which in this case are normally 2800 characters each) or to set and clear a great many word marks for the machine table look-up.

Every business or industry uses tables in one form or another. Any ordered group of records may in fact be thought of as a table. From searching through a simple list of city codes to compiling symbol tables to looking for a record on a disk track, tables are an integral part of data processing. The true binary search proves to be the most efficient method of performing these searches on the vast majority of IBM computers and can be extremely helpful on the rest.

213-26