SESUG 2022 Paper 191

Array Searching Algorithms and Techniques

Paul M. Dorfman, Independent SAS® Consultant

ABSTRACT

As SAS data structures, arrays have a variety of uses, all based on quick direct accessibility of each array element by its own unique index. However, the latter is most important for using arrays for key lookup and data retrieval: Since array indices can be programmatically manipulated in every conceivable way, any known search algorithm can be implemented using SAS arrays. This paper is a classified compendium of array searching algorithms and their practical implementations in the SAS language.

INTRODUCTION

Table lookup being one of the fundamental processing operations, it comes as no surprise that SAS offers a rich collection of 4GL searching tools. The DATA step match-merge, SQL joins, formats, string searching functions, direct access by the observation number, SAS indexes, and particularly the hash object - all serve, whether explicitly or implicitly, the purpose of searching relevant data. However, no set of canned tools can always cover all possible searching scenarios. Thankfully, SAS also offers various forms of an array - a data structure ideally suited to implementing of just about any searching algorithm using the SAS language. Of course, array-based searches need to be programmed, tested, and tuned. But first, SAS programming is fun! Furthermore, the array approach is more flexible than using canned tools and can often results in faster programs using fewer resources.

In this paper, a number of common array-searching algorithms are presented along with their DATA step turnkey implementations. First, we will look at lookup schemes based on comparing a search key to keys stored as array elements. Second, we will discuss direct-addressing search schemes based on digital properties of the array keys. Throughout the paper, the terms *file*, *data set*, *table* are used interchangeably, as well as the terms *observation*, *record*, *row*.

PROBLEM SETTING

We have a *key array* K populated with *key-values* and another, parallel, *data array* D sized and bound identically to K and populated with *data-values*. Also, we have a *search-key* variable SK of the same data type as the key array K. Our task is as follows:

1. Discover if any key-value K[x]=SK. If so, set a flag variable KF=1; else leave KF at null.
2. Optionally, in case if KF=1, retrieve the data-value D[x] and set a variable DF=D[x].
3. Optionally, display the values of KF and DF.

It can be illustrated by the following DATA step arrangement:

```
data _null_ ;
  array K [19] (76 59 19 32 36 90 84 56 20 48 23 85 71 12 17 66 82 88 33) ;
  array D [19] ( 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19) ;
  SK = <value> ;
  link search ;
  put (SK KF DF) (=) ;
  stop ;
  search:
  call missing (KF, DF) ;
  < array-searching routine >
```

```
run ;
```

It should be noted that:

- Both arrays can be character or numeric.
- If the data type of the key array K matters, it will be duly noted, while the data type of the data array makes no difference in any case.
- The arrays can incorporate PDV variables (as above) of be temporary.
- To simplify the discussion, we assume (as above) that the key-values are unique.

CANNED CASES

SAS offers two canned tools to search an array: The IN operator and the WHICHN/WHICHC functions. Internally, both methods are based on the sequential search (see below) because they assume that the data organization of the key array can be arbitrary, hence they cannot make use of a specific data organization (such as the sorted order, for example). Let us look at them one at a time.

USING THE IN OPERATOR

The most trivial case occurs if it is only required to discover whether the key array contains the search-key SK. In this case, we need neither the data array nor the variable DF, and the entire <*array-searching code*> is reduced to a single expression (shown in boldface):

```
data _null_ ;
  array K [19] (76 59 19 32 36 90 84 56 20 48 23 85 71 12 17 66 82 88 33) ;
  do SK = 56, 99 ;
    link search ;
    put KF= ;
  end ;
  stop ;
  search:
  call missing (KF) ;
  if SK in K then KF = 1 ;
run ;
------------------
KF=1
KF=.
```

USING THE WHICHN/WHICHC FUNCTIONS

These functions are designed to return the first array index X at *which* (hence the name of the functions) K[x]=SK. Thus, they can be used not only to discover whether SK is in the array but also retrieve the corresponding data-value D[x]:

```
data _null_ ;
  array K [19] (12 17 19 20 23 32 33 36 48 56 59 66 71 76 82 84 85 88 90) ;
  array D [19] (14 15  3  9 11  4 19  5 10  8  2 16 13  1 17  7 12 18  6) ;
  do SK = 56, 99 ;
    link search ;
    put (SK KF DF) (=z2.) ;
  end ;
  stop ;
  search:
  call missing (KF, DF) ;
  X = whichN (SK, of K[*]) ;
```

```
  if X then do ;
    KF = 1 ;
    DF = D[x] ;
  end ;
run ;
-----------------
SK=56 KF=01 DF=08
SK=99 KF=.  DF=.
```

One disadvantage of this method is that it is data type specific, so if the array K were character, the WHICHC function should be hard coded instead; and hard coding makes the program less dynamic. Another shortcoming is that the functions assume that the lower bound of the array being searched is 1, so for an array bound differently the response X will not return the correct array index if SK=K[X]. To go around this limitation, a degree of pretty cumbersome computational gymnastics would be needed.

PART 1: KEY-COMPARISON ALGORITHMS

Key-comparison algorithms are based on comparing the search-key SK with some or all key-values K[x] in the array. An alert reader may ask: "But how can exist an algorithm that does not make any such key comparisons at least once?" The answer is that such algorithms do exist; moreover, there is a family of search algorithms which use some key comparisons, but it is not their primary searching mechanism. We will look at such direct-addressing algorithms in the second part of the paper.

SEQUENTIAL SEARCH

"*Start at one end and proceed towards the other; if the right key is found, then stop.*" This brute force approach termed the *sequential* (and also *serial* or *linear*) search is the most natural inclination to do an array lookup. It is also the easiest to program. Note that below, the LINK routine named *search* is called twice: for a search-key present in the array and for a search-key absent from it.

```
data _null_ ;
  array K [19] (76 59 19 32 36 90 84 56 20 48 23 85 71 12 17 66 82 88 33) ;
  array D [19] ( 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19) ;
  do SK = 56, 99 ;
    link search ;
    put (SK KF DF) (=z2.) ;
  end ;
  stop ;
  search:
  call missing (KF, DF) ;
  do x = lbound (K) to hbound (K) until (KF) ;
    if K[x] = SK then KF = 1 ;
  end ;
  if KF then DF = D[x] ;
run ;
-----------------
SK=56 KF=01 DF=08
SK=99 KF=.  DF=.
```

It should be obvious from the nature of the algorithm that it won't win any races since the array items are examined one at a time. If the search-key is not found, every key-value K[x] in the array is compared to SK. If it is found, then (assuming the key-values are random), on the average, half of the key-values are compared to SK. Furthermore, if the array size grows N times, the number of key comparisons in both cases also grows N times.

Correspondingly, the execution time also grows N times. Formally this fact is usually expressed by saying that the algorithm runs in *O*(N) time (the so-called "big *O* notation").

However, it does not mean at all that the sequential search is practically useless; quite the contrary! First, it is actually the fastest key-comparison algorithm when the number of key-values is approximately fewer than 7. This is because the reduction in key comparisons by more complex algorithms (such as the binary search below) fails to outweigh their additional computational overhead.

Second, it is practically the only scheme that can be used for non-static arrays - that is, when before the search routine is executed, any key-value can change. For example, if an array is to be searched for every observation read from a SAS data set using the binary search, the array would have to be resorted every time before it could be searched. That would ruin any extra efficiency of the binary search compared to the sequential since sorting is much more computationally involved than merely scanning the entire array. Incidentally, this is the reason why behind-the-scenes the "if SK in K" trivial search described above is based on the sequential search algorithm.

Third, performance of the sequential search can be inexpensively improved by using the "sentinel technique" to eliminate a number of hidden comparisons.

QUICK SEQUENTIAL SEARCH

The sequential search implementation as presented above makes 2 comparisons per every key-value K[x]: (1) it explicitly checks if SK=K[x] and (2) it implicitly checks the TO endpoint of the DO loop by testing x>hbound(K). Fortunately, the latter can be eliminated by using the so-called "*sentinel technique*".

Namely, we can add an extra item to the key array – for example, with index 0 - and place a *sentinel equal to the search-key SK* into this extra slot, i.e., set K[0] = SK. Then we can scan the array backwards and merely check if SK=K[x]. If SK is found somewhere at x>0, the loop will stop iterating there; if not, it will stop at K[0]=SK. In the SAS language:

```
data _null_ ;
  array K [0:19] (. 76 59 19 32 36 90 84 56 20 48 23 85 71 12 17 66 82 88 33) ;
  array D [  19] (   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19) ;
  do SK = 56, 99 ;
    link search ;
    put (SK KF DF) (=z2.) ;
  end ;
  stop ;
  search:
  call missing (KF, DF) ;
  K[0] = SK ;
  do x = hbound (K) by -1 until (K[x] = SK) ;
  end ;
  if x > 0 then KF = 1 ;
  if KF then DF = D[x] ;
run ;
-----------------
SK=56 KF=01 DF=08
SK=99 KF=.  DF=.
```

Though this simple inexpensive trick cannot change the *O*(N) nature of the sequential search, it can make it run up to 30 per cent faster by eliminating the excessive comparisons.

ORDERED SEQUENTIAL SEARCH

Let us now assume that the key array K is presorted by K[x] and the data array D is permuted accordingly:

```
 array K [0:19] (. 12 17 19 20 23 32 33 36 48 56 59 66 71 76 82 84 85 88 90) ;
 array D [  19] (  14 15  3  9 11  4 19  5 10  8  2 16 13  1 17  7 12 18  6) ;
```

(Sorting parallel arrays is beyond the scope of this paper. However, it can be easily done using the %qsort macro routine described in [Dorfman, 2001a] or by using the hash object.)

It should come as no surprise that making use of the sorted order can make the sequential search run faster on the average. Indeed, if we want, for example to search for SK=99, we no longer have to scan through the entire array to discover that it is absent: We will know it already on the first hit K[19]=90 because 90>99, so it is impossible for any key-value with X<19 to be 99.

The quick sequential routine can be easily modified to take advantage of the sorted order (the needed additions are shown in red):

```
data _null_ ;
  array K [0:19] (. 12 17 19 20 23 32 33 36 48 56 59 66 71 76 82 84 85 88 90) ;
  array D [  19] (  14 15  3  9 11  4 19  5 10  8  2 16 13  1 17  7 12 18  6) ;
  do SK = 56, 99 ;
    link search ;
    put (SK KF DF) (=) ;
  end ;
  stop ;
  search:
  call missing (KF, DF) ;
  K[0] = SK ;
  do x = hbound (K) by -1 until (K[x] < = SK) ;
  end ;
  if x > 0 and K[x] = SK then KF = 1 ;
  if KF then DF = D[x] ;
run ;
---------------
SK=56 KF=1 DF=8
SK=99 KF=. DF=.
```

BINARY SEARCH

While accounting for the sorted order makes the sequential search run a tad faster on the average, it still does not change its fundamental *O*(N) run nature. However, the fact that the key array is sorted can be used in a radically different manner to cut the number of comparisons of K[x] to SK dramatically as the number of array items grow.

To do so, it is sufficient to mimic programmatically more or less what we do when looking up an entry in a dictionary. Since its entries are sorted, we do not start at the first or last entry but instead somewhere in the middle. If the entry being searched is there, we have found it. Otherwise, we look at the only part of the book where the entry of interest can only exist according to the sorted order and then repeat the process until the entry is either found or there is nothing more to divide.

It can be interpreted more formally as follows: Suppose we have chosen some array element K(M), sometimes called a *pivot*. It divides the array into three parts: the key-values lesser (lexicographically) than K(M), K(M) itself, and the key-values greater than K(M).

Accordingly, three mutually exclusive outcomes are possible:

1. SK = K(M). The algorithm terminates successfully.

2. SK < K(M). All items with the indices > M are eliminated from consideration.

3. SK > K(M). All items with the indices < 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 most keys at once; but on the other hand, we may end up having to search most keys almost from scratch. In the worst-case scenario, such a process may degenerate into the sequential search.

Hence, if the only fact known about the key-values is that they are sorted, the choice suggesting itself naturally is to start searching by comparing SK to the *middle* item in the array. 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 log2(dim(K)) iterations, we will have either found SK or established that it is absent. This procedure is most widely known as the binary search, although other names such as "logarithmic search" also exist.

Though the basic idea of the binary search is fairly transparent, many a programmer fail to implement it correctly at the first attempt. The most conventional way of coding the algorithm is to maintain three pointers: L to point to the bottom of the current search interval, M to point to its middle, and H to point to its top. Then we can proceed as follows:

STEP 1. Initially, set L and H to the lower and upper bounds of the array.

STEP 2. Compute M as the half-midpoint between L and H.

STEP 3. If SK < K[M], set H = M-1.

STEP 4. Else if SK > K[M], set L = M+1.

STEP 5. Else If SK = K(M), SK is found. Set KF to 1.

STEP 6. If L > H (the pointers crossed) or KF = 1, terminate. Else go to step 2.

STEP 7. If KF = 1, retrieve the data-value by setting DF = K[M].

In the SAS language:

```
data _null_ ;
  array K [19] (12 17 19 20 23 32 33 36 48 56 59 66 71 76 82 84 85 88 90) ;
  array D [19] (14 15  3  9 11  4 19  5 10  8  2 16 13  1 17  7 12 18  6) ;
  do SK = 56, 99 ;
    link search ;
    put (SK KF DF) (=z2.) ;
  end ;
  stop ;
  search:
  call missing (KF, DF) ;
  L = lbound (K) ;
  H = hbound (K) ;
  do until (L > H or KF) ;
    M = floor ((l + h) / 2) ;
    if      SK < K[M] then H = M - 1 ;
```

```
    else if SK > K[M] then L = M + 1 ;
    else KF = 1 ;
  end ;
  if KF then DF = K[M] ;
run ;
```

```
-----------------
SK=56 KF=01 DF=8
SK=99 KF=.  DF=.
```

Counting the number of the DO loop iterations above (same as the number of key comparisons; let us denote it Q) shows that Q=4 for SK=56 (found) and Q=5 for SK=99 (not found). That is, both lie between log2(DK) and 1+log2(DK) where DK=dim(K). In other words, with 19 key-values to search, the binary search makes 4 to 5 key comparisons, hit or miss, whereas the sequential search makes 10 and 19, respectively. However, it is far more important how the binary search scales. For example, for an array size DK=2^10=1024, the number of key comparisons Q will never exceed just 11=1+log2(DK), and for DK=2^20=1048576, Q will never exceed 21. Obviously, the sequential search cannot match it by orders of magnitude.

To put it at a different angle, if the array dimension grows N times, the number of key comparisons for the binary search grows as log2(N). Thus, in the "big *O*" notation, the binary search runs in *O*(log2(N)) time.

We can conclude that if our search array is static, i.e., if it needs to be sorted only once or maybe resorted a few times throughout the duration of the DATA step, the binary search is the searching method of choice in the realm of the methods based on comparisons between keys. In fact, no key-comparison search can do better than the binary search if all we know about the key-values is that they are sorted.

BINARY SEARCH VARIANTS

There exist two variants of binary search: One, applicable in any situation and the other, applicable when the key-values are distributed uniformly. The two will be conceptually described in the following subsections and offered to the readers to implement in SAS code as exercises (though references to existing implementations will be given as well).

Uniform Binary Search

Performance-wise, the binary search can be somewhat improved by noting that every time it is performed, the midpoints for every partition must be computed while it is running. However, since for an array K with fixed bounds they are the same for every search, they can be computed beforehand and stored in a small separate array (named Z, say). Then the values from Z can be used instead of recomputing them on the fly. As to the size of Z, it can be safely hardcoded as Z[40] with no risk of overflow because this is enough to store the midpoints of an array with DK=2**40=1E12 items – in other words, any conceivable SAS array. The algorithm with this kind of performance improvement is for some reason called the *uniform binary search*.

Note that in the DATA step implementation below, the array K has an extra *sentinel* item K[0] permanently set to null. Without it, the program can index K out of its lower bound trying to refer to the item K[0] when the number of items dim(K) is even. The code filling the array Z only once is segregated into the LINK routine PREPZ called by the main LINK routine SEARCH.

```
data _null_ ;
  array K [0:19] (. 12 17 19 20 23 32 33 36 48 56 59 66 71 76 82 84 85 88 90) ;
  array D [  19] (  14 15  3  9 11  4 19  5 10  8  2 16 13  1 17  7 12 18  6) ;
  do SK = 71, 99 ;
```

```
    link search ;
    put (SK KF DF) (=z2.) ;
  end ;
  stop ;
  prepz:
    array Z [40] _temporary_ ;
    diff = dim (K) - 1 ;
    do p = 1 to log2 (diff) + 2 ;
      diff = divide (diff, 2) ;
      z[p] = floor (diff + 0.5) ;
    end ;
  return ;
  search:
    q ++ 1 ;
    if q = 1 then link prepz ;
    call missing (KF, DF) ;
    p    = 1 ;
    M    = z[1] ;
    diff = M ;
    do p = 2 by 1 while (diff > 0) ;
      diff = z[p] ;
      if SK < K[M] then M +- diff ;
      else if SK > K[M] then M ++ diff ;
      else KF = 1 ;
    end ;
    if KF then DF = D[M] ;
  return ;
run ;
-----------------
SK=71 KF=01 DF=13
SK=99 KF=.  DF=.
```

Of course, the uniform binary search does not change the *O*(log2(N)) run time of the binary search – it just eliminates repeated on-the-fly calculations. In case of small lookup arrays the difference is hardly worth the trouble of significantly more complex code. However, the uniform scheme can noticeably improve search performance for large lookup arrays. For example, for dim(K)=2**20=1048576, the binary search loop iterates about 21 times if the search-key is not in the array (worst-case scenario). Accordingly, it computes the expression floor((l+h)/2) approximately 16 times per search, on the average. If the search is used, for instance, to join a file with 100 million records using the sorted array as a lookup table, the uniform binary search will compute floor((l+h)/2) only 21 times, while the straight binary search would have to do it 2.1 billion times.

Interpolation Search

While 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 eliminates the worst-case behavior (without computing random pivots). Second, it lends itself to the implementation of uniform binary search.

However, the naturality of such a decision becomes questionable as soon as we once again recall how we look for an entry in a dictionary. Indeed, if our entry begins with A or Z, we will never open the book in the middle but rather closer to the beginning and the end, respectively. In fact, when deciding where to open it, we subconsciously *estimate* how far to open the book from its beginning based on (a) the first letter of the entry being looked up and (b) the loose assumption that the dictionary entries are distributed within their first letters more or less uniformly.

The same principle can be applied to the binary search to speed it up by parting the current interval closer to the likely location if the search-key based on its numerical value relative to the minimum and maximum key-values in the array. The algorithm based on this principle is called the *interpolation search* because it hinges on making an educated guess where to look for SK in the current search interval by predicting a calculated *interpolation point*.

Let us consider a search interval defined by the key-values K[L]<K[H]. If the value of SK is P percent of the difference K[H]-K[L] relative to K[L], we could start looking for SK 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 expressed as

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

The main idea of the binary search remaining intact, this expression can be simply plugged into the binary search routine instead of M = FLOOR((L+H)/2). However, we need a small but important change to avoid a zero division if L and H should become equal during some iteration of the search loop. This can be achieved merely by adding 1 to the denominator, for 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 in our sample array setting:

```
data _null_ ;
  array K [19] (12 17 19 20 23 32 33 36 48 56 59 66 71 76 82 84 85 88 90) ;
  array D [19] (14 15  3  9 11  4 19  5 10  8  2 16 13  1 17  7 12 18  6) ;
  do SK = 0, 56, 99 ;
    link search ;
    put (SK KF DF) (=z2.) ;
  end ;
  stop ;
  search:
  call missing (KF, DF) ;
  L = lbound (K) ;
  H = hbound (K) ;
  do until (L > H or KF) ;
    M = floor (L + (SK - K(L)) * (H - L) / ((K(H)-K(L)) + 1)) ;
    if not (L <= M <= H) then leave ;
    if      SK < K[M] then H = M - 1 ;
    else if SK > K[M] then L = M + 1 ;
    else KF = 1 ;
  end ;
  if KF then DF = K[M] ;
run ;
-----------------
SK=00 KF=. DF=.
SK=56 KF=01 DF=56
SK=99 KF=. DF=.
```

Above, the differences between the interpolation search and straight binary search are shown in boldface. Note that an extra condition with LEAVE is inserted into the code to prevent M from indexing K[M] out of the bounds of the current partition interval.

It can be proven mathematically that in the ideal case of the uniform distribution of the key-values, the interpolation search makes at most 1+log2(log2(DK)) key comparisons (where DK=dim(K)). That is, for an array with 2^20= 1,048,576 items, the number of comparisons is 5 or less, hit or miss. However, real-world keys may have quite a bit of skewness, thus making the interpolation search no better than the straight binary search and rendering the extra complexity of interpolation unjustified. On the other hand, if you know – for example,

from the nature of your business - that the key-values are distributed uniformly, using the interpolation search makes perfect sense.

An alert reader may ask: Fine, but what if the array K and the search-key SK are of the *character* data type? It is a good question. Indeed, in this case we obviously cannot perform numeric computations for the partition point M directly. The answer is that we can convert several leftmost characters of SK, K[L], and K[H] to numbers using the informat PIBw. For example, we can choose w=6 and plug INPUT(SK,PIB6.), INPUT(K[L],PIB6.), and INPUT(K[H],PIB6) into the formula for M instead of SK, K[L], and K[H], respectively. The PIB informat converts a sequence of characters (actually 256-radix digits from 0 to 255) to the corresponding decimal numbers with the same sorting order. Choosing w=6 is prudent because this is the highest PIB width at which the informat results in a number always within the integer precision of the SAS numeric variable.

PART 2: DIRECT-ADDRESSING ALGORITHMS

To this end, we have been trying to extract maximum performance from comparing a search-key SK with at least some key-values K[x] in the array, striving to reduce the number of comparisons, computations, and data movement as much as possible. However, all comparison-based methods have a *principal limitation*: For arbitrarily distributed key-values, no such method can search using fewer key comparisons than the binary search.

However, there exist several algorithms that either (a) do not rely on key comparisons at all or (b) rely on them only in the final phase after the algorithm has already reduced the potential key matches to only a few key-values. The algorithms of this type use digital properties of the keys to associate each key-value with a specific index in the array – a process termed *direct addressing*. Accordingly, these schemes are called *direct-addressing algorithms* and include the key-indexed search, bitmap search, and hash search.

KEY-INDEXED SEARCH

The claim that looking up a search-key in an array can be done without comparing it to any key-values in the array might at first seem perplexing. However, if we look closer at our sample key-values, we can notice that all of them are confined to two-digit integers. Hence, if we have an array KX sized [0:99], we can store every possible key-value in the array item *whose index is equal to the key-value* itself. In other words, going left to right through our arrays K and D, we can set:

```
KX[12]=D[1]=14, KX[12]=D[2]=15, KX[19]=D[3]=3, … , KX[90]=D[6]
```

The KX items not linked to any key-value remain missing. Of course, programmatically it would be done in a loop, for instance:

```
array KD [0:99] _temporary_ ;
do x = lbound (K) to hbound (K) ;
  KD[K[x]] = D[x] ;
end ;
```

Note that the array KD does not necessarily have to be loaded from the arrays K and D as above – instead, it can be loaded, for example, from a SAS data file or another data structure. In order to insert a value into KD, we need nothing more than a single array reference KD[<*key-value*>]. Displayed 10 items per line, the array KD will look as follows (H being the array index):

```
-------------------------------------
  H    |           KD[H]
-------+-----------------------------
 0- 9  |  .  .  .  .  .  .  .  .  .  .
10-19  |  .  . 14  .  .  .  . 15  .  3
```

```
20-29  |  9  .  . 11  .  .  .  .  .  .
30-39  |  .  .  4 19  .  .  5  .  .  .
40-49  |  .  .  .  .  .  .  .  . 10  .
50-59  |  .  .  .  .  .  .  8  .  .  2
60-69  |  .  .  .  .  .  . 16  .  .  .
70-79  |  . 13  .  .  .  .  1  .  .  .
80-89  |  .  . 17  .  7 12  .  . 18  .
90-99  |  6  .  .  .  .  .  .  .  .  .
-------------------------------------
```

Now that the search array is organized in this *key-indexed* manner, let us think how we would search it given a search-key SK=56. Looking at H=56, we see that KD[H]=8. Therefore, the search-key is found, and the corresponding data-value is 8. If we search for SK=99 instead, we see that KD[99] is null; therefore, SK=99 is not found. That is it!

If we have the array KD key-indexed as above from the outset (no matter what kind of source it is loaded from), the search routine dwindles down to a single array reference (shown below in boldface):

```
data _null_ ;
  array KD [0:99] _temporary_ (
  .  .  .  .  .  .  .  .  .  .
  .  . 14  .  .  .  . 15  .  3
  9  .  . 11  .  .  .  .  .  .
  .  .  4 19  .  .  5  .  .  .
  .  .  .  .  .  .  .  . 10  .
  .  .  .  .  .  .  8  .  .  2
  .  .  .  .  .  . 16  .  .  .
  . 13  .  .  .  .  1  .  .  .
  .  . 17  .  7 12  .  . 18  .
  6  .  .  .  .  .  .  .  .  .
  ) ;
  do SK = 56, 99 ;
    link search ;
    put (SK KF DF) (=z2.) ;
  end ;
  stop ;
  search:
  call missing (KF, DF) ;
  DF = KD[SK] ;
  if not missing (DF) then KF = 1 ;
run ;
-----------------
SK=56 KF=01 DF=08
SK=99 KF=.  DF=.
```

It should be obvious that no array lookup algorithm can perform a search faster than key-indexing since it completes it via a single array reference, regardless of whether the search is successful or not.

Also, unlike any pure key-comparison algorithm, the key-indexed search possesses a rather remarkable property: Its run time does not depend on the number of key-values "inserted" in the array. Formally, this can be expressed in the "big *O*" notation by saying that key-indexing runs in *O*(1) time or *constant* time.

It should be equally obvious that the fascinating speed and utter simplicity of the key-indexed search raises the question: Why not forego any other array-searching algorithm and use the key-indexed search exclusively? The answer is that there is no such thing as a free lunch. Key-indexing can do wonders if the key-values are integer and limited in range

from 0 to 99. It can do equally well up to when the range is much wider. For example, all realistic SAS date values are more than covered by an array KD[-200000:200000], whose memory footprint is about 400001*8=3,200,008=3 MB – a loose change with today's memories. However, if the key is, say, a 9-digit SSN, the integer key range jumps to 1E9 (1 billion) with the memory footprint of 7.45 GB. And if the key-values are of the character type, their respective 256-radix digital values already at only character length 4 reach 4,294,967,296 with the array footprint of 32 GB.

This problem can be *somewhat* alleviated if the data retrieval is not needed - that is, if we only need to know if the search-key is in the table – by using a *bitmap* based on either numeric or character arrays. A bitmap uses 1 bit rather than 8 bytes (64 bits, like in the array KD) to indicate a presence or absence of a search-key, and so it can potentially reduce the memory footprint of the key-indexed search by a factor of 64. Bitmap search in SAS is an intriguing subject for a curious SAS DATA step geek. Note that the bitmap search, just like the key-indexed search, runs in $O(1)$ time – which should come as no surprise since both are essentially based on the same direct-addressing scheme. Unfortunately, the bitmap search is too big a topic to fit in this paper; but fortunately, it is covered elsewhere rather painstakingly (Dorfman, 2019).

Fortunately, there is another, more reliable, way out of the key-indexed search's impassible key range (and thus memory footprint). It is described below.

HASH SEARCH

The central idea of the key-indexed search, which makes it so incredibly fast, is mapping the keys to their locations in the KD array based on their digital values. It is possible to keep the idea but get less radical about not making *any* comparisons between the search-key and key-values in the array. Instead of mapping every possible key-value to its own unique array index (otherwise termed *address*), we can relax this rigor and allow more than one key-value to map to the same address and – unlike with key-indexing - actually store these key-values in the array. After the array address of a search-key is obtained, we can then compare it with the keys mapping to the same address to see if there is a match.

If the mapping routine – called a *hash function* – is constructed in such a way that it maps only a few distinct key-values to the same address, it will take only a few key-comparisons to find or reject the search-key. As with the key-indexed search, we should use a fair number of *unoccupied addresses* with null-values in the array to separate the clusters of key-values mapping to different addresses. However, having about as many unoccupied addresses as occupied is usually enough to achieve decent searching performance. So, under any circumstances, we would need an array only about 1.5-2 times as large as the number of the key-values. (This solves the principal issue we face with key-indexing – the inability to work with long-range keys requiring giant memory footprints.) An array constructed in this manner is called a *hash table*. A programming routine coded to tell the keys mapping to the same address apart is called a *collision resolution policy* (because the key-values mapping to the same address are said to *collide* there).

Now let us translate all these words into a working algorithm. First, we need a hash function in the form F(key-value)=address. It should satisfy two conditions: (a) be fast and (b) distribute key-values as evenly across the hash table as possible. If the size of our hash table is HS, its index range is [1:HS], so we need a hash function mapping any key-value into the integer interval [1:HS]. This is easy to achieve by calculating the remainder of the key-value divided by HS using the MOD function. The range of its response being from 0 to HS-1, we only need to add 1 to the result to get it into our intended range [1:HS]. Since the MOD function is very fast, the condition (a) above is met. The condition (b) can be met if the denominator HS is a prime number, for it is a well-known fact that the modulo from dividing by a prime is quite random (which is why they are used in random number generators).

In turn, it means that our hash table size HS should be prime. If we decide to make the table size about 1.5 the number of key-values, i.e. 19*1.5=28.5 in our case, the closest prime is 29, so let us set HS=29. Accordingly, let us define our hash table array as HK[29] and a parallel array for the data-values - as HD[29].

As to the collision resolution policy, here we will adopt the simplest scheme called the *linear probing*. It means that if a key-value should map to an address already occupied (by a previously stored key-value), we simply store it in the next available unoccupied address. In other words, we are *probing* the addresses up by one starting with the hashed address until we have found an empty slot to insert the key-value. If during such probing the array H index should run over the top of the table, i.e., become H>hbound(HK), we merely go to the bottom of the table by setting H=1 and continue upward from there.

Translating the algorithm into the SAS language, we can load the hash table from our original arrays K and D (the code loading the table is shown in blue):

```
data _null_ ;
  array K [19] (12 17 19 20 23 32 33 36 48 56 59 66 71 76 82 84 85 88 90) ;
  array D [19] (14 15  3  9 11  4 19  5 10  8  2 16 13  1 17  7 12 18  6) ;
  array HK [29] _temporary_ ;
  array HD [29] _temporary_ ;
  do x = lbound (K) to hbound (K) ;
    do h = 1 + mod (K[x], hbound (HK)) by 1 until (cmiss (HK[h])) ;
      if h > hbound (HK) then h = 1 ;
    end ;
    HK[h] = K[x] ;
    HD[h] = D[x] ;
  end ;
run ;
```

Note again that above, the hash table HK/HD is loaded from the arrays K and D only as an example. In reality (for instance, for file matching) it is much more likely to be loaded from a SAS data file containing, say, numeric variables K and D, in which case the references K[x] and D[x] above would be replaced with K and D, respectively.

If we now display the contents of the arrays HK and HD from H=1 to 29, we will observe the following picture:

```
. 59 88 32 33 90  . 36 66  .  .  . 12 71  .  .  . 17 76 19 20 48  . 23 82  . 84 56 85
. 02 18 04 19 06  . 05 16  .  .  . 14 13  .  .  . 15 01 03 09 10  . 11 17  . 07 08 12
```

What we see is 5 clusters of key-values separated by one or more unoccupied items. No cluster has more than 5 items, which means that even in the worst-case scenario, no more than 5 key comparisons will have to be made per search. It is important that this would not change if we had more than 19 key-values because then we would merely increase the size of HK (and HD) proportionally. Therefore, the speed of the hash search does not depend on the number of key-values in the table, provided that the ratio of the number of key-values to the table size (called the *load factor*) is maintained constant. Thus, the hash search shares this property with other direct-addressing algorithms (such as key-indexing and bitmapping) and also runs in $O(1)$ (constant) time.

Searching for a search-key SK in the hash table HK/HD is principally no different from loading a key-value:

1. Hash SK to its address H.
2. If HK[h] is missing, SK in not in the table, so stop.
3. If HK[h] is not missing but HK[h]≠SK, repeatedly increment H by 1. If H>hbound(HK), set H=1. If in the process HK[h]=SK, SK is found; if HK[h] is missing, it is not found. In either case, exit the loop.

4. If HK[h] is not missing, set KF ("key-found") flag to 1 and set DF ("data-found") to HD[h].

In the SAS language:

```
data _null_ ;
  array HK [29] _temporary_ (. 59 88 32 33 90  . 36 66  .  .  . 12 71 .
                             .  . 17 76 19 20 48  . 23 82  . 84 56 85  ) ;
  array HD [29] _temporary_ (. 02 18 04 19 06  . 05 16  .  .  . 14 13 .
                             .  . 15 01 03 09 10  . 11 17  . 07 08 12  ) ;
  do SK = 56, 99 ;
    link search ;
    put (SK KF DF) (=z2.) ;
  end ;
  stop ;
  search:
  call missing (KF, DF) ;
  do h = 1 + mod (SK, hbound (HK)) by 1 until (cmiss (HK[h]) or SK = HK[h]) ;
    if h > hbound (HK) then h = 1 ;
  end ;
  if not cmiss (HK[h]) then do ;
    KF = 1 ;
    DF = HD[h] ;
  end ;
run ;
-----------------
SK=56 KF=01 DF=08
SK=99 KF=. DF=.
```

Note that there exist two other collision resolution policies, which may work a tad better than the linear probing for tight (i.e., high load factor) hash tables. A curious reader can find their descriptions and SAS implementations in (Dorfman, 2001b).

It can hardly escape the attention of an alert reader that the hash function:

```
mod (SK, hbound (HK))
```

used in this section relies on the fact that the key-values (and search-value SK) are numeric. Logically, the question will be raised what happens if the key-values are of the character data type. The simplest solution is to convert the 6 leftmost bytes of the *character* SK to an integer using the PIB6 informat and change the hash function to:

```
mod (input (SK, pib6.), hbound (HK))
```

and then proceed as usual without altering anything else in the code. Under some rare specific circumstances, it can happen that the first 6 leading characters of SK are mostly the same across all the key-values, which would make the hash function return too many collisions. In this case, we can first convert the entire SK (i.e., all its characters) into the extremely random 16-byte response of the function MD5 and only then apply the PIB6 informat:

```
mod (input (md5 (SK), pib6.), hbound (HK))
```

In most situations, though, using PIB6 alone should suffice.

CONCLUSION

The author hopes that the readers will find this array-searching compendium useful and, if necessary, extend their curiosity far enough to experiment with these techniques and acquire more erudition by reading related literature.

REFERENCES

Dorfman, Paul. 1999a. "Array Lookup Techniques: From Sequential Search to Key-Indexing (Part1)". Proceedings of the SESUG 1999 Conference, Mobile, AL: SAS Press.

Dorfman, Paul. 1999b. "Array Lookup Techniques: From Key-Indexing To Hashing (Part2)". Proceedings of the SESUG 1999 Conference, Mobile, AL: SAS Press.

Dorfman, Paul. 2001a. "Quick Sorting an Array". Proceedings of SUGI 2001 Conference, Long Beach, CA: SAS Press.

Dorfman, Paul. 2001b. "Table Lookup by Direct Addressing: Key-Indexing, Bitmapping, Hashing". Proceedings of SUGI 2001 Conference, Long Beach, CA: SAS Press.

Dorfman, Paul. 2019. "Re-Mapping a Bitmap". Proceedings of the SAS Global Forum 2019 Conference, Dallas, TX: SAS Press.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Paul M. Dorfman
sashole@gmail.com