Paper 052-2012

The DOW-loop: a Smarter Approach to your Existing Code

Fuad J. Foty, U.S. Census Bureau, Suitland, MD

ABSTRACT

Following the frequent use of the DOW-loop's logic, it became apparent to me that the DOW-loop structure could be easily extended to a greater percentage of existing SAS® code. Some SAS® code includes PROC SQL statements performed on large datasets that can easily be replaced by a DOW-loop construct. Other SAS code includes PROC SUMMARY datasets that can be simply aggregated and merged into a new SAS dataset without having to leave the DATA step. In addition to reading the code in a more logical way, the process enables faster and more efficient performance. In this paper, I examine existing SAS code examples and discuss how to convert them to include this new and exciting DOW-loop method.

INTRODUCTION

The DOW-loop is an abbreviation of a "DO" and a "W" combined together. The "W" refers to the first letter of Dr. Ian Whitlock's last name, which in February 16, 2000, responded to an inquiry on the University of Georgia's SAS-L listserv about a way to access values of variables, especially when there is more than one value to deal with. The solution to that famous question was a simple array filling and manipulation using the DO UNTIL (LAST.ID) construct. The "DO" either refers to the DO-loop or as suggested by Dorfman and Shajenko's paper SS-01, indicates the first two letters of Don Henderson's first name. Ian Whitlock may have heard about this looping construct through Mr. Henderson as early as the late eighties. While it is unclear whether the notion originated with one sole person, for the purposes of this paper, I would like to recognize all of those who contributed and developed the DOW-loop method over the past two decades. Having said that, let us consider next the details of the main scheme of the DOW-loop DATA step structure.

ASSUMPTIONS AND PREREQUISITES

Rather than go through Paul Dorfman's elegant and thorough examples presented in his previous papers, I would rather ask the reader to simply read his **SS-01** paper mentioned above and referenced in this paper in order to properly understand each DOW-loop example presented in this paper.

There are several ways to start thinking about the logic behind the DOW-loop. Think about grouping the code components into before, during, and after code-subgroups. It is important to understand how the DOW-loop is nested within the implied data-step-loop and why there is no need to retain summary-variables across the observations. When the DOW-loop iterates, it holds values within each BY group and outputs a single record per BY-group as required, without the need of any conditional statements. When program-control is passed back to the top of the implied loop, the non-retained variables are reset to missing values or in other words naturally reinitialized.

It is also important to understand the "DO _N_ = 1 By 1 UNTIL …" DOW-loop construct. It is another clever way to loop using _N_ SAS index variable. The particular example to pay attention to in Dorfman's paper is the "dow_agg" dataset and how the "**DO _N_ = 1 by 1 UNTIL (LAST.ID)**" DOW-loop has been used to sum up the values of "**num**" into "**summa**" for each BY group. In addition, you should know how to calculate the average (**summa/_n_**) after the DOW-loop.

Reviewing the above mechanisms and examples will help the reader understand the examples that will be presented in this paper.

EXAMPLE 1: PROC SUMMARY AND THE DOUBLE DOW-LOOP

The PROC SUMMARY SAS procedure is widely used here at the U.S. Census Bureau especially when trying to aggregate statistical survey data using well defined BY groups. The SAS software has been around for quite some time here, yet a large number of SAS programmers struggle through the logic of the SAS loop construct. Developers and statisticians love the PROC SUMMARY for its simplicity but have a hard time constructing it and usually take quite some time to tell PROC SUMMARY exactly what they want. The CLASS and the VAR statements in the PROC SUMMARY are easily confused.

Let us consider an existing SAS-code example of a PROC SUMMARY specified in the following box:

The Input SAS Dataset:

SUMFILE has **157893** observations and **23** variables sorted by
SFLAG, STATE, COUNTY, STRATUM, MFLAG

The Old code:

```
proc summary nway data= SUMFILE missing;
  class sflag state county stratum mflag;
  id cpos rs mos;
  var selected mos cpos;
  output out= CCS(drop=_type_ _freq_)
  n(selected) = celig
  sum = csamp cexpr cexp;
run;
```

Sort and Merge the Summary variables back to the SUMFILE:

```
Proc sort data= CCS;
  by sflag state county stratum mflag;
run;

data SUMFILE_AGG;
  merge SUMFILE (in=a)
        CCS  (in=b);

  by sflag state county stratum mflag;

  if a then output;
run;
```

The Output SAS Dataset:

The dataset **CCS** has **10953** observations and **12** variables.
The dataset **SUMFILE_AGG** has **157893** observations and **23** variables.

The example code above consists of a PROC SUMMARY statement on a sorted SAS dataset, which outputs **four** summary variables (CSAMP, CEXPR, CEXP and CELIG) that get added to the aggregate file (SUMFILE_AGG) after a sort and a merge of those datasets. Notice that there are two SAS datasets that are outputted after a sort and a merge. If we can avoid the sort and the merge, then this whole process will go much faster, hence, the DOW-loop.

Now let us look at the alternative **Double DOW-loop** method:

```
The Double DOW-loop method: (Note: SUMFILE would need to be sorted first)

Line01.  data SUMFILE_AGG
Line02.       CCS(keep=sflag state county stratum mflag
Line03.           cpos rs mos celig csamp cexp cexp);
Line04.
Line05.    do _n_ = 1 by 1 until (last.mflag);
Line06.       set SUMFILE;
Line07.       by sflag state county straturm mflag;
Line08.       csamp = sum (csamp, selected);
Line09.       cexpr = sum (cexpr, mos);
Line10.       cexp  = sum (cexp,  cpos);
Line11.       celig = sum(celig ,1);
Line13.    end;
Line14.
Line15.    output CCS;
Line16.
Line17.    do _n_ = 1 by 1 until (last.mflag);
Line18.       set SUMFILE ;
Line19.       by sflag state county stratum mflag;
Line20.       output SUMFILE_AGG;
Line21.    end;
Line22.
Line23.  run;
```

Analyzing the above code, one can see that there are two identical SET statements wrapped in their own DOW-loop. One must keep in mind that the above two SUMFILE datasets in the DOW-loops are the equivalent of having two independent SAS datasets (SUMFILE1 and SUMFILE2). Below are the detailed SAS DATA step program controls (PC):

PC starts the implied loop at line one (Line01). This is the top of the DATA step where we are defining two output SAS datasets, **SUMFILE_AGG** and **CCS**. All non-retained variables are set to missing values, and _N_ is set to one (1).

PC enters the first DOW-loop at line five (Line05) and proceeds to iterate through the first BY group (sflag state county stratum mflag).

Every time PC hits the SET statement (**Line06**), it reads the next record from SUMFILE1, which is the first virtual copy of dataset SUMFILE. When PC finishes and exits the DOW-loop, all **157893** records from SUMFILE1 have been read, and the statistics for that BY group is accumulated for the summary variables (CSAMP, CEXPR, CEXP, CELIG).

PC proceeds to line fifteen (Line15) where the SAS dataset, CCS, is outputted.

Next, PC enters the second DOW-loop and hits the SET SUMFILE statement on line eighteen (Line18). Now, the SET statement reads a record from SUMFILE2, which is the **second virtual copy** of SUMFILE. Since nothing has been read from that copy yet, the SET statement reads the first record from the first BY group (sflag state county stratum mflag).

However, in this second DOW-loop, instead of computing the sum, we have the explicit OUTPUT statement, so it releases the current program data vector (PDV) content to SUMFILE_AGG.

Please note that before the loop started, summary variable (CSAMP, CEXPR, CEXP, and CELIG), had been already computed and are now sitting in the PDV.

The iterations of the second DOW-loop have no effect on their values, since they are modified neither inside this DOW-loop nor by the implied DATA step loop. Hence, the record is output with its current values read from SUMFILE2, and the aggregate values computed before the loop started.

The second DOW-loop then continues to iterate, reading the rest of the records from the SUMFILE2's first BY group, spitting them out to SUMFILE_AGG together with the aggregates computed beforehand.

At the end of the second loop, the SUMFILE_AGG will contain the SUM variables (CSAMP, CEXPR, CEXP, and CELIG) for every member of the BY group.

Now, PC is at the bottom of the implied DATA step loop, and both buffers SUMFILE1 and SUMFILE2 have been lowered by the record count with the first BY group.

The implicit OUTPUT statement is turned off due to the fact that the explicit OUTPUT statement is present elsewhere in the DATA step (in the second DOW-loop). Therefore, no output happens at this time, and PC simply loops back to the top of the implied loop.

The non-retained aggregates variables (CSAMP, CEXPR, CEXP and CELIG) are set to missing values, _N_ is set to **two** (2) and PC **repeats** the same action as it performed in the first iteration of the implied loop:

- reads the 2nd BY group from buffer SUMFILE1 and computes aggregates using the first DOW-loop
- then reads the 2nd BY group from buffer SUMFILE2, writing out each record to SUMFILE_AGG
- Finally, PC exits the second DOW-loop and goes back to the top of the DATA step, at which point, the second BY group has been added to SUMFILE_AGG.

When PC arrives at its final BY-group content at the top of the implied loop, PC now enters the first DOW-loop for the LAST+FIRST time when all records from both buffers (virtual files) SUMFILE1 and SUMFILE2 have been read, and the buffers are empty.

The SET statement in the first DOW-loop tries to read from the empty buffer SUMFILE1, and it immediately causes the step to STOP.

While we now can see how the Double DOW-loop construct operates, the question still remains as to whether it is a more efficient procedure or not.

The PROC SUMMARY method for the above example took **1.06 real (0.41 cpu)** seconds to execute, while the Double DOW-loop method took only **0.24 real (0.21 cpu)** seconds, which is about **442** (or **195**) **percent** faster and more efficient in **real time** (or **cpu time**) measure. In either case, real or cpu time, the Dow-loop method beats the old method hands down.

EXAMPLE 2: THREE SUMMARIES AND THREE MERGES VERSUS A DOW-LOOP

In this example, you will see how to simplify **three** PROC SUMMARIES and **three** temporary SAS datasets **merged** via a DATA step using a **one variable BY** group (StateCode). The example summarizes weights per FIPS state numeric code, **StateCode** (United States, District of Columbia, and Puerto Rico "72"). It calls a random number, which gets used to calculate the sampling interval (**SI**), a straightforward generalized procedure that is utilized in various surveys here at the Census Bureau.

Let us consider now the existing SAS-code for EXAMPLE2 in the following box:

The Input SAS Dataset:
HWR has **1940293** obs and **2** variables (StateCode and whrf) sorted by **StateCode**
SAMPS has **1950** obs and **2** variables (StateCode and hu10) sorted by **StateCode**

The Old Code:

```
libname mydir "";
proc summary nway data=mydir.hwr;
   class StateCode;
   var whrf;
   output out=univs (drop= _type_ _freq_) n=univ;
run;
proc summary nway data=mydir.samps;
   class StateCode;
   var hu10;
   output out=sampsize sum=hu;
run;
proc summary nway data=mydir.hwr(where=(statecode='72'));
   class StateCode;
   var whrf;
   output out=sampsizepr sum=hu;
run;

data mydir.parms (keep=statecode hu survsamp si rs univ);
   merge sampsize sampsizepr univs;
   by StateCode;
   survsamp = ceil(.01*hu);
   si = univ / pumssamp;
   seed = input('7812223'||statecode,9.);
   call ranuni(seed,ran);
   rs = si*ran;
run;
```

The Output SAS Dataset:
The dataset **PARMS** has **52** observations and **6** variables.

As you can see, the above code consists of a three PROC SUMMARIES and three temporary SAS datasets MERGED together by StateCode. It outputs a dataset by StateCode, which contains the number of housing units in the survey. The calculation identifies the universe of housing units in the survey and the actual sample size from those universes with their associated sampling intervals.

Now let us take a look at the alternative **DOW-loop** method:

The **DOW-loop** method: **(Note: HWR and SAMPS would need to be sorted first)**

```
Line01. libname mydir "";
Line02.
Line03. data mydir.PARMS;
Line04.   keep statecode hu survsamp si rs univ;
Line05.
Line06.   do _n_ = 1 by 1 until (last.statecode);
Line07.      set mydir.HWR;
Line08.      by statecode;
Line09.      univ = sum (univ, 1);
Line10.      if statecode eq '72' then do;
Line11.         hu = sum (hu, whrf);
Line12.      end;
Line13.   end;
Line14.
Line15.   if statecode ne '72' then do;
Line16.      do _n_ = 1 by 1 until (last.statecode);
Line17.         set mydir.SAMPS;
Line18.         by statecode;
Line19.         hu = sum (hu, hu10);
Line20.      end;
Line21.   end;
Line22.
Line23.   survsamp = ceil(.01*hu);
Line24.   si = univ / pumssamp;
Line25.   seed = input('7812223'||statecode,9.);
Line26.   call ranuni(seed,ran);
Line27.   rs = si*ran;
Line28. run;
```

The above code shows that there are two DOW-loops with certain logic relating to **Statecode** being “72” or not “72” (Puerto Rico or U.S). Unlike Example1, this example outputs one SAS dataset (**PARMS**) that is generated as a result of the calculations in both DOW-loops. Below are the detailed SAS DATA step program controls (PC) for this example:

PC starts at line one (Line01) where the libname gets assigned. PC goes to line three (Line03) where the DATA step starts by assigning an output SAS dataset PARMS. At line six (Line06), the first DOW-loop starts using StateCode variable BY statement. The loop reads **HWR** on line seven (Line07) and creates a variable UNIV which calculated the number of observations in that BY statement. If StateCode is “72” (Puerto Rico), then it sums up the values of WHRF (household weight) in a new variable called **HU**. PC keeps looping from line six (Line06) to line thirteen (Line13), calculating the values for variables UNIV and HU until all State Codes (last StateCode) have been exhausted. PC moves to line fifteen (Line15) where StateCode **is not equal** to “72” (U.S. only). If that is true, a second DOW-loop starts at line sixteen (Line16) using the same BY variable that is used in the previous DOW-loop (StateCode) on SAS dataset SAMPS. Here, variable HU is being aggregated by summing up the values of HU10 on the SAMPS dataset in that BY variable (StateCode). PC goes now to line twenty-three (Line23) through twenty-seven (Line27) where variable SURVSAMP, SI, SEED, RAN and RS are created at the conclusion of both DOW-loops. At line twenty-eight (Line28), one numeric StateCode observation gets outputted to **PARMS**. PC moves back to line three (Line03) and repeats the entire process again for another StateCode BY variable.

The old code for the above example took **1.03 real (0.77 cpu)** seconds to execute, while the double DOW-loop method took only **0.33 real (0.30 cpu)** seconds, which is about **312** (or **257**) **percent** faster and more efficient in **real time** (or **cpu time**) measure. Again, the Dow-loop method beats the old method for the second time.

EXAMPLE 3: PROC TRANSPOSE VERSUS DOW-LOOP

For each variable specified in the VAR statement of the PROC TRANSPOSE, a one record per variable in the output SAS dataset is produced. We might need a single record per BY statement outputs instead of every observation. We can do this by using the DOW-loop method.

Suppose, we have a SAS dataset, which contains one variable, specified in the VAR statement of the PROC TRANSPOSE. The input dataset has seven variable, six of those variable are use in the BY statement and one variable is used in the VAR statement.

Let us take a closer look at the following box which contains the PROC TRANSPOSE:

```
The Input SAS Dataset:
   The dataset POP_SMRACE has 62155 observations and 7 variables
       (irank1 - irank5 survest smrace).
   Sorted by: SURVEST IRANK1 IRANK2 IRANK3 IRANK4 IRANK5

The Old Code:

libname mydir "";

proc transpose data=mydir.pop_smrace
               out=mydir.srace(drop=_name_)
               prefix=smrace;
   by survest irank1 irank2 irank3 irank4 irank5;
   var smrace;
run;

The Output SAS Dataset:
   The dataset SRACE has 2005 observations and 37 variables
   (survest rank1-rank5  smrace1-31).
```

PROC TRANSPOSE requires **three** passes through the entire dataset to accomplish its final transposed output. It has to read the data first, split it, and then merges it back. For small SAS datasets, it might not matter much, but if you have a large SAS dataset, it will take a much more significant execution time and resources to do the job.

Alternatively, we can choose the DOW-loop method. Initially we thought a one DOW-loop is sufficient to do the entire job. However, we would like to dynamically calculate the dimension of the array, which holds the variables that are being created during the execution of the transposing process. For that reason, we need to use a DOW-loop to figure out the dimension of the array that is going to hold those variables and then use that dimension in a second DOW-loop that fills the array values based on the SAS dataset that is being pushed through.

Below is the actual code to do that:

The **DOW-loop** method: **(Note: POP_SMRACE would need to be sorted first)**

```
Line01. libname mydir "";
Line02. data _null_ ;
Line03.    do until (last.irank5);
Line04.       set mydir.pop_smrace end=done;
Line05.       by survest irank1 irank2 irank3 irank4 irank5;
Line06.       i=sum(i,1);
Line07.       if i gt max_i then max_i = i;
Line08.       if done then call symput('max_i',put(max_i,4.));
Line09.    end;
Line10. run;
Line11.
Line12. %macro main(max_i);
Line13.    data mydir.srace_new (drop=i race);
Line14.       array smrace {*} $1 smrace1-smrace&max_i;
Line15.
Line16.       do until (last.irank5);
Line17.          set mydir.pop_smrace (rename=(smrace=race));
Line18.          by survest irank1 irank2 irank3 irank4 irank5;
Line19.          i=sum(i,1);
Line20.          smrace{i} = race;
Line21.       end;
Line22.       output;
Line23.    run;
Line24. %mend main;
Line25. %main(&max_i);
```

The above code first calculates **max_i**, which gets passed as a parameter during runtime in the macro **main** to define the dimension of array SMRACE. Below are the detailed SAS DATA step program controls (PC):

PC starts at line one (Line01) where the libname gets assigned. PC goes to line two (Line02) where a DATA step _null_ gets started. At line three (Line03), a DOW-loop is called with a **last.irank5** break-event. Line four (Line04) reads SAS dataset **POP_SMRACE** using line five (Line05) BY group variables. Variable "I" gets incremented to "1" and max_i is set to "1". Keep looping through the data until you reach the end of the POP_SMRACE file. When that happens, line eight (Line08) calls **symput**, which stores a character value in a SAS macro variable (**max_i**).

PC goes to line twelve (Line12) where MACRO **main** gets defined (Lines12 to Line24). PC moves to line twenty-five (Line25) where the value of **max_i** is fed to macro **main**. Line thirteen (Line13) is the start of the DATA step and line fourteen (Line14) defines an array **SMRACE** with elements **smrace1-smrace&max_i**. Line sixteen (Line16) is the start of the second DOW-loop and line seventeen (Line17) reads in the POP_SMRACE SAS dataset after renaming variable smrace to race. This step is important since we need the transposed the SAS dataset to have variables smrace1 through smrace{max_i} in an array called smrace{max_i}. Line nineteen (Line19) is similar to line six (Line06) and line twenty (Line20) fills array SMRACE. At the end of each BY group statement, one output statement executed at line twenty-two (Line22). The DOW-loop terminates when the last observation is read from line seventeen (Line17).

The old code for the above example took **1.76 real (0.09 cpu)** seconds to execute, while the double DOW-loop method took only **0.41 real (0.08 cpu)** seconds, which is about **429** (or **11**) **percent** faster and more efficient in **real time** (or **cpu time**) measure. Once again, the Dow-loop method beats the old code method for the **third** time.

CONCLUSION

The power and beauty of the DOW-loop construct is being realized more and more as SAS users begin to take a second look at their existing code. DATA step BY-processing is a fundamental programming tool that has been in practice for the past three or so decades. Once the DOW-loop construct has been understood, SAS programmers usually get excited about adopting it as the method of choice. When something clicks in one's mind about the DOW-loop's logic, certain ideas start popping out of one's mind for the purpose of improving existing SAS code that has been around for a long time. Changing the SAS code requires a great deal of understanding of the existing code relative to the simplified DOW-loop equivalent.

REFERENCES

- *In Lockstep with the DoW-Loop, Paul M. Dorfman, Independent SAS Consultant, Jacksonville, FL, and Lessia S. Shajenko, Senior Quantitative Analyst, Bank of America, Boston, MA, SESUG 2011, Step by Step,* Paper SS-01, http://analytics.ncsu.edu/sesug/2011/SS01.Dorfman.pdf, 10/2011.
- *The DOW-Loop Unrolled, Paul M. Dorfman, Independent SAS Consultant, Jacksonville, FL, SESUG 2010, Beyond the Basics,* Paper BB-13, http://analytics.ncsu.edu/sesug/2010/BB13.Dorfman.pdf, 9/2010.
- *The DOW-Loop Unrolled, Paul M. Dorfman, Senior SAS Developer, Red Buffalo, Inc., Jacksonville, FL, and Koen Vyverman, Manager Technical Support & IT/MIS, SAS Netherlands, SAS Global Forum 2009, Beyond the Basics,* Paper 038-2009, http://support.sas.com/resources/papers/proceedings09/038-2009.pdf, 3/2009.
- *The DOW-Loop Unrolled, Paul M. Dorfman, Senior SAS Developer, Red Buffalo, Inc., Jacksonville, FL and Koen Vyverman, Manager Technical Support & IT/MIS, SAS Netherlands, SAS Global Forum 2009, Beyond the Basics,* Paper 038-2009, http://support.sas.com/resources/papers/proceedings09/038-2009.pdf, 3/2009.
- *DATA step Objects as Programming Tools, Paul M. Dorfman, Independent SAS Consultant, Jacksonville, FL, and Koen Vyverman, SAS Netherlands, SUGI 31, Step by Step,* Paper 241-31, http://www2.sas.com/proceedings/sugi31/241-31.pdf, 2005.
- *2 PROC TRANSPOSEs = 1 DATA step DOW-Loop,Nancy Bucken, i3 Statprob, Ann Arbor, Michigan.* Paper CC12, http://www.lexjansen.com/pharmasug/2007/cc/cc12.pdf.
- *The Magnificent DO, Paul M. Dorfman, Independent Consultant, Jacksonville, FL, SESUG 2002, Beyond the Basics,* http://www.devenezia.com/papers/other-authors/sesug-2002/TheMagnificentDO.pdf, 2002.
- *Ian Whitlock. Re: SAS novice question. Archives of the SAS-L listserve, 16 Feb. 2000.* http://www.listserv.uga.edu/cgi-bin/wa?A2=ind0002C&L=sas-l&P=R5155.
- *sasCommunity.org, Do until last.var Discussion page.* http://www.sascommunity.org/wiki/Do_until_last.var.

ACKNOWLEDGMENTS

Special thanks to Ahmed Al Attar, William E Zupko II, and Arumugam Sutha of the U.S. Census Bureau for advice and technical help received during the development of this paper.

CONTACT INFORMATION

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

Fuad J Foty
U.S. Census Bureau
4600 Silver Hill Road, Room 3K460F
Suitland, MD 20746
301-763-5476
fuad.j.foty@census.gov

* *

Appendix A- Example1

```
libname mydir "";

proc sort data=mydir.SUMFILE out=SUMFILE;
     by sflag state county stratum mflag;
run;

data SUMFILE_AGG
     CCS (keep=sflag state county stratum mflag cpos
                    rs mos celig csamp cexp cexp);

     do _n_ = 1 by 1 until (last.mflag) ;
         set SUMFILE;
         by sflag state county stratum mflag;

         csamp = sum (csamp, selected);
         cexpr = sum (cexpr, mos);
         cexp  = sum (cexp,  cpos);
         celig = sum(celig ,1);
     end;

     output CCS;

     do _n_ = 1 by 1 until (last.mflag) ;
         set SUMFILE ;
         by sflag state county stratum mflag;
         output SUMFILE_AGG;
     end ;

run ;
```

Appendix B- Example2

```
libname mydir "";

proc sort data=mydir.hwr; by statecode; run;
proc sort data=mydir.samps; by statecode; run;

data mydir.PARMS;
   keep statecode hu survsamp si rs univ;

   do _n_ = 1 by 1 until (last.statecode);
      set mydir.HWR;
      by statecode;
      univ = sum (univ, 1);
      if statecode eq '72' then do;
         hu = sum (hu, whrf);
      end;
   end;

   if statecode ne '72' then do;
      do _n_ = 1 by 1 until (last.statecode);
         set mydir.SAMPS;
         by statecode;
         hu = sum (hu, hu10);
      end;
   end;

   survsamp = ceil(.01*hu);
   si = univ / survsamp;
   seed = input('7812223'||statecode,9.);
   call ranuni(seed,ran);
   rs = si*ran;
run;
```

Appendix C- Example3

```
libname mydir "";

proc sort data=mydir.pop_smrace;
   by survest irank1 irank2 irank3 irank4 irank5;
run;

data _null_ ;
   do until (last.irank5);
      set mydir.pop_smrace end=done;
      by survest irank1 irank2 irank3 irank4 irank5;
      i=sum(i,1);
      if i gt max_i then max_i = i;
      if done then call symput('max_i',put(max_i,4.));
   end;
run;

%macro main(max_i);
   data mydir.srace_new (drop=i race);
      array smrace {*} $1 smrace1-smrace&max_i;

      do until (last.irank5);
         set mydir.pop_smrace (rename=(smrace=race));
         by survest irank1 irank2 irank3 irank4 irank5;
         i=sum(i,1);
         smrace{i} = race;
      end;
      output;
   run;
%mend main;

%main(&max_i);
```