ARRAY LOOKUP TECHNIQUES:
FROM SEQUENTIAL SEARCH TO KEY-INDEXING (PART 1)

Paul M. Dorfman
Citibank Universal Card Services
Jacksonville, FL

ABSTRACT

Arrays have a variety of uses in the SAS® language. In particular, they represent a powerful structure with direct addressability, ideal for quick data lookup. In this tutorial, we shall address the following array search techniques:

1. Sequential search -- the simplest lookup technique, which, however, may be preferable in many cases.
2. Binary search -- a universal, fast, and efficient method of searching an ordered array.
3. Interpolation search -- an extension of the binary search idea that delivers superior performance in the case of uniformly distributed keys.
4. Key-indexed search -- based on direct addressing rather than comparison of keys. Although it is confined to integer keys falling into a limited range, it has many uses and is fastest.

All lookup techniques presented in the tutorial are demonstrated using a real-life example of matching two files by a common key. The methods are compared to each other in terms of simplicity of implementation, area of applicability, resource efficiency, and performance.

INTRODUCTION

Table lookup is one of the most frequent operations performed in SAS data processing. It is no surprise, therefore, that SAS provides us with a rich collection of built-in searching techniques. Merging and SQL joins, formats and string searching functions, direct access to SAS data files and SAS indexes, to name a few, - all serve the purpose of looking up relevant data, explicitly or implicitly. In addition to these 4GL features, SAS Language incorporates data structures ideally lending themselves to programmatic implementations of just about any searching algorithm. Such structures are SAS arrays. Being 3GL by nature, arrays are not ready-to-go tools: Array-based table lookups have to be coded and tuned for performance. But, firstly, programming is fun! Secondly, such "sniper" approach is more flexible and often results in programs that search faster and use fewer resources than the "heavy artillery" does.

To make the discussion less abstract, we will consider a common task of matching two SAS datasets, one "small" and one "large", by a key variable. This way, we will also be able to see how different array lookup techniques compare to each other and to some of the "heavy artillery" methods. First, we will consider, from the simplest to the most elaborate, lookup schemes based on comparing the given key to the keys in the array - sequential, binary, and interpolation searches - and discuss the details of implementing them efficiently. At the end, we will do away with key comparisons altogether and instead consider key-indexing, which is the simplest form of table lookup based on direct addressing.

Throughout the tutorial, the terms *file, dataset*, *table*, and *observation*, *row*, *record*, will be used interchangeably.

THE SAMPLE PROBLEM

Suppose we have two unsorted SAS datasets, SMALL and LARGE, with N_SMALL and N_LARGE observations respectively. The small file has a *key variable* KEY assuming distinct values and some additional information stored in a variable S_SAT. The large file has a key variable KEY of the same type and length as in the small file, but not necessarily unique, and an extra variable L_SAT. The non-key variables S_SAT and L_SAT are usually termed *satellite information*.

Let us further imagine that we have to perform one of the most common tasks in data processing, namely, to *match* the files. In other words, we need to create a SAS dataset MATCH with only those records from the file LARGE whose keys are also present in the file SMALL and with the satellite information from the small dataset corresponding to the matching keys. We will also stipulate an additional condition, namely, that the matching must be achieved without sorting the file LARGE, for in a variety of real-world situations, massive sorting may be either impractical or undesirable. Another assumption we will make is that all the keys from the small file can be fit in the computer memory in the form of a SAS temporary array.

Leaving the large file unsorted rules out methods based on sequential matching, such as the MERGE or a pair of sequential-access SET statements. For the purposes of this tutorial, we will try to approach the problem using the following plan:

1. Store all the keys and satellites from SMALL in memory organized as some kind of lookup table.
2. For each record read from the large file, search the table for KEY coming with the record.
3. If search has been successful then output KEY and the relevant satellite information.

SAS offers three means of facilitating an in-memory table lookup:

1. Load the lookup keys into an array and invoke some mechanism of searching it.
2. Place the lookup keys in a user-defined format using CNTLIN= option.
3. Join the tables using PROC SQL.

Each of the three has its advantages and shortcomings. Format search is sure fast, but its memory consumption is

high, it may take a lot of time to compile a sizable format, and it may not release memory after the search is finished. SQL is the easiest to program, and if it decides to hash the keys from the smaller file in memory, it searches quite rapidly. However, we can only invite, not force, the optimizer to take this path by supplying a large BUFFERSIZE value.

"Some mechanism" of searching an array has to be programmed in a DATA step. Whilst this can be perceived as a drawback, it has the advantage of greater flexibility. Firstly, having lookup keys in an array, one can code virtually any existing search algorithm. Secondly, the very nature of the keys can be accounted for resulting in speed and efficiency gains no ready-to-go search recipe can approach.

Let us consider several facts related to SAS arrays significant from the standpoint of the task at hand:

- SAS arrays are sized at compilation time. That is, and very unfortunately, we cannot allocate an array on the fly using SAS variables to indicate its dimensions. Instead, we are forced to either guess the maximum number of array items, or determine this value beforehand and assign it to a macro variable. In our case, the lookup array should be able to accommodate all the keys from the small file, so we can predict its size, for instance, by reading the descriptor of the file SMALL:

```
DATA _NULL_;
   SET SMALL NOBS=SIZE;
   CALL SYMPUT
   ('SIZE',LEFT(PUT(SIZE,BEST.)));
   STOP;
RUN;
```

- There are several good reasons we should use *temporary* arrays to organize lookup processes in a DATA step. First, a non-temporary SAS array is limited to 32767 items minus the number of variables already present in the Program Data Vector. As opposed to that, the size of a temporary array is limited only by the amount of memory available to the DATA step, for since its elements are not SAS variables, they occupy no space in the symbol table. Second, the latter means that we need not worry about dropping auxiliary variables. And third, the elements of a temporary array are automatically retained.
- However, the *techniques presented in the paper are not restricted to searching temporary arrays* only and can be applied to arrays of any type.
- Memory for temporary arrays is allocated in 8-byte multiples per item. If the items are numeric, they are naturally sized at 8 bytes per element. However, if the items are of the character type, each will occupy the number of bytes equal to the nearest multiple of 8 greater or equal to the declared length.

Now we are ready to implement the strategy outlined above:

STEP 1. Allocate a temporary array K to hold all the keys from the file SMALL and a *parallel* array S to hold corresponding satellites from SMALL.
STEP 2. Read SMALL and load its keys and satellites into the respective arrays.
STEP 3. Read a record from LARGE and invoke some routine searching K for KEY coming with the record.
STEP 4. If a match is found, use the index of the matching key to return the corresponding satellite from SMALL and write an output record.
STEP 5. Repeat steps 3 and 4 until LARGE is exhausted.

Or, expressing the algorithm in the dialect of the SAS DATA step:

```
DATA MATCH (KEEP=KEY S_SAT L_SAT);
   ARRAY K (&SIZE) _TEMPORARY_ ;
   ARRAY S (&SIZE) _TEMPORARY_ ;
   DO X=1 TO &SIZE;
      SET SMALL;
      K(X) = KEY;
      S(X) = S_SAT;
   END;
   DO UNTIL (END);
      SET LARGE END=END;
*******************************;
*** SEARCH ARRAY RIGHT HERE! ***;
*******************************;
      IF X > . THEN DO;
         S_SAT = S(X);
         OUTPUT;
      END;
   END;
   STOP;
RUN;
```

The only vital part absent from this code is *some* lookup routine that would:

- Search the array K for every value of KEY coming from the file LARGE.
- If search is successful, return the index of the matching key into a variable X.
- Otherwise set X to missing.

Thus, the index X returned by the lookup routine plays two roles outside the searching module: A) tells whether or not a matching key has been found, and B) helps pull the relevant satellite information from SMALL. Because the lookup routine is invoked every time a record from the large file is read, the efficiency of the entire matching process is completely determined by the efficiency of searching. The less comparisons and data moves it makes, the faster the program will run.

A WORD ABOUT TESTING

Since we are going to use the files LARGE and SMALL as a test bed to evaluate the relative efficiency of different lookup techniques, it is important to populate them with proper test data. Generally speaking, successful and unsuccessful searches occur at different speeds. So, to test

the performance of a certain searching method *on the average*, we have to concoct test data where A) search keys resulting in hits and misses are represented approximately equally and B) the keys would be distributed more or less uniformly throughout the range. Test files satisfying both conditions can be created, for instance, as follows:

```
%LET N_LARGE = < integer > ;
%LET N_SMALL = < integer > ;
DATA LARGE (KEEP=KEY L_SAT);
   DO L_SAT=1 TO &N_LARGE;
      KEY = CEIL(RANUNI(1)*1E6);
      OUTPUT;
   END;
RUN;
DATA SMALL (KEEP=KEY S_SAT);
   SET LARGE (KEEP=KEY OBS=&N_SMALL);
   S_SAT ++ 1;
   IF S_SAT > &N_SMALL*.5 THEN KEY = -KEY;
RUN;
```

All the keys for SMALL are thus chosen from LARGE and then one half of them are turned negative. Hence, exactly one half of the keys in SMALL will have their matches in LARGE and the rest are guaranteed not to match. All test keys are random 6-digit integers by design, and the satellites are given unique natural values simply because it helps identify them in the file MATCH. Varying the contents of the macro variables N_SMALL and N_LARGE, we can create test data of any sane size possessing the required sampling properties. All test results presented in the tutorial have been obtained on a P-II 233 MHz NT workstation with 196 MB of RAM running SAS version 6.12.

Now that we know exactly what needs to be achieved and how to test the results, we can concentrate on the main topic: *Array lookup techniques, their pros and cons.*

SEQUENTIAL SEARCH

"Start at one end and proceed towards the other end; if the right key is found, then stop." This brute force approach termed *sequential*, *serial*, or *linear* search, is the most natural inclination to do a lookup. It is also the easiest to administer programmatically:

```
*** PLAIN SEQUENTIAL SEARCH ***;

DO X=LBOUND(K) TO HBOUND(K);
   IF K(X) = KEY THEN LEAVE;
END;
IF X > HBOUND(K) THEN X = . ;
```

It does not take a Certified SAS programmer to realize that this algorithm is not a top performer. Assume that all inputs occur with equal probability, and we have N keys in the array. Then it will take (N+1)/2 comparisons on the average to complete a successful search; in the case of an unsuccessful search, all N items will have to be examined. Considering that more advanced search methods require no more than log(N)+1 comparisons, hit or miss, sequential search is slow, indeed. With very modest N_LARGE=100,000 and N_SMALL set to mere 1,000, the sequential search takes a *long 86 seconds* to match the files. In spite of that, sequential search has its pluses:

1. It is the simplest and, unlike certain more advanced schemes, it performs no computations other than setting an index up by unity. Hence, serial search does no worse than more complex techniques if the list of keys is quite short, and it can be of great value in final stages of hybrid high-performance algorithms.
2. It requires no special data organization, e.g. sorting, for the keys are traversed serially.
3. It is a natural rational choice in the situations when an unordered array has to be searched once.

For these reasons, the algorithm above is widely used, and so it would not hurt to have it as efficient as possible. Surprisingly, very few people realize that it is almost always *not the right way* to search sequentially! A closer look at the DO loop - the *inner loop* of the routine - can help tell why. Every time it iterates, *two* comparisons are made: One explicit and one implicit. First, K(X) = KEY, is made in the body of the loop; second, X > HBOUND(K), is made at the bottom.

A simple change can reduce the two comparisons to just one. Namely, let us allocate the array K with an additional dummy item at the upper bound. It can be done even without altering the value of SIZE:

```
ARRAY K (%EVAL(&SIZE+1)) _TEMPORARY_;
```

(The satellite array can remain intact.). If we populate the dummy slot with the search key, it will serve as a natural sentinel making the bottom loop-terminating comparison redundant. The explicit comparison can be then moved from the body of the loop to its bottom. Thus, we arrive at the following routine:

```
*** QUICK SEQUENTIAL SEARCH ***;

K (HBOUND(K)) = KEY;
X = LBOUND(K) - 1;
DO UNTIL (K(X)=KEY);
   X ++ 1;
END;
IF X = HBOUND(K) THEN X = . ;
```

Theory [1] predicts that this simple change results in about 30 percent faster average execution time whenever N > 8. This conjecture is corroborated by experimental data: Replacing the plain sequential search by the "quick" routine cuts the execution time down to *61 seconds*. Still, it is a far cry from the performance of more efficient methods discussed below.

Obviously, the sequential search is a poor choice for tackling our sample problem unless we only have a dozen keys in the small file. So, why use it in the first place? First, in terms of the sequential search, the sample problem is merely a vehicle to demonstrate the sufficiency, rather than efficiency, of the method, and help gauge its relative performance. Second, as noted above, for the tasks where an unordered array has to be searched once, the serial search is just the ticket.

The following analogy may help understand why: If we have 100 decks of playing cards and need to find a certain card in each, it is faster to go through every deck serially than to sort it first in order to search faster. By the same token, imagine a dataset containing variables V1-V100 having to be searched for the value of another variable *in every observation*. In this case, it will take much less time to simply search V1-V100 sequentially than to prepare the variables, in every record, for a speedier form of searching, for instance, by sorting them. And, of course, regardless of the reason the sequential search is employed, it is almost always better to use its "quick", rather than "plain", variant, for it runs considerably faster at the mere expense of one extra memory location.

BINARY SEARCH

In sequential search, no care has been taken to organize the lookup array in any particular way prior to searching - it is searched "as is". Whilst it is perfectly justified when a table, or a number of *different* tables, have to be searched *only once*, the absence of data organization results in a severe performance penalty if *the same* array needs to be looked up *repeatedly*. Continuing the "playing cards" analogy, if we have to find a different card 100 times in a row in the same deck, we can make the search process dramatically more efficient by taking time to appropriately arrange the cards in the deck in advance. Our sample problem, where searching is performed for each key coming from a potentially huge file, is just the case.

Out of great many ways array items can be arranged to facilitate efficient searching, the simplest is *sorting*. Having the elements ordered, we possess enough knowledge about them *beforehand* to make a judgement about the location of the search key in the lookup array *without comparing it to the majority of items*. Hence, we should expect that the order relation, if exploited properly, would result in a major performance gain.

Having a variety of sorting methods at our disposal, we should have little difficulty rearranging a lookup array into order. We can use PROC SORT or PROC SQL to sort and unduplicate the file SMALL by KEY, assign the number of output observations to SIZE and load the keys into the array K just as shown above. Or, alternatively, we can *first* load the keys into K, and *then* sort and unduplicate it completely in memory using one of high-performance sorting algorithms (see, for example, [6]). We will assume, therefore, that SMALL is already ordered by KEY and the duplicates, if any, have been removed.

At this point, the main question is how exactly can we take advantage of the order relation between the keys? Suppose we have chosen some array element K(M), sometimes called *a pivot*. That will divide the array into three parts: the items in the locations lower than K(M), K(M) itself, and the items in the locations higher than K(M). Accordingly, three mutually exclusive outcomes are possible:

1. KEY = K(M). The algorithm terminates successfully.
2. KEY < K(M). All elements with indices lower than M are eliminated from consideration.
3. KEY > K(M). All elements with indices higher than M are eliminated from consideration.

Thus, whilst the sequential search is limited to a *two-way decision* (equal or unequal), ordering enables us to continue search based on a *three-way decision*. Regardless of the way it branches, substantial advance has been made. However, it is important to choose M correctly. If we select M very close to one of array bounds, we may be lucky to eliminate the majority of keys at once, but on the other hand, we may end up having to search the majority of keys almost from scratch. In the worst case scenario, such a process may degenerate into serial search.

Therefore, if the only fact known about the keys is that they are sorted, the choice suggesting itself naturally is to start searching by comparing KEY to the *middle* item in the array. The result of the comparison will either locate the right key or tell which half to search next, and the same process can be used again. As a result, after at most about log(N) iterations, we will have either found the key or established that the array does not contain it. This procedure is most widely known as *binary search*, although other names such as "logarithmic search" are also used.

The basic idea of binary search seems to be transparent. However, many programmers (including yours truly) have coded it wrong the first time they attempted it! One of the most popular correct ways of implementing binary search is to maintain three indices: L pointing to the lower limit of the current search interval, M pointing to its middle, and H pointing to its upper limit. Then we can proceed as follows:

STEP 1. Initially, set L and H to the lower and upper bounds of the array, and set X to missing.
STEP 2. Compute M as the half-midpoint between L and H.
STEP 3. If KEY < K(M), set H to (M-1) leaving L intact.
STEP 4. Else if KEY > K(M), set L to (M+1) leaving H intact.
STEP 5. Else If KEY = K(M), search is successful. Set X to M and terminate the routine.
STEP 6. If the pointers L and H cross then stop, else repeat steps 2 through 5.

```
*** BINARY SEARCH ***;

L = LBOUND(K);
H = HBOUND(K);
X = .;
DO UNTIL (L > H);
   M = FLOOR((L + H)*.5);
   IF      KEY < K(M) THEN H = M - 1;
   ELSE IF KEY > K(M) THEN L = M + 1;
   ELSE DO;
      X = M;
      LEAVE;
   END;
END;
```

Note that every time the inner loop iterates, two comparisons are made: one inside the loop, and one (L > H) at its bottom, plus some extra time is spent calculating the midpoints. Theoretically, binary search will start outperforming quick sequential search at N satisfying N/2= log(N)-1 for a successful search and N = log(N)+1 if the search is unsuccessful. Accounting for the extra computation

and sorting, it roughly corresponds, on the average, to N = 12.

So, what practical benefits do we reap by replacing the 6 lines of quick serial search with 13 lines of binary search? Using the same input (N_LARGE=100,000 and N_SMALL=1,000), binary search completes the task in *2.4* seconds - almost 25 times the difference! To facilitate further performance comparisons, we can run the sample program with binary search on a larger scale, say, N_LARGE=1,000,000 and N_SMALL=100,000. It gets the job done in about *42 seconds*. It seems OK, yet in order to tell whether the time is actually good or bad, we will have to compare it with the times returned by other methods.

UNIFORM BINARY SEARCH

Why does binary search perform so much better than sequential? Largely, it is due to the fact that we have made use of what we knew about our data, i.e., that they were ordered, *prior to searching*. But we have not exploited the full potential of the knowledge! Remember, we decided to select a pivot in the middle of the search interval to ensure that no weird input would elicit worst-case behavior, but we could have achieved the same goal by selecting it totally at random. However, choosing a pivot in the middle has another merit: No matter how many times the same array is searched, the pivot locations in binary search are predetermined. We did not use this information, having made our routine compute the midpoints in each iteration. Instead, we can take advantage of the bisection by *precomputing* the midpoints before commencing on reading the large file and storing them in a small auxiliary array. Inside the inner loop, we can then simply use the precomputed midpoints via an index. As we already know, the inner loop of binary search never iterates more than log(N)+1 times, so allocating the auxiliary array with 50 items will suffice for a lookup table with 2**50 = 1.1E15 keys, more than anyone will ever be able to use. Thus, the auxiliary array can be safely hardcoded as D (50) with no risk of overflow.

The variety of binary search just outlined is known as *uniform binary search* due to fairly intricate theoretical reasons. Before the technique can be implemented properly, two things must be done. First, we will need to add a slot indexed as 0 to our lookup array and fill it permanently with a missing value, which will play the role of a sentinel. Otherwise, uniform binary search can index K out of its lower bound trying to refer to the zero item when N is even. With that in mind, the array K should now be allocated as

```
ARRAY K (0:&SIZE) _TEMPORARY_;
```

Second, the auxiliary array must be properly populated prior to reading LARGE. This is done by inserting the following excerpt into the main program, immediately prior to DO UNTIL (END) statement:

```
ARRAY D (50) _TEMPORARY_;
DIFF = DIM(K) - 1;
DO P=1 TO LOG2(&SIZE) + 2;
   DIFF  = DIFF * .5;
   D(P) = FLOOR(DIFF + .5);
END;
K(0) = .;
```

The last line places a sentinel value into the zero occurrence of the lookup array. Now we are ready to go ahead:

```
*** UNIFORM BINARY SEARCH ***;

X    = .;
P    = 1;
M    = D(1);
DIFF = D(1);
DO WHILE (DIFF > 0);
   P ++ 1;
   DIFF = D(P);
   IF      KEY < K(M) THEN M +- DIFF;
   ELSE IF KEY > K(M) THEN M ++ DIFF;
   ELSE DO;
      X = M;
      LEAVE;
   END;
END;
```

Superficially, it does not look too much different from ordinary binary search, yet it reduces the time required to complete the sample task to about *35 seconds* from 42 achieved by plain binary search, i.e., runs about 17 percent faster. The sole purpose of the variable DIFF is to replace two array references to D(P) (at the top and in the body of the loop) with just one. It saves time, for an array reference is a computation, too, and every single instruction taken out of the inner loop tends to improve performance. To understand better how the routine operates, let us print the items 1 through log(N)+2 of the array D for N=1000:

```
D[01] = 500    D[07] = 008
D[02] = 250    D[08] = 004
D[03] = 125    D[09] = 002
D[04] = 063    D[10] = 001
D[05] = 031    D[11] = 000
D[06] = 016
```

Each time the inner loop iterates and the index P into the array D(P) is incremented by a unity, a new level of binary division is reached, and a new midpoint is obtained from M by adding or subtracting the current value of D(P). In the case of a direct hit, the LEAVE statement terminates the loop, indicating a successful search. If the final level, D(11)=0, is reached, but a match has not been found (corresponding to the situation when pointers L and H have crossed in the ordinary binary search), the search has been unsuccessful, and the loop is terminated at the top.

The only advantage of the plain binary search is that it can be better encapsulated as a called module because of its simplicity. If the latter is not paramount, there is no real reason to opt for a slower routine, and uniform binary search should be definitely preferred.

INTERPOLATION SEARCH

Using the divide-and-conquer paradigm to construct binary search, we made the seemingly "natural" choice of parting the search interval in the middle, with two reasons in mind. First, it allowed eliminating worst-case behavior without resorting to computing pivots at random. Second, it lent itself to the implementation of uniform binary search.

However, the "naturality" of such a decision is quite questionable! Imagine yourself writing an elaborate binary search routine. When you reach for the SAS Language manual to further deepen your immense knowledge of arrays, you instantly forget about the clever algorithm that you are busy coding. Instead of opening the index in the middle, just as binary search would prescribe, you look for the word "ARRAY" *at the beginning*, for you know full well that it starts with "A". Having improved your understanding of arrays, you proceed with coding, and, unsure whether WHILE or UNTIL would suit your needs better, open the manual again. This time, you go straight *to the end* of the index, for you are too smart to blindly search for "W" in the middle! At last, you reach for the manual because you are using a SELECT block instead of IF-THEN-ELSE and wonder why the routine loops forever, even though you have instructed the program to LEAVE. Of course, now you open the index in the middle - not because you have finally decided to follow the binary search recipe, but simply because "L" is located near the center of the alphabet.

Apparently, the procedure of looking the words up in the index worked just fine! Is it possible to apply a similar reasoning in order to make binary search smarter? The answer is "yes", and it is not difficult, once a suitable way has been found telling the program how to mimic the heuristic process. Such an algorithm is called *interpolation search* because it hinges on making an educated guess where to look for KEY in its current search interval by predicting a calculated *interpolation point* .

Let us consider a search interval defined by keys K(L) < K(H). If the value of KEY is P percent of the difference K(H) - K(L) relative to K(L), we would start looking for KEY at a location M about P percent of (H - L) relative to L. In other words, the pivot index M can be derived from a simple proportion and expressed as

```
M = L+(KEY-K(L))*(H-L)/(K(H)-K(L)) .
```

The main idea of binary search remaining intact, this expression can be simply plugged into the binary search routine instead of M = FLOOR((L+H)*.5). However, it needs an important programming change: We need to avoid a zero division if L and H should become equal in some iteration of the inner loop. This can be achieved simply by adding 1 to the denominator, since in all practicality, it will not shift the interpolation point far away from its target - it is there only *approximately* in the first place. This leads to the following implementation of interpolation search:

```
*** INTERPOLATION SEARCH ***;

L = LBOUND(K);
H = HBOUND(K);
X = .;
DO UNTIL (L > H);
   M = CEIL(L+(H-L)*(KEY-K(L))/
        (K(H)-K(L)+1));
   IF M < L OR M > H THEN LEAVE;
   IF      KEY < K(M) THEN H = M - 1;
   ELSE IF KEY > K(M) THEN L = M + 1;
   ELSE DO;
      X = M;
      LEAVE;
   END;
END;
```

Notice that compared to binary search, another conditional LEAVE instruction has been added. Its purpose is to stop the inner loop if a computed value of M turns out to be *outside* of the current search interval, which means (in addition to the case of the pointers L and H having crossed) that the search has resulted in a miss. It also insures that if KEY is out of the overall lookup range, the algorithm will also terminate unsuccessfully after the very first iteration.

It can be proven [1] that if lookup keys are distributed evenly throughout the range, interpolation search needs at most log(log(N)) iterations to either find the right key or to establish that it is not present. In practice, it means that it will have to loop no more than 5 times to search *any array*. Indeed, even if N equals to a billion, log(log(N)) will not exceed 5. Searching 100,000 keys, as in our test data, interpolation search will iterate at most 3 times compared to 16 times for binary search.

Seemingly, compared to binary search, we must have achieved a giant leap in performance. In reality, interpolation search gets the job done in about *24 seconds* vs. 42 and 35 seconds for binary search and uniform binary search, respectively. 50 percent faster time is not a small improvement, but should we not expect more? There are two reasons why not. First, computing an interpolation point is much more expensive than a simple multiplication by 0.5. Second, the inner loop contains the additional (double) comparison. These factors are not essential in the case of *external* interpolation search where I/O drives execution times, but they become very significant when searching is conducted *internally* in memory.

Still, interpolation search runs much faster than even uniform binary search, so why not use it all the time? Because the superiority of interpolation process hinges on two assumptions: A) the keys are numeric and B) they are distributed uniformly.

The first assumption is not very critical. Searching a word in a dictionary, we never use more than three leftmost letters of the word before transitioning to a serial scanning. Likewise, we need only a few most significant bytes of a key to estimate a pivot location within a search interval. Hence, if the keys were alphanumeric, we could replace KEY, K(H) and K(L) in the interpolation formula with numbers having the same order relation by using INPUT function with the informat S370FPIBw., and it would more than suffice. Even better, memory permitting, we could create another parallel array and populate it with such numbers computed during

the load time, thus taking the computation out of the inner loop.

The second assumption is *very* critical. In the real world, keys are often distributed unevenly, and it can fool interpolation search quite badly. For instance, if RANNOR function were used to create the test data instead of RANUNI, interpolation search would still work but have to go through up to 30 (not three!) iterations per search with 100,000 keys in the array.

KEY-INDEXED SEARCH

To this end, we have been trying to extract the maximum of performance from *comparing a search key to the keys* in the array, striving to reduce the number of comparisons, computations and amount of data movement to a minimum. However, all comparison-based methods have a *principal limitation*: Unless the keys are distributed uniformly, none can search, success or failure, in fewer iterations than binary search, no matter how cleverly details are implemented.

The only way to overcome the limitation is to *do away with comparisons between keys altogether* and base searching on a radically different type of data organization known as *direct array addressing, key-indexing, or distribution.* Imagine that our keys represent one-digit numbers. Hence, we know in advance that no key can have values other than 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Assume that the file SMALL consists of the following 5 records:

KEY	**2**	**3**	**5**	**7**	**9**
S_SAT	**1**	**2**	**3**	**4**	**5**

and we wish to search for KEY=1 and KEY=7. Let us allocate an array having *1 slot for each possible key value*:

0	1	2	3	4	5	6	7	8	9
[.]	[.]	[.]	[.]	[.]	[.]	[.]	[.]	[.]	[.]

Now, let us read SMALL and *distribute* each S_SAT into the array location *whose index equals to the value of the corresponding KEY*:

0	1	2	3	4	5	6	7	8	9
[.]	[.]	[**1**]	[**2**]	[.]	[**3**]	[.]	[**4**]	[.]	[**5**]

We have just prepared a *key-indexed table* comprising two types of entries: *empty and occupied*. From that point on, searching is trivial. To look for a given key, we simply examine the location in the table whose index is equal to the key. If the location is empty, the key is not present in the table. If it is occupied, the key has been found, and its contents will return the corresponding satellite into S_SAT. For example, KEY=1 is not found because the address 1 of the table is empty, whereas KEY=7 is found with K(KEY)=4 returning the satellite.

Notice that there is no need to sort the file SMALL. If it happens to contain duplicate keys, they will be removed automatically by the process of loading a key-indexed table, but only the last instance of S_SAT corresponding to a repeating key will be written out. If SMALL has no satellites or we are not interested in obtaining them, the entries of the key-indexed table could be marked as occupied using any non-missing value, for instance, a unity. In this case, duplicate keys do not present any problem at all and are removed on the fly as the table is loaded.

Remember, the keys in our test datasets are 6-digit numbers (made such on purpose to facilitate key-indexed search making comparisons with other methods possible). Therefore, the universe of their possible values is restricted by integer numbers from -999999 to +999999, and we should allocate a key-indexed array of the appropriate size. Mimicking the process described above, we arrive at the following implementation of key-indexed search (the entire matching program, not just the searching routine, is shown):

```
DATA MATCH;
   ARRAY K (-1000000:1000000)
   _TEMPORARY_;
   DO UNTIL (EOF);
      SET SMALL END=EOF;
      K(KEY) = S_SAT;
   END;
   EOF = 0;
   DO UNTIL (EOF);
      SET LARGE END=EOF;
      S_SAT = K(KEY);
      IF S_SAT > . THEN OUTPUT;
   END;
RUN;
```

We can plainly see from the nature of the algorithm that *no other existing lookup method can run faster than key-indexing*: It completes any search, successful or not, without comparing any keys, using a single array reference. Because of the latter, it possesses an amazing property setting it apart from all other methods except hashing: Its speed is independent of the number of keys in the lookup table. Regardless of whether SMALL has 10 or 1,000,000 keys, one act of key-indexed search takes precisely the same time.

These considerations are confirmed experimentally. Running against our test files, key-indexed search completes the task in a brief *4.2* seconds. Not only is it 7 times faster than interpolation search, the fastest technique we have tested to this end, but also 3 times faster than MERGE would match the files if both were already sorted.

So, what is the catch? If key-indexed search is so great, why not forget everything else and use it all the time? The fly on the ointment of this beautiful idea is that *key-indexed search is only applicable when lookup keys are integers falling into a limited range or if they can be rapidly converted to such integers*. Our test keys are signed six-digit integers, so there are only 1,999,999 distinct values a given key can assume, and array space can be allocated for all of them at the expense of 16 MB of memory. Given enough memory, one could probably get away with seven-digit keys. But if the keys represented 9-digit social security numbers, an array with 1 billion elements would be needed which is almost impossible, whilst 16-digit credit card numbers would make key-indexing plainly unfeasible. The situation is even worse in the case of alphanumeric keys, whose 256-radix makes decimals representing them grow beyond physically possible array sizes at only 4 bytes of length.

On the other hand, in many real-world situations when key data do fall into a limited integer range, key-indexing, given its blazing speed and simplicity, is beyond competition. Here are some examples:

1. SAS date values. A limited range of integers covers any practical date range. For instance, a key-indexed table sized as [-138061:380217] will account for all SAS dates up to the 4th millenium.
2. SAS time values. An array sized as [0:86400] will suffice to key-index any SAS time value.
3. ICD9/CPT4 codes are short, mainly digital, strings. If some character is a letter, it can be converted into a number in 1-26 range and merged with the remaining digital bytes. (The idea belongs to Don Stanley).
4. 1-byte alphanumeric keys map onto the range [0:255] as RANK(KEY). INPUT(KEY,S370FPIB2.) will map all 2-byte keys map onto the range [0:65535].
5. Any fractional keys that result in limited range integers when multiplied by a suitable constant.

Most importantly, however, the idea of direct addressing on which key-indexing is based can be generalized to embrace keys of *any type and range* resulting in a group of fascinating search methods called *hashing*. Compared to key-indexed search, hashing yields dramatic savings in memory retaining virtually all performance advantages of key-indexing. It will be discussed in detail in Part II of the tutorial given in the Advanced Tutorials section.

PERFORMANCE COMPARISONS

Now that we have an idea how our home-cooked search methods compare to each other, it is interesting to compare them to the fast food supplied by SAS in the form of user-defined formats and SQL - that was the purpose of having the sample problem in the first place. For testing, format search and SQL join have been coded as follows:

```
DATA CNTLIN (DROP=KEY S_SAT);
   RETAIN FMTNAME 'SEARCH' TYPE 'N';
   DO UNTIL (END);
      SET SMALL END=END;
      START = KEY;
      LABEL = PUT(S_SAT,BEST.);
      OUTPUT;
   END;
   HLO = 'O'; LABEL = ' '; OUTPUT;
RUN;
PROC FORMAT CNTLIN=CNTLIN; RUN;
DATA MATCH;
   SET LARGE;
   S_SAT = PUT(KEY,SEARCH.);
   IF S_SAT > ' ' THEN OUTPUT;
RUN;
OPTION MSGLEVEL=I;
PROC SQL _METHOD BUFFERSIZE=2000000;
   CREATE TABLE MATCH AS
   SELECT S.KEY, S_SAT, L_SAT
   FROM SMALL S, LARGE L
   WHERE S.KEY = L.KEY;
QUIT;
```

For each method, run times were measured in two stages: A) loading a lookup table into memory and B) matching itself. In the case of formatting, load time consists of creating CNTLIN and format compilation. SQL needs no preparation time at all. BUFFERSIZE was chosen rather huge to make the optimizer use SQXJHSH join method (the SAS log showed it did). For binary, uniform, interpolation, and key-indexed search, stage (A) amounts to populating the arrays. Tests were run for N_LARGE fixed at 1,000,000 records, but N_SMALL was varied. Let us examine the following table where the measured times are given in seconds, and memory usage - in megabytes:

N_SMALL	METHOD	LOAD	SEARCH	ALL	MEM
100,000	**BINARY**	**0.3**	**41.0**	**41.3**	**7.3**
	UNIFORM	**0.3**	**34.9**	**35.2**	**7.3**
	INTERPOL	**0.3**	**22.9**	**23.2**	**7.3**
	KEY-INX	**0.2**	**4.0**	**4.2**	**20.2**
	SQXJHSH	N/A	14.6	14.6	18.4
	FORMAT	15.3	22.9	38.2	17.1
200,000	**BINARY**	**0.5**	**43.8**	**44.3**	**8.6**
	UNIFORM	**0.5**	**37.2**	**37.7**	**8.6**
	INTERPOL	**0.5**	**23.5**	**24.0**	**8.6**
	KEY-INX	**0.5**	**4.3**	**4.8**	**20.2**
	SQXJHSH	N/A	18.8	18.8	19.2
	FORMAT	28.5	24.6	53.1	20.5
300,000	**BINARY**	**0.8**	**45.5**	**46.3**	**10.0**
	UNIFORM	**0.8**	**39.5**	**40.3**	**10.0**
	INTERPOL	**0.8**	**23.9**	**24.7**	**10.0**
	KEY-INX	**0.7**	**4.7**	**5.4**	**20.2**
	SQXJHSH	N/A	22.7	22.7	22.9
	FORMAT	45.0	26.1	71.1	27.1
400,000	**BINARY**	**1.1**	**47.1**	**48.2**	**11.2**
	UNIFORM	**1.1**	**40.0**	**41.1**	**11.2**
	INTERPOL	**1.1**	**24.3**	**25.4**	**11.2**
	KEY-INX	**0.9**	**5.0**	**5.9**	**20.2**
	SQXJHSH	N/A	27.3	27.3	22.9
	FORMAT	58.5	27.3	85.8	34.2
500,000	**BINARY**	**1.2**	**50.8**	**52.0**	**12.3**
	UNIFORM	**1.2**	**41.3**	**42.5**	**12.3**
	INTERPOL	**1.2**	**25.7**	**26.9**	**12.3**
	KEY-INX	**0.9**	**5.2**	**6.1**	**20.2**
	SQXJHSH	N/A	27.9	27.9	28.1
	FORMAT	65.0	28.8	93.8	40.4

As we see, hand-coded array lookup methods perform on par or better than ready-to-go routines, such as formats and SQL. The latter retain some edge as long as the volume of data is low, but all comparison-based techniques searching ordered arrays become relatively more efficient as the number of lookup keys grow. Starting at N=200,000, all of them outclass formatting in both overall execution time and memory usage. Although SQL in its hashing mode runs handsomely (albeit at the expense of heavy memory consumption) it gets outperformed by interpolation search and especially key-indexing if the nature of keys makes these two approaches applicable.

It should be noted that the sample problem, as well as the comparisons with formatting and SQL, are being used here only to gauge performance and make sure that array lookup techniques perform acceptably compared to other available methods. File match problems can be solved using array lookup, but other methods will work, too. However, there are applications where array search is the only way to go. In all such cases, various array lookup techniques should be carefully considered to account for such factors as the task at hand, programming simplicity, data volume, the nature of the keys, and encapsulation.

A WORD ABOUT ENCAPSULATION

Any general-purpose high-performance routine is viewed as a potentially reusable component, and therefore it is better to modularize it. This can be done in two ways: Either as a LINK subroutine or a macro.

If a module is going to be called a few times during the course of a DATA step, LINK will do just fine. However, if the module has to be called repeatedly, as it does in the file match problem, invoking it as a LINK routine induces certain overhead that may result in a mild performance penalty. Besides, the reusability of a LINK module is restricted by the limits of the current DATA step.

Macroizing would be a better solution. First, the macro will place the code it assembles in-line. Second, it provides for unlimited reusability. Third, a parameterized macro is much more flexible and better encapsulated. As an example, let us consider how plain binary search could be macroized:

```
%MACRO BSEARCH (ARRAY=,KEY=,RETURN=X);
   %LOCAL L M H;
   %LET L=L%SUBSTR(%SYSFUNC(RANUNI(0)),3,7);
   %LET M=M%SUBSTR(%SYSFUNC(RANUNI(0)),3,7);
   %LET H=H%SUBSTR(%SYSFUNC(RANUNI(0)),3,7);
   DROP &L &M &H &RETURN;
   &L=LBOUND(&ARRAY);
   &H=HBOUND(&ARRAY);
   &RETURN = .;
   DO UNTIL (&H < &L);
      &M = FLOOR((&L+&H)*.5);
      IF      &KEY < &ARRAY(&M) THEN &H=&M-1;
      ELSE IF &KEY > &ARRAY(&M) THEN &L=&M+1;
      ELSE DO;
         &RETURN=&M;
         LEAVE;
      END;
   END;
%MEND BSEARCH;
```

This macro could be then placed in an autocall library and invoked from an appropriate location in the DATA step simply as

```
%BSEARCH (ARRAY=AKEY,KEY=KEY,RETURN=X) .
```

Note that the names for the "internal" variables L, M, and H were selected at random to ensure that they would not clash with other variables the DATA step may incorporate. This is a lazy approach; however, in SAS versions greater than 6 where a random sequence of digits much longer than 7 can be built in a variable name, it will be bulletproof. A fault-tolerant eager approach based on random macro aliases, can be found in [6].

CONCLUSION

Based on theoretical considerations and experimental results, it is possible to decide which array lookup technique should be preferred under different circumstances.

If an array or a number of different arrays must to be searched just once, for instance, if you need to do a lookup in every observation of a dataset, quick sequential search should be used. It is as simple as plain serial search, but saves about 30 percent of execution at the expense of only one additional array item.

If the same array has to be searched repeatedly, the choice depends on the number of factors.

When the keys are integers falling into a limited range or if they can be converted to such integers inexpensively, key-indexed search is beyond competition: Its speed cannot be matched by any other known searching technique.

Otherwise, we will have to use some method based on searching an ordered table, so the array must be sorted, if necessary, along with satellite fields. If the keys are distributed uniformly within their range, interpolation search is the best choice. Compared to binary search, it requires one extra instruction, but may be twice as fast.

If the keys lack the properties making key-indexing or interpolation applicable, it leaves us with ordinary or uniform binary search. Uniform binary search is about 15 percent faster and therefore should be preferred. However, if simplicity, rather than speed, is paramount, ordinary binary search will have no problem getting the job done, either.

Finally, it might be important to encapsulate a searching routine. Every algorithm discussed above can be easily modularized, but plain binary search and interpolation search, consisting of several instructions concentrated next to each other, would probably be preferable.

REFERENCES

1. D. E.Knuth, *The Art of Computer Programming*, **3**.
2. D. E. Knuth, *Computing Surveys* **6** (1974), 266.
3. D. H. Lehmer. *Proc. Symp. Appl. Math* **10** (1960), 180.
4. W. W. Peterson. *IBM J. Res. & Devel.* **1** (1957), 131.
5. Paul M. Dorfman, Proceedings of SUGI 24 (1999), 1362.
6. Paul M. Dorfman, *Proceedings of SESUG'98* (1998), 97.

ACKNOWLEDGEMENTS

The author would like to acknowledge the contributions of the following individuals:

Doris H. Bogar
E. Winston Churchill
Michael V. Dorfman
Victor P. Dorfman
Yuri Katsnelson
F. Joseph Kelley
Vladimir A. Kirillov
Eugenia P. Kravchenko
Michael A. Raithel
Vera M. Voloshin
Matthew A. Wilson
Ian Whitlock

AUTHOR CONTACT INFORMATION

Paul M. Dorfman
10023 Belle Rive Blvd. 817
Jacksonville, FL 32256
(904) 564-1931
sashole@earthlink.net