Paper SC13

Solving Jumble Puzzles

Dictionaries, Hashes and Permutations

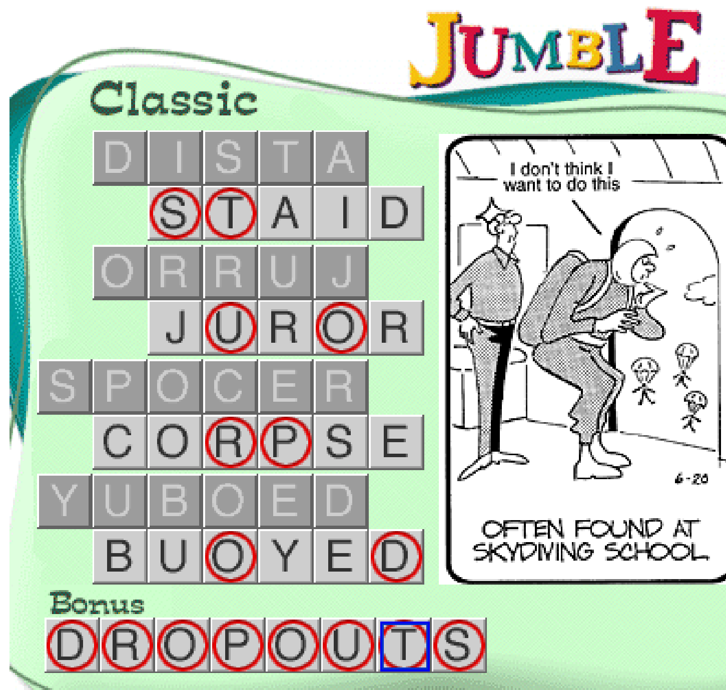
Richard A. DeVenezia, Independent Consultant

Abstract

Many systems pass through a *guilt* point in the arc of their use. This point is reached when a vastly capable and highly tuned system is focused on a trivial or personal matter. You have definitely reached this point if the phrase "world's most expensive calculator" is bandied about in conversation. What should be said when a system that is the culmination of thousands of man years of development and testing is used to solve a morning puzzle? I'm not here to make that judgement -- instead I'll be solving the Jumble, a popular newspaper word puzzle. Along the way, you will see some fancy DATA Stepping, hash objects and permutation functions.

The Puzzle

JUMBLE®, "THAT SCRAMBLED WORD GAME" by Henri Arnold and Mike Argirion is over 40 years old and appears in thousands of newspapers each day. Four scrambled words are presented to the reader. The circled letters of the unscrambled words form a scramble of the answer to a situation shown in a cartoon panel.



June 20, 2006 puzzle found at http://www.jumble.com

This paper discusses how to represent the puzzle in SAS® software and solve it using permutation looping and dictionary lookups. Discussion of artificial-intelligence approaches can be found in "Fluid Concepts and Creative Analogies"[1] by Douglas Hofstadter.

1 Douglas Hofstadter and the Fluid Analogies Research Group, *Fluid Concepts and Creative Analogies; Computer Models of the Fundamental Mechanisms of Thought*, Basic Books/HarperCollins, 1995.

Representation

Two tables will store the puzzles: Answers and Jumbles.

```
data
  jumbles (keep=date jumble circle)
  answers (keep=date answer)
;
input date yymmdd10. answer $char50.;
answer = scan (answer,1,'=');
output answers;
do i = 1 to 4;
  input jumble:$6. circle:$6. ;
  output jumbles;
end;
input;
format date yymmdd10.;
cards4;
2006-06-20 ---- ----
DISTA OO---
ORRUJ -O-O-
SPOCER --OO--
YUBOED --O--O
```

Each letter of the answer is represented by a dash. The words of the answer are separated by a space. The letters of the jumbled word are listed. Each letter of the unscrambled word is represented by a dash or capital O.

General Approach

The letters of each word are permuted until a dictionary match is made. The letters falling in an O position are placed in an answer pool. When all the words have a dictionary match, the letters in the answer pool are permuted until each word in the answer has a dictionary match.

Permutations

The letters of each word are permuted...

The CALL ALLPERM routine can be used to permute the elements of an array.

```
data _null_;
  array x[3] (1:3);
  do k = 1 to fact(3);
    CALL ALLPERM (k, of x[*]);
    put k @2': ' x[*];
  end;
run;
```

```
LOG
1: 1 2 3
2: 1 3 2
3: 3 1 2
4: 3 2 1
5: 2 3 1
6: 2 1 3
```

SAS software documentation states as follows: "Because each permutation is generated from the previous permutation by a single interchange, the algorithm is very efficient." The colorized log demonstrates the interchanges of N=3.

Lexicographic order

The interchange algorithm does not generate permutations in lexicographic order.

http://www.cut-the-knot.org/do_you_know/AllPerm.shtml#lexicographic states as follows:

> Permutation f precedes a permutation g in the lexicographic (alphabetic) order iff for the minimum value of k such that f(k) ≠ g(k), we have f(k) < g(k). Starting with the identical permutation f(i) = i for all i, the second algorithm generates sequential permutations in the lexicographic order. The algorithm is described in [Dijkstra[2], p. 71].
>
> Program 2. *void getNext() ...implementation of algorithm as described by Dijkstra*

The benefit of lexicographic ordering is that large numbers of subsequent permutations can be avoided when a permutation fails some prefix test. For the Jumble puzzle, the test is "Does the word, or word portion, constructed from this permutation correspond to a word or word portion in the dictionary?"

Program 1 at http://www.scielo.br/scielo.php ? pid=S0104-65002001000200009 & script=sci_arttext[3] is nearly identical to Program 2. attributed as described by Dijkstra.

A blending of the next permutation algorithm exhibited by the two programs, translated into SAS software DATA Step syntax for a 1-based array, is as follows:

```
next_perm:
  i = n;
  do while (i > 1);  if (v[i-1] <= v[i]) then leave; i + (-1); end;
  if i = 1 then return;

  j = i + 1;
  do while (j <= n); if (v[i-1] >= v[j]) then leave; j + 1; end;

  * swap ;
  ix1 = i - 1; ix2 = j - 1; h = v[ix1]; v[ix1] = v[ix2]; v[ix2] = h;

  * reverse v[i..n];
  ix1 = i; ix2 = n;
  do while (ix1 < ix2);
    h = v[ix1]; v[ix1] = v[ix2]; v[ix2] = h;
    ix1 + 1; ix2 + (-1);
  end;
return;
```

The sequence generated for N=4 is as follows (coloring explained in next section):

```
1 2 3 4     2 1 3 4     3 1 2 4     4 1 2 3
1 2 4 3     2 1 4 3     3 1 4 2     4 1 3 2
1 3 2 4     2 3 1 4     3 2 1 4     4 2 1 3
1 3 4 2     2 3 4 1     3 2 4 1     4 2 3 1
1 4 2 3     2 4 1 3     3 4 1 2     4 3 1 2
1 4 3 2     2 4 3 1     3 4 2 1     4 3 2 1
```

2 Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, 1997 [FACSIMILE] (Paperback).

3 ITAI, Alon. Generating permutations and combinations in lexicographical order. J. Braz. Comp. Soc., 2001, vol.7, no.3, p.65-68. ISSN 0104-6500.

Color coding

Let P be a permutation that fails some prefix rule of length L
Let P* be the next lexicographical permutation having a prefix of length L different from P.
Let P^ be permutation prior to P*.

P is white on blue, P* is black on yellow, and P^ is black on cyan.

Simple inspection shows that the length N-L suffix of P^ is sorted in descending order. Thus, to advance from P to P* one must sort the N-L tail of P in descending order to achieve P^ prior to invoking the next_perm code block. One simple algorithm for sorting a segment of an array, when the initial values of the array are v[i] = i, is as follows:

```
  array w[N] (N*0);
  do i = LEFT to RIGHT;
    w [ v [ i ] ] = 1;
  end;
  j = 0;
  do i = 1 to N while (LEFT + j <= RIGHT);
    if w[i] = 0 then continue;
    v [ RIGHT - j ] = i;
    j + 1;
  end;
```

Dictionary

Word lookups are a form of spell checking. SAS software contains built-in spell checking: the documented SPELL command and the undocumented SPELL procedure. These features are insufficient for word lookup from within a DATA Step. There are no functions such as SPELL or ISWORD. There is, however, the hash object that "stores and retrieves data based on lookup keys."[4] The keys for the Jumble program will be dictionary words and word prefixes. There will be no associated data; the presence of a key alone will serve as a positive lookup.

Word lists can be downloaded from many places on the Internet. The open source repository sourceforge.net has several in group_id=10079 & package_id=9919. Jumble uses the agid-4 zip containing file "infl.txt". The file is read into a SAS table with the following code:

```
%let userdir = %sysget (USERPROFILE);
data _null_;
  length word $25 pos $10;

  declare hash words (ordered:'A');
  words.defineKey ('word');
  words.defineData ('word');
  words.defineDone ();

  filename rawdict "&userdir.\My Documents\dictionaries\infl.txt";
  do while (not end);
    infile rawdict dlm=' ,|' missover end=end;

    input word pos @;
```

4 SAS Documentation – "Using the Hash Object"

```
    do until (word='');
      word = compress (word,'~<!?');
      if not indexc (word,'123456790.{}') then
        words.replace();

      input word @;
    end;
    input;
  end;

  words.output (dataset:'sasuser.agid_dictionary');
  stop;
run;
```

A hash object is used to accumulate a list of all words in the dictionary, and the output method is used to create the permanent SAS table "sasuser.agid_dictionary" for later use. The code needs to be run only once, or whenever the infl.txt file is updated.

Solve Macro

The %solve macro wraps the entire problem solving system. This is necessary because SAS DATA Step code needs to be generated.

```
%macro solve (date, dictionary = sasuser.agid_dictionary);
  %local datev ;
  %let datev = %sysfunc(inputn(&date.,yymmdd10.));
```

The following puzzle data will be used in explanations:

```
2005-07-12 ------- ----
PALLE  O-O--
CHEEN  O-OO-
THAAMS O--O-O
PECDIT OO---O
```

```
%solve (2005_07_12, dictionary=sasuser.agid_dictionary)
```

Unscrambling

The letters of each word are permuted until a dictionary match is made (e.g. Each jumble is unscrambled).

The number of permutations of the clue words is 1,680 (2x5!+2x6!) which is relatively small. The code below uses a loop that iterates over every permutation of each word.

Consider the jumble "a-z-o-k-o", two different permutations result in the word "k-a-z-o-o." An unscrambled word that is repeated in the output table (work.jumble) will cause duplicates in the answer pools that are examined later. Thus, a hash (named "cand" for candidate) is used to ensure an unscrambled word will not be repeated.

```
data jumble
( label="Permutations of Clues that are Words"
  keep=_i word circled circle wurd /* wurd is for displaying only */
);
```

```
declare hash dict ();               * hash to contain dictionary words;
dict.defineKey ('word');
dict.defineDone ();

declare hash cand ();               * hash to contain candidates, i.e. the ;
cand.defineKey ('word', '_i');      * unscrambled clue words found in dictionary;
cand.defineData ('word', '_i');
cand.defineDone ();

length word circled $6;

* Load the 5- and 6- letter words from the dictionary into hash;
do until (end_dict);
  set &dictionary. (where=(length(word) between 5 and 6)) end=end_dict;
  if dict.check() eq 0 then continue;
  dict.add();
end;

* Examine all permutations of each clue word (jumbled word);
do until (end_jumble);
  set jumbles end=end_jumble;  * get next clue from puzzle table;
  where date = &datev.;         * for the date of interest;
  _i + 1;                      * _i`th clue (there are four);
  jumble = lowcase(jumble);    * dictionary lookups are lowercase;
  link allperm;                * jump to section that examines all permutations;
end;

put nn=;  * should be 1,680 ;
stop;
```

The **allperm** section uses POKELONG and PEEKCLONG functions to move bytes between "letters," a DATA Step array, used with the CALL ALLPERM routine and "word," the PDV variable which is a key variable in the two hashes prepared earlier.

allperm:

```
* Clues are 2 5-letter and 2 6-letter words;
* Load the letters of a clue (jumbled word) into an array;
array letters $1 letters1-letters6 ( 6 * ' ' );
L = length (jumble);
call pokelong (jumble,addrlong(letters[1]));

* Check each permutation of the letters of the clue word;
do i = 1 to fact (L);

  * Next permutation of the letters;
  if L = 5 then call allperm (i, of letters1-letters5); else
  if L = 6 then call allperm (i, of letters1-letters6);
  nn + 1;

  * Load the array permutation into pdv hash host variable "word";
  word = peekclong (addrlong (letters(1)), L);
```

```
    * If the word is in the dictionary and is a new candidate
    * store it in the hash for later lookup and OUTPUT it and the circled letters;
    if word ne jumble
       and dict.check () = 0
       and cand.check () ne 0
    then do;
      k = 1;
      circled = '';     * circled letters of the unscrambled word;
      wurd = word;      * "wurd" is the word with the circled letters in uppercase;
      do j = 1 to length (circle);
        if substr(circle,j,1) = 'O' then do;
          substr(circled,k,1) = substr(word,j,1);
          substr(wurd,j,1) = upcase(substr(wurd,j,1));
          k + 1;
        end;
      end;
      put jumble= word= circled
      OUTPUT;
      cand.add();
    end;
  end;
return;
```

```
LOG
jumble=palle  word=lapel  circled=lp
jumble=cheen  word=hence  circled=hnc
jumble=thaams word=asthma circled=aha
jumble=thaams word=matsah circled=msh
jumble=pecdit word=depict circled=det
nn=1680
```

Jumbled words can unscramble to more than one word (as does "thaams") .

Combinations

Each combination of four unscrambled words will create a different pool of circled letters available for the answer.

```
* pool contains one row per combination of unjumbled words;
* if all jumbled words have only one unjumbled match, pool will have only one row;
proc sql;
  select answer into :answer from answers
  where date = &datev;

  create table pool as
  select a.circled as A
       , b.circled as B
       , c.circled as C
       , d.circled as D
       , a.wurd as _A length=6
       , b.wurd as _B length=6
       , c.wurd as _C length=6
       , d.wurd as _D length=6
  from   jumble as a
       , jumble as b
       , jumble as c
       , jumble as d
  where  a._i = 1
```

```
    and  b._i = 2
    and  c._i = 3
    and  d._i = 4
  ;
quit; %put answer=&answer;
```

VIEWTABLE: Work.Pool

	B	C	D	_A	_B	_C	_D
1	hnc	aha	det	LaPel	HeNCe	AstHmA	DEpicT
2	hnc	msh	det	LaPel	HeNCe	MatSaH	DEpicT

The permutations of all of these pools need to be searched for dictionary words that match the letter lengths of the cartoon answer. Unabridged dictionaries contain more words than other dictionaries and will tend to have multiple unscrambled words, leading to more combinations, which can result in many "final answers". From the answers, one or more can be picked by a human as matching the cartoon panel.

Answer guide

```
answer= ------- ----
```

The answer guide is examined and parsed into word lengths and numbers of words. The lengths and counts are used to generate DATA Step statements in the final section of the solve macro.

```
%* examine the answer guide
%* determine number of and length of each dash group
%* wordlens is used later as part of a where IN clause
%*;

%local N maxlen wordlens len0 word;

%let N = 1;
%let len0 = 0;
%let maxlen = 0;

%do %while (%qscan(&answer,&N,%str( )) ne );

  %let word = %qscan(&answer,&N,%str( ));

  %local len&N;
  %let len&N = %length (&word);

  %let wordlens = &wordlens , &&len&N;

  %if &&len&N > &maxlen %then %let maxlen = &&len&N;

  %let N = %eval (&N+1);
%end;

%* number of words in answer;
```

```
%let N = %eval (&N-1);

%* number of letters in answer;
%let L = %length (%sysfunc(compress(&answer,%str( ))));

%put wordlens=&wordlens maxlen=&maxlen N=&N L=&L;
```

The sample puzzle yields this log information:

```
wordlens=, 5 , 7 maxlen=7 N=2 L=12
```

More permuting

The L letters in the answer pool must be arranged into N words of length L_1 ... L_N. There are L! permutations to be considered, but large numbers of them can be excluded with appropriate lookup tests.

Mapping

The active pool is the letter arrangement corresponding with the permutation P (maintained in array "indx") being examined. The pool is computed by mapping the original pool (or $pool_0$) through indx in the following manner:

```
  do ix = 1 to dim(pool);
    pool [ ix ] = pool0 [ indx [ ix ] ] ;
  end;
```

Loop index - ix	***Original Pool***	***Permute Array - indx***	***Active Pool***
1	w	3	r
2	o	2	o
3	r	1	w
4	d	4	d

Suppose the cartoon is a policeman with a notepad, standing next to a curmudgeon who is scowling at a parade of youngsters walking by. The anwer key is "- - - - - - - - - - - -." A 12-letter answer pool "z-k-t-i-f-g-o-o-r-e-a-h" is available. The pool can be permuted 12!, or 479,001,600 ways (half of which are duplicates due to the two ohs). The time required to CALL ALLPERM 4.8E8 times and perform at least 4.8E8 hash check()s is longer than most people are willing to wait.

Shortcuts

The number of permutations that are selected for examination can be greatly reduced. If a word, W, constructed from the active pool is not in the dictionary, the next permutation that needs to be examined is the one that lexicographically follows the smallest prefix of W that is not in the prefix dictionary.

Suppose P is 1-2-* and corresponds to z-k-*. There are no 4- or 7- letter words in the dictionary starting with z-k. Thus, compute P^ as 1-2-*<descending> and invoke next_perm to reach P* (1-3-* and active pool z-t-*)

Last step

The last data step utilizes everything discussed previously. There is an additional hash , named "soln," that is used to record solutions and the unjumbled words from which the answer arose.

```
data _null_;
  * word variable for dictionary existence via hash check();
  length word prefix $&maxlen. ;

  * answer words;
  length %do i = 1 %to &N.; word&i. $&&len&i %end; ;

  array words word1-word&N.;
  array wordlens[0:&N.] (0 &wordlens.);

  * hash as dictionary of words;
  declare hash dict ();
  dict.defineKey ('word');
  dict.defineDone ();

  * hash as dictionary of word prefixes;
  declare hash part ();
  part.defineKey ("length", "count", "prefix");
  part.defineDone ();

  * hash for solutions;
  declare hash soln (ordered:'a');
  soln.defineKey  ('combo' %do i = 1 %to &N.; , "word&i." %end; );
  soln.defineData ('combo' %do i = 1 %to &N.; , "word&i." %end;
                                      , "_A", "_B", "_C", "_D");
  soln.defineDone ();

  * populate hashes acting as dictionaries, only need to consider words
  * of length corresponding to word lengths in the answer guide;
  do until (end_dict);
    set &dictionary. (where=(length(word) in (-1 &wordlens.))) end=end_dict;

    * add word to dictionary hash;
    if dict.check() eq 0 then continue;
    dict.add();
    _ndict+1;

    * add word prefixes to prefix dictionary hash;
    length = length(word);
    do count = 2 to length-1;
      prefix = substr(word,1,count);
      if part.check() eq 0 then continue;
      part.add();
      _npart + 1;
    end;
  end;
```

```
put (_ndict _npart)(=/);  * log the number of items in the hashes;

array pool0[&L.] $1;  * letter pool - original;
array pool [&L.] $1;  * letter pool - active perm;
array indx [&L.]   ;  * permutation indices;
array circles [4] $6 a b c d;

* add letters to letter pool and set up permutation array;
do combo = 1 by 1 until (end);
  set pool end=end;
  ix = 1;
  do i = 1 to dim(circles);
    do j = 1 to length (circles[i]);
      pool0[ix] = substr (circles[i],j,1);
      pool [ix] = pool0[ix];
      indx [ix] = ix;
      ix + 1;
    end;
  end;

  * log what is being searched;
  put / _a _b _c _d '-> ' a b c d ;

  * Note: i=1 when control returns from the next_perm: section and there are
  * no more permutations in the lexicographic sequence.
  * Thus i=1 is the loop exit test;

  do until (i=1);

    p = 1;  * start position of active candidate word in letter pool;

    do q = 1 to &N.; * loop over each word in the answer guide;

      * extract candidate word from letter pool (of some permutation P);
      length = wordlens[q];
      word = peekclong (addrlong (pool(p)), length);

      if dict.check() ne 0 then do;
        * word not found, find smallest prefix not found;
        do count = 2 to length-1;
          prefix = substr(word,1,count);
          if part.check() ne 0 then leave;
        end;

        * Sort suffix of P by descending order to reach P^;
        * (so next_perm advances to P*);

        array map[&L.];
        call missing (of map[*]);
```

```
        LEFT = p + count;
        RIGHT = &L.;
        do ix = LEFT to RIGHT;
          map [ indx [ ix ] ] = 1;
        end;
        j = 0;
        do ix = 1 to &L. while (LEFT + j <= RIGHT);
          if not map[ix] > 0 then continue;
          indx [ RIGHT - j ] = ix;
          j + 1;
        end;

        LEAVE; * the DO q loop;
      end;

      * record the word that was found in the dictionary;
      words[q] = word;

      * advance p to start of next candidate word;
      p + length;
    end; * of DO q loop;

    * does the permutation map to a complete answer not previously recorded?;
    if (q > &N.) and (soln.check() ne 0) then do;
      * record the answer;
      soln.add();
      * log the answer;
      put +2 words[*]; * / (pool[*]) ($char3.-r) / (indx[*]) (3.);
    end;

    link next_perm;
    next_perms+1;    * count number of permutations examined;
  end;
end;

put / next_perms=;
soln.output(dataset:"solution_&date.");    * store results in a table;
stop;

next_perm:
  i = &L.;
  do while (i > 1);  if (indx[i-1] <= indx[i]) then leave; i + (-1); end;
  if i = 1 then return;

  j = i + 1;
  do while (j <= &L.); if (indx[i-1] >= indx[j]) then leave; j + 1; end;

  * swap ;
  ix1 = i-1; ix2 = j-1;
```

```
  h = indx[ix1]; indx[ix1] = indx[ix2]; indx[ix2] = h;

  * reverse v[i..n];
  ix1 = i; ix2 = &L.;
  do while (ix1 < ix2);
    h = indx[ix1]; indx[ix1] = indx[ix2]; indx[ix2] = h;
    ix1 + 1; ix2 + (-1);
  end;

  * map original pool to active pool;
  do ix = 1 to dim(pool);
    pool [ ix ] = pool0 [ indx [ ix ] ] ;
  end;
return;

run;
```

Conclusion

New features in SAS software version 9, which include the Hash object, the CALL ALLPERM, routine and the PEEKCLONG function, can be utilized to solve a variety of combinatoric problems. A solid solution requires good problem representation and detailed knowledge of how both old and new features operate. Solving a JUMBLE a day exercises the cobwebs away. Programming a JUMBLE solver provides a real workout.

About the Author

Richard A. DeVenezia has previously presented papers at SUGI, SESUG and NESUG, and is an active contributor on SAS-L. He is an independent consultant with fifteen years of SAS experience. He has worked with an extensive mix of SAS products in a variety of industries, including manufacturing, retail and pharmaceutical companies.

The entire %solve macro source code can be found at the author's website. Visit http://www.devenezia.com and follow the link to Papers.

Recommended Reading

DATA Step Hash Objects as Programming Tools, SUGI 31 Proceedings, Paul Dorfman and Koen Vyverman

DATA Step Hash Objects as Programming Tools, SUGI 30 Proceedings, Paul Dorfman and Koen Vyverman

Hash Component Objects: Dynamic Data Storage and Table Look-Up, SUGI 29 Proceedings, Paul Dorfman and Koen Vyverman

An Animated Guide: Hashing in V9.1, NESUG 2005 Proceedings, Russell Lavery, Independent Consultant

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

This document was produced using OpenOffice.org Writer.